EMBEDDED SYSTEMS DESIGN AND PROGRAMMING

Course code:AEC024

IV. B.Tech II semester

Regulation: IARE R-16

BY

M. SUGUNA SRI

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

DUNDIGAL, HYDERABAD - 500 043

CO's	Course outcomes
CO1	Understand the basic concepts of embedded system and various applications and characteristics, formalisms for system design of embedded system design
CO2	Discuss the concepts of C and develop the C programming examples with Keil IDE, and understand the concepts of interfacing modules using embedded C.
CO3	Understand the basic embedded programming concepts in C and assembly language.

COs	Course Outcome
CO4	Understand the fundamentals of RTOS and its programming and Task communication, Task synchronization with its issues and techniques. Develop examples using embedded software and understand the debugging techniques.
CO5	Discuss the concepts of advanced processors like ARM and SHARC and protocols of I2C and CAN bus.

UNIT-I

EMBEDDED COMPUTING

CLOs	Course Learning Outcome
CLO1	Understand basic concept of embedded systems.
CLO2	Analyze the applications in various domains of embedded system.
CLO3	Develop the embedded system and Design process and tools with examples.
CLO4	Understand characteristics and quality attributes of embedded systems, formalisms for system design.

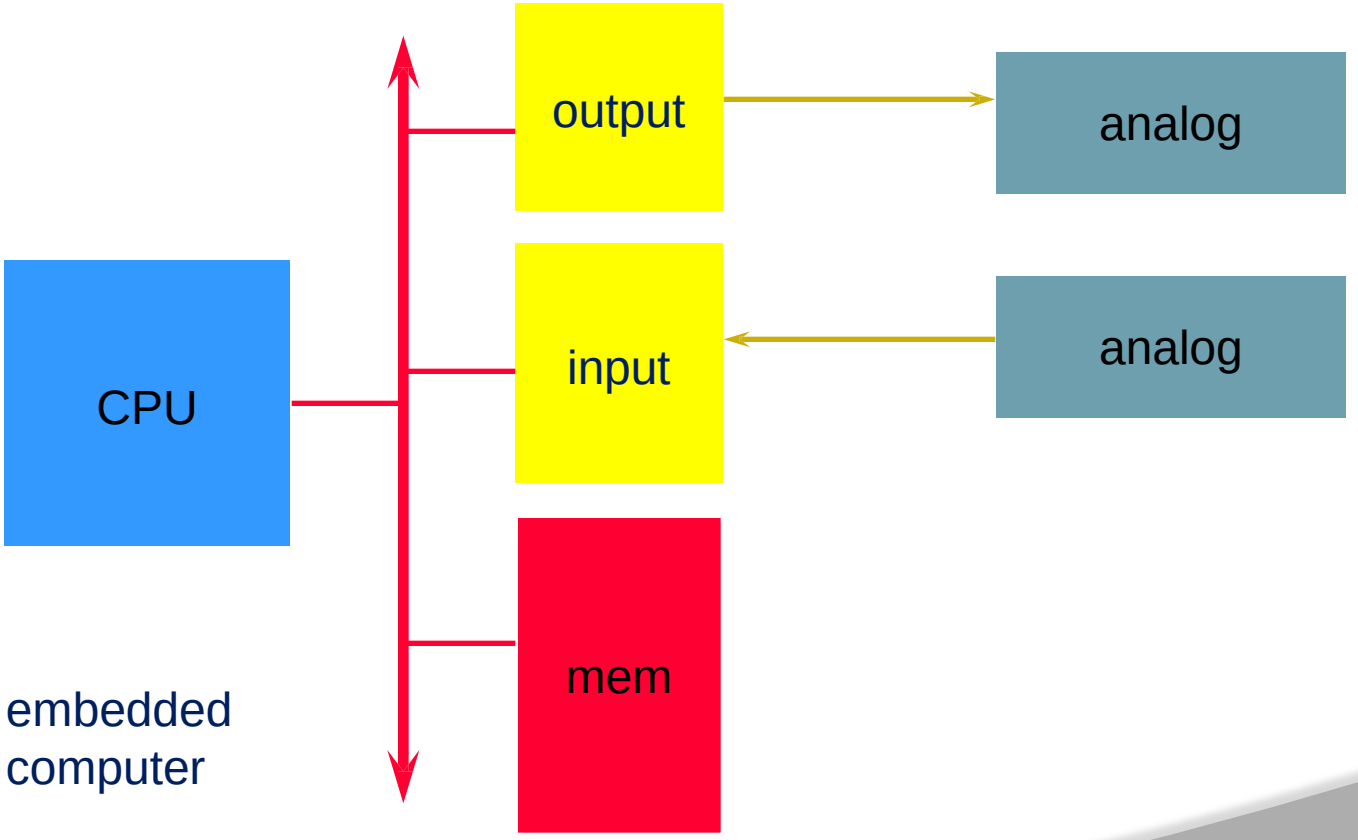
Definition

- It is an Electronic/Electro-mechanical system designed to perform a specific function and is a combination of both hardware & software.

OR

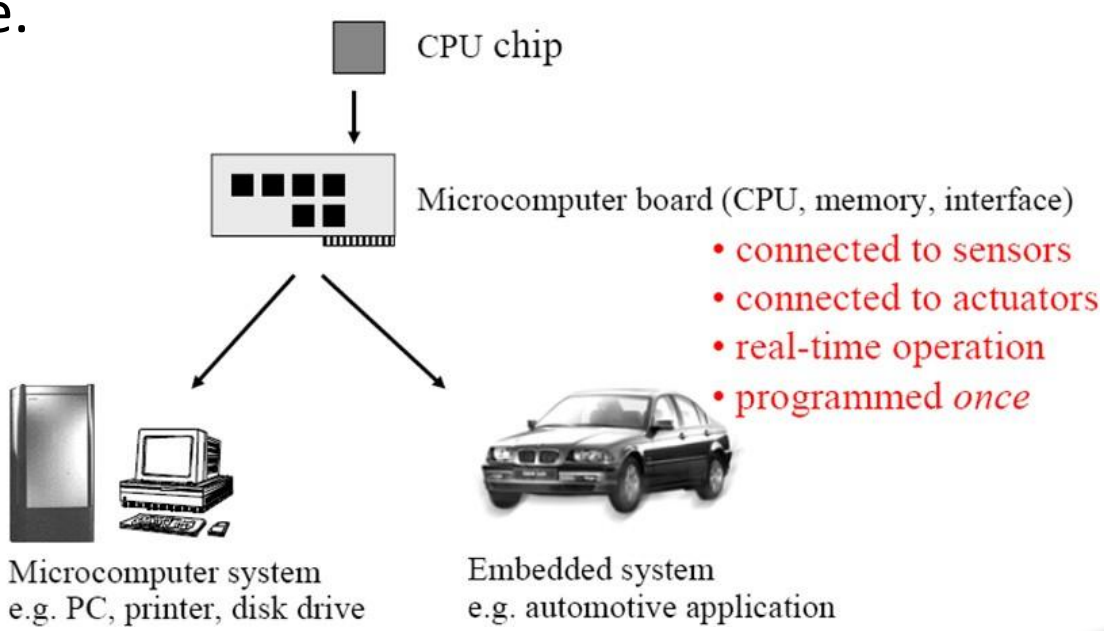
- A combination of hardware and software which together form a component of a larger machine.

Embedding a computer



Example

- An example of an embedded system is a microprocessor that controls an automobile engine.
- An embedded system is designed to run on its own without human intervention, and may be required to respond to events in real time.



Early history

- Late 1940's: MIT Whirlwind computer was designed for real-time operations.
- Originally designed to control an aircraft simulator.
- First microprocessor was Intel 4004 in early 1970's.
- HP-35 calculator used several chips to implement a microprocessor in 1972.

Early history, cont'd.

- Automobiles used microprocessor-based engine controllers starting in 1970's.
- Control fuel/air mixture, engine timing, etc.
- Multiple modes of operation: warm-up, cruise, hill climbing, etc.
- Provides lower emissions, better fuel efficiency.

Automotive embedded systems

- Today's high-end automobile may have 100 microprocessors:
- 4-bit microcontroller checks seat belt;
- microcontrollers run dashboard devices;
- 16/32-bit microprocessor controls engine.

BMW 850i brake and stability control system

- Anti-lock brake system (ABS): pumps brakes to reduce skidding.
- Automatic stability control (ASC+T): controls engine to improve stability.
- ABS and ASC+T communicate.
  - ABS was introduced first---needed to interface to existing ABS module.

Embedded Systems Vs General-Purpose Systems

- Embedded System is a special-purpose computer system designed to perform one or a few dedicated functions -- Wikipedia
  - In general, it does not provide programmability to users, as opposed to general purpose computer systems like PC
  - Embedded systems are virtually everywhere in your daily life

Embedded Systems (Contd)

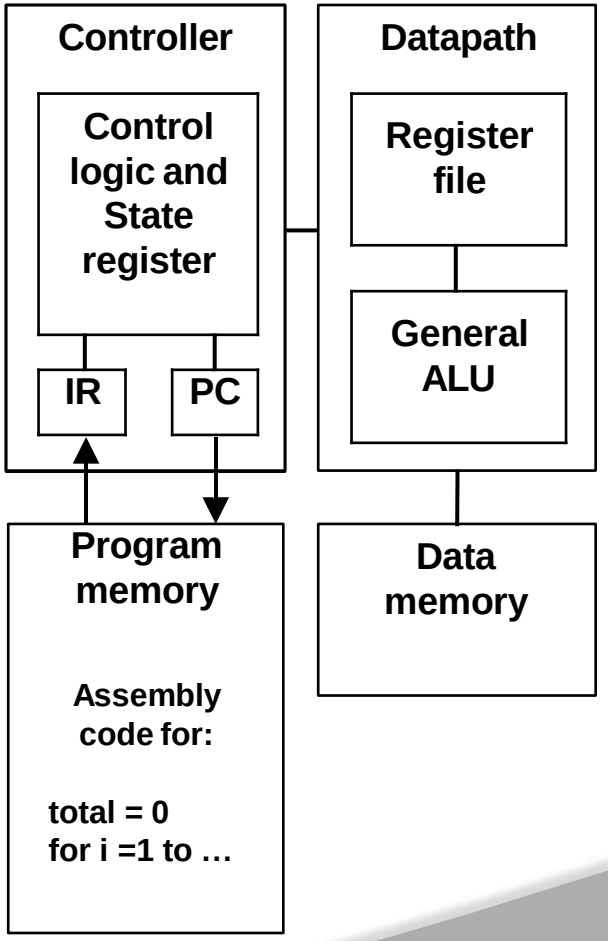
- Even though embedded systems cover a wide range of special-purpose systems, there are common characteristics
  - Low cost
    - Should be cheap to be competitive
      - Memory is typically very small compared to a general purpose computer system
      - Lightweight processors are used in embedded systems
  - Low power
    - Should consume low power especially in case of portable devices
    - Low-power processors are used in embedded systems

Embedded Systems (Contd)

- High performance
  - Should meet the computing requirements of applications
    - should be in sync with video
    - Gaming Users want to watch video on portable devices
- Real-time property
  - Job should be done within a time limit
  - Aerospace applications, Car control systems,
  - Medical gadgets are critical in terms of time

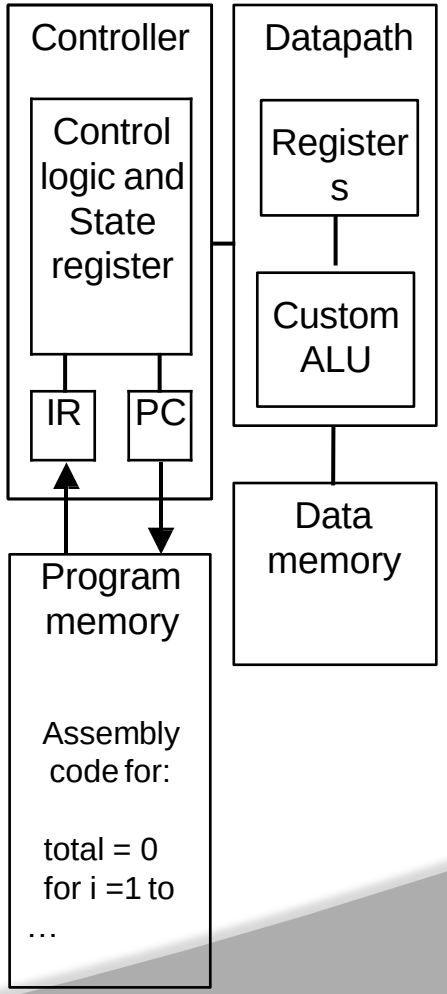
General-purpose processors

- Programmable device used in a variety of applications
  - Also known as “microprocessor”
- Features
  - Program memory
  - General data path with large register file and general ALU
- User benefits
  - Low time-to-market and NRE costs
  - High flexibility



Application-specific processors

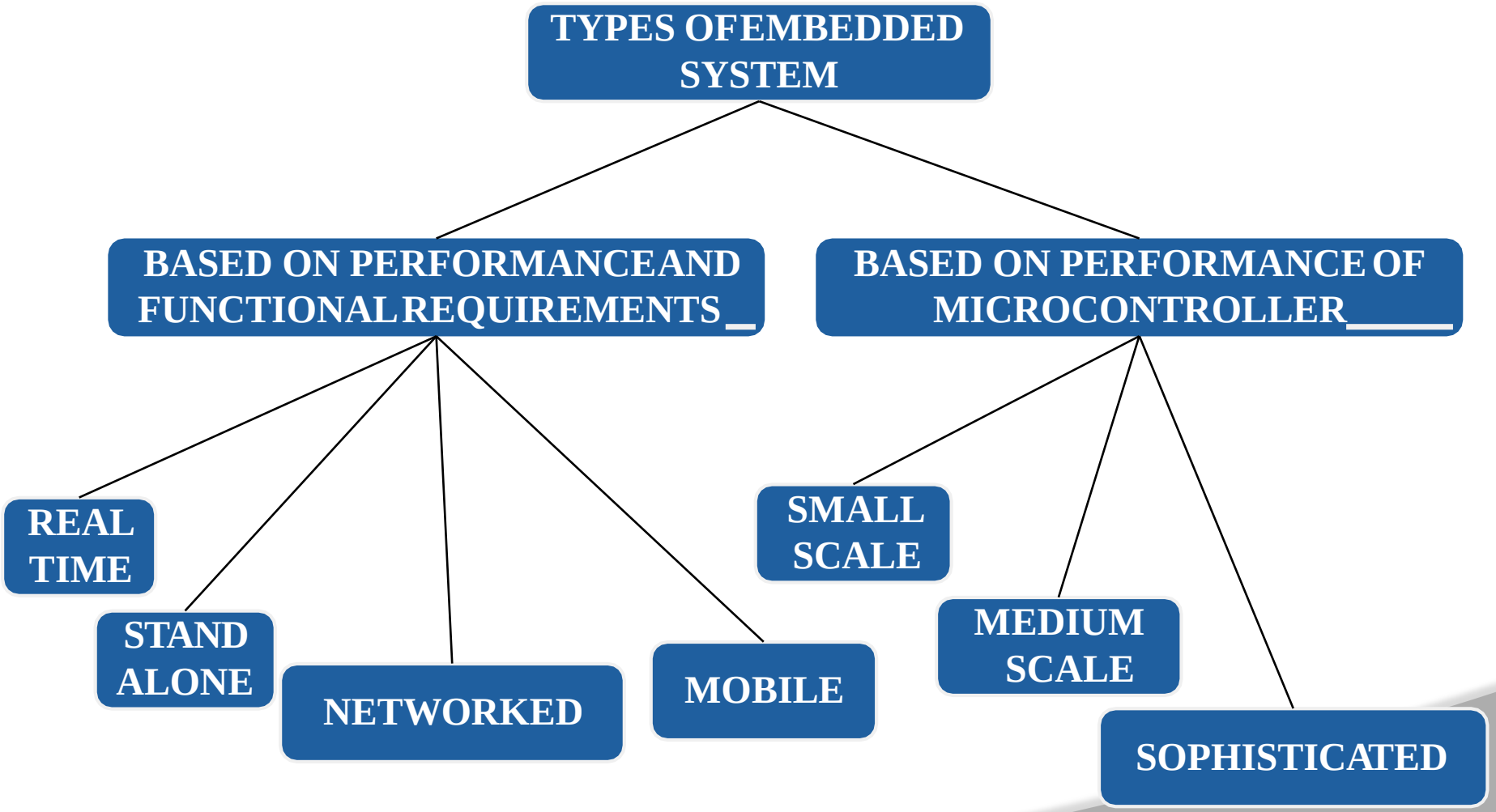
- Programmable processor optimized for a particular class of applications having common characteristics. Compromise between general-purpose and single-purpose processors
- Features
  - Program memory
  - Optimized datapath
  - Special functional units
- Benefits
  - Some flexibility, good performance, size and power



General Computer Purpose VS Embedded system

Criteria	General Computer Purpose	Embedded system
Contents	It is combination of generic hardware and a general purpose OS for executing a variety of applications.	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" factor is key	Application specific requirements are key factors.
Power Consumption	More	Less
Response Time	Not Critical	Critical applications for some

Classification of Embedded Systems



BASED ON PERFORMANCE AND FUNCTIONAL REQUIREMENT

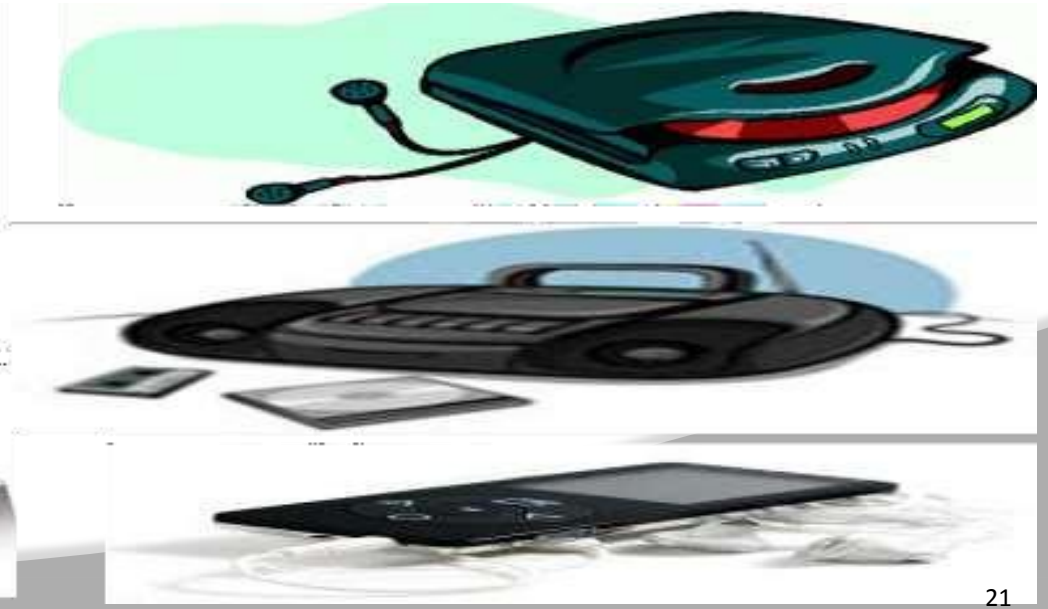
1.REAL TIME EMBEDDED SYSTEM

Real-time embedded systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced.

- Hard real-time systems (e.g., Avionic control).
- Firm real-time systems (e.g., Banking).
- Soft real-time systems (e.g., Video on demand).

2.STAND ALONE EMBEDDED SYSTEM

- A standalone device is able to function independently of other hardware. This means it is not integrated into another device.
- It takes the input from the input ports either analog or digital and processes, calculates and converts the data and gives the resulting data through the connected device-Which either controls, drives and displays the connected devices.
- For example, a TiVo box that can record television programs , mp3 players are standalone devices



3.NETWORKED EMBEDDED SYSTEM

- These types of embedded systems are related to a network to access the resources.

- The connected network can be LAN, WAN or the internet. The connection can be any wired or wireless. This type of embedded system is the fastest growing area in embedded system applications. .

4.MOBILE EMBEDDED SYSTEMS

- Mobile embedded systems are used in portable embedded devices like cell phones, mobiles, digital cameras, mp3 players and personal digital assistants, etc.
- The basic limitation of these devices is the other resources and limitation of memory.

Major Application Areas Of Embedded Systems

1. **Consumer Electronics**
   - Camcorders, Cameras, etc…
2. **Household Appliances**
   - Television, DVD Player, Washing machine, fridge, microwave oven, etc.
3. **Home automation and security system**
   - Air conditioners, Sprinkler, intruder detection alarms, fire alarms, closed circuit television cameras, etc
4. **Automotive industry**
   - Anti-lock breaking system (ABS), engine control, ignition control, automatic navigation system, etc..
5. **Telecommunication**
   - Cellular telephones, telephone switches, Router, etc…

Major Application Areas Of Embedded Systems

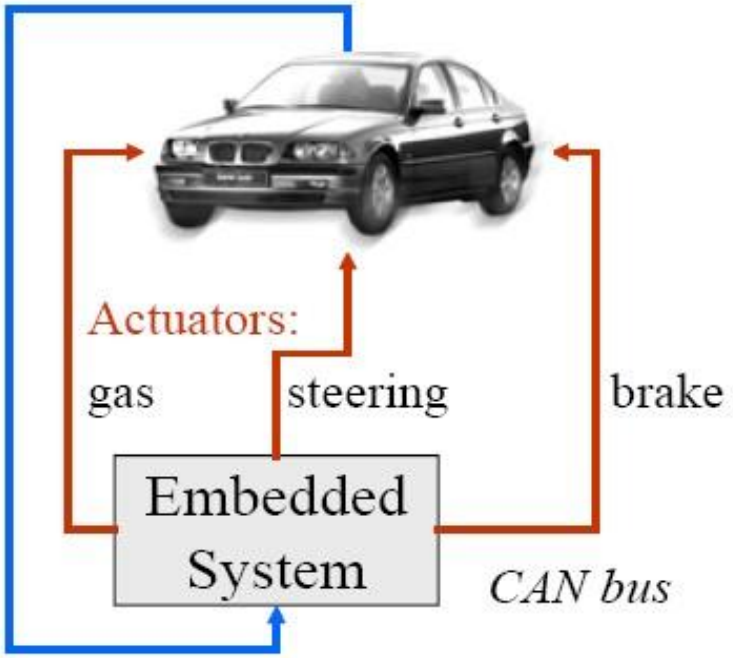
6. **Computer peripherals**
   - Printers, scanners, fax machines, etc…
7. **Computer Networking systems**
   - Network routers, switches, hubs, firewalls, etc…
8. **Health care**
   - CT scanner, ECG , EEG , EMG ,MRI, Glucose monitor, blood pressure monitor, medical diagnostic device, etc.
9. **Measurement & Instrumentation**
   - Digital multi meters, digital CROs, logic analyzers PLC systems, etc…
10. **Banking & Retail**
    - Automatic Teller Machine (ATM) and Currency counters, smart vendor machine, cash register ,Share market, etc..
11. **Card Readers**
    - Barcode, smart card readers, hand held devices, etc…

Automobiles

Autonomous cars:

- Electronic gas
- Electronic brake
- Electronic steering

See: *The Daimler Story*



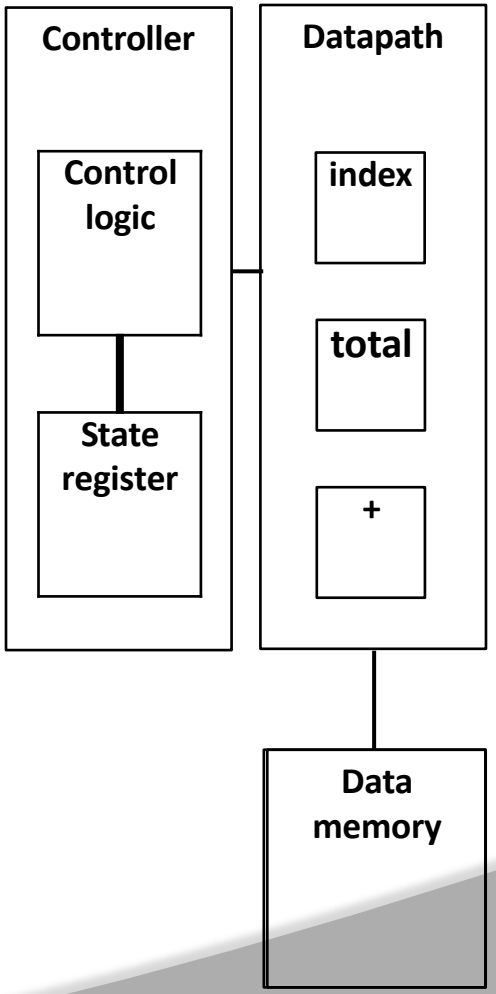
Application Areas

Automobiles

2002: Opel Vectra has over 40 sensors (25 types)

Single-purpose processors

- Digital circuit designed to execute exactly one program
  - a.k.a. coprocessor, accelerator or peripheral
- Features
  - Contains only the components needed to execute a single program
  - No program memory
- Benefits
  - Fast
  - Low power
  - Small size



HW/SW Stack of Embedded Systems

- **Identical to the general-computer systems**

Application Software

VxWorks

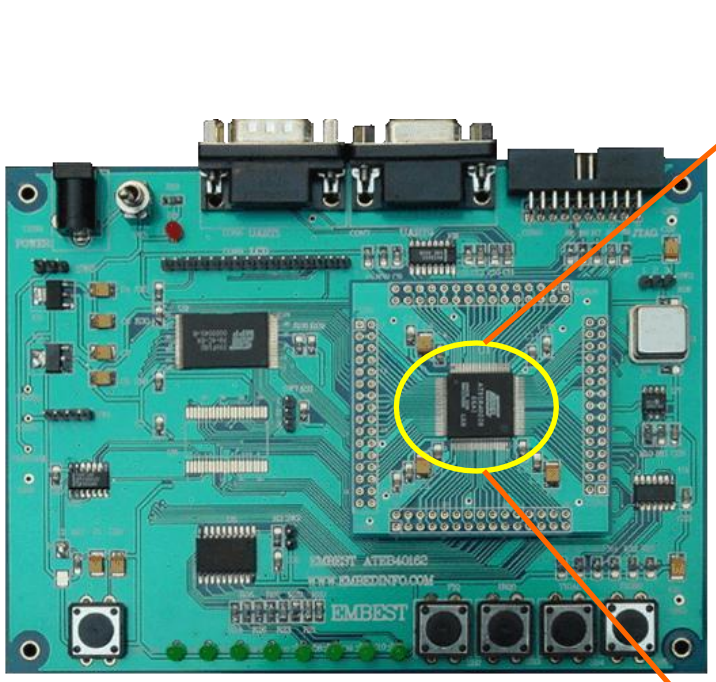
Microsoft Windows Embedded

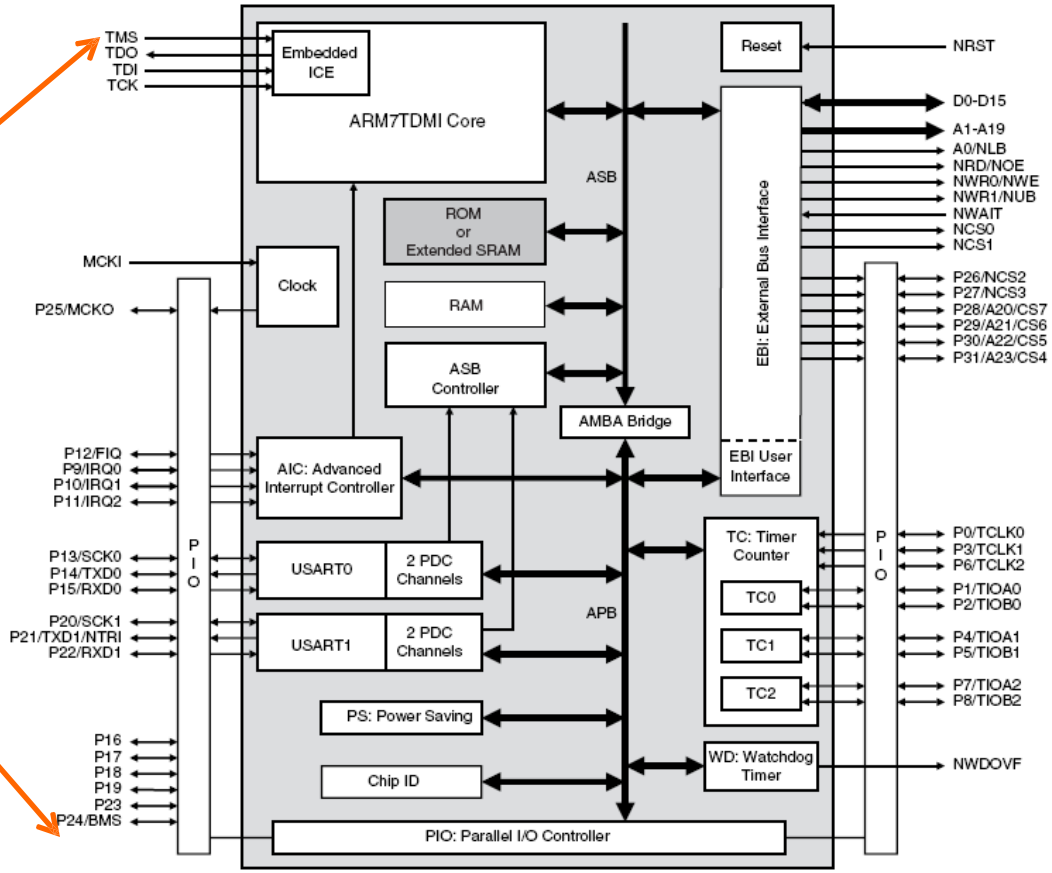
OS / Device Drivers

Hardware

Components of Embedded Systems

- Hardware
  - It is mainly composed of processor (1 or more), memory, I/O devices including network devices, timers, sensors etc.





Components of Embedded Systems

- Software
  - System software
    - Operating systems
  - Many times, a multitasking (multithreaded) OS is required, as embedded applications become complicated
  - Networking, GUI, Audio, Video
  - Processor is context-switched to process multiple jobs
  - Operating system footprint should be small enough to fit into memory of an embedded system
  - In the past and even now, real-time operating systems (RTOS) such as VxWorks or uC/OS-II have been used because they are light-weighted in terms of memory requirement
  - Nowadays, little heavy-weighted OSs such as Windows-CE or embedded Linux (uClinux) are used, as embedded processors support computing power and advanced capabilities such as MMU (Memory Management Unit)

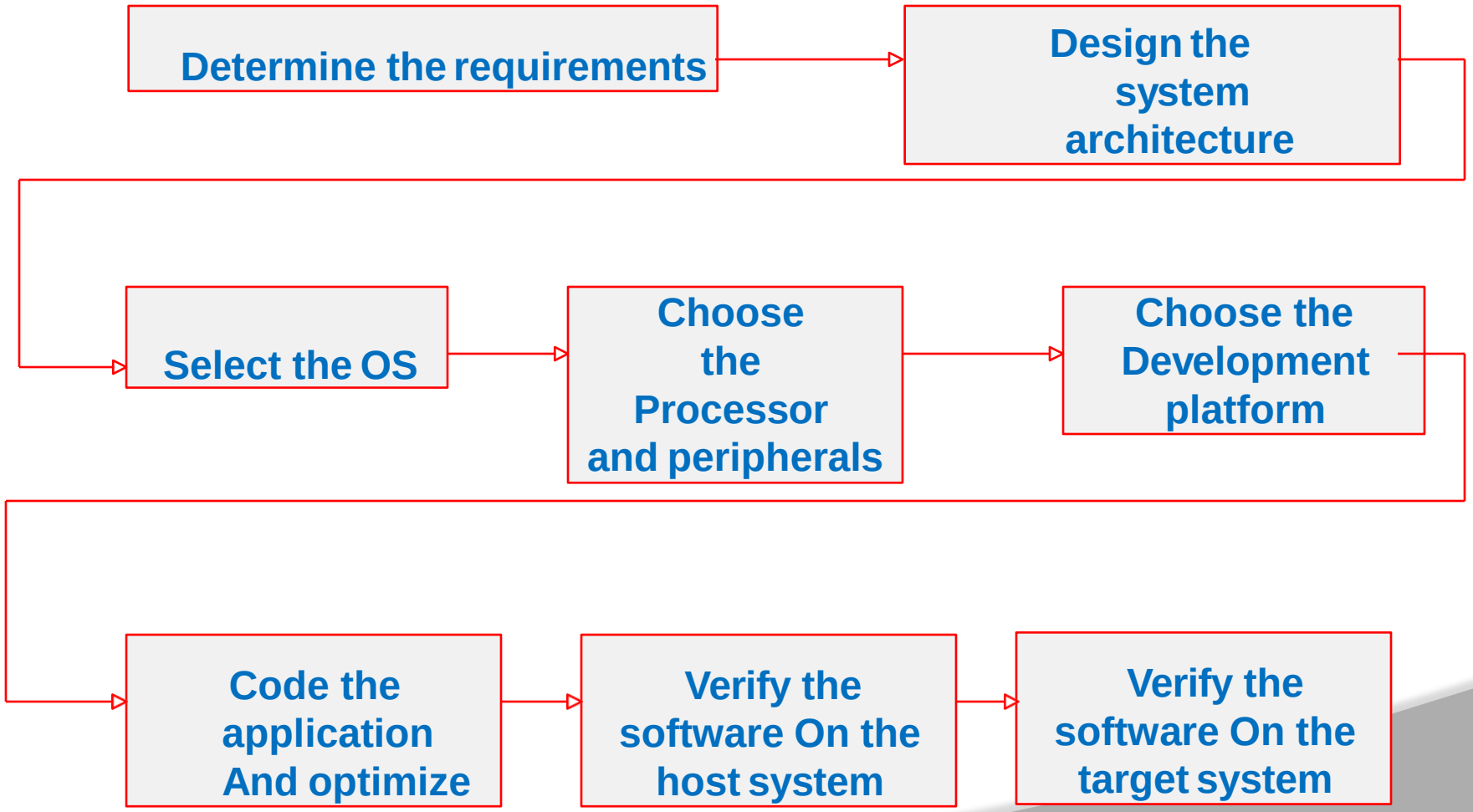
Components of Embedded Systems

- Software (cont.)
  - Device drivers for I/O devices
  - Application software
    - Run on top of operating system
    - Execute tasks that users wish to perform
      - Web surfing, Audio, Video playback

Real-Time System

- Real-time operating system (RTOS)
  - Multitasking operating system intended for real-time applications
  - RTOS facilitates the creation of real-time systems
  - RTOS does not necessarily have a high throughput
  - RTOS is valued more for how quickly and/or predictably it can respond to a particular event
    - Hard real-time systems are required to complete a critical task within a guaranteed amount of time
    - Soft real-time systems are less restrictive
  - Implementing real-time system requires a careful design of scheduler
    - System must have the priority-based scheduling
- Real-time processes must have the highest priority
  - Priority inheritance (next slide)
    - Solve the priority inversion problem
  - Process dispatch latency must be small

EMBEDDED SYSTEM DESIGN PROCESS



REQUIREMENTS

- Functional and non-functional.

  Multi function or Multi mode system.
  Size, cost, Weight etc.

- Selecting the H/W components.
  - Application specific H/W. External interfaces.
  - Input devices. Output devices.

DESIGN SYSTEM ARCHITECTURE

System architecture depends on,

- Whether the system is real time.
- Whether OS needs to be embedded.
- Size, Cost, Power consumption etc.

Choose operating system

If OS needed we can select,

- Real time OS (such as RTLinux,Vx Works, VRTX, pSOS, QNX etc.).
- Non-real time OS (such as Windows CE, embedded Windows XP etc).

Choose processor

We can select any one of the following,

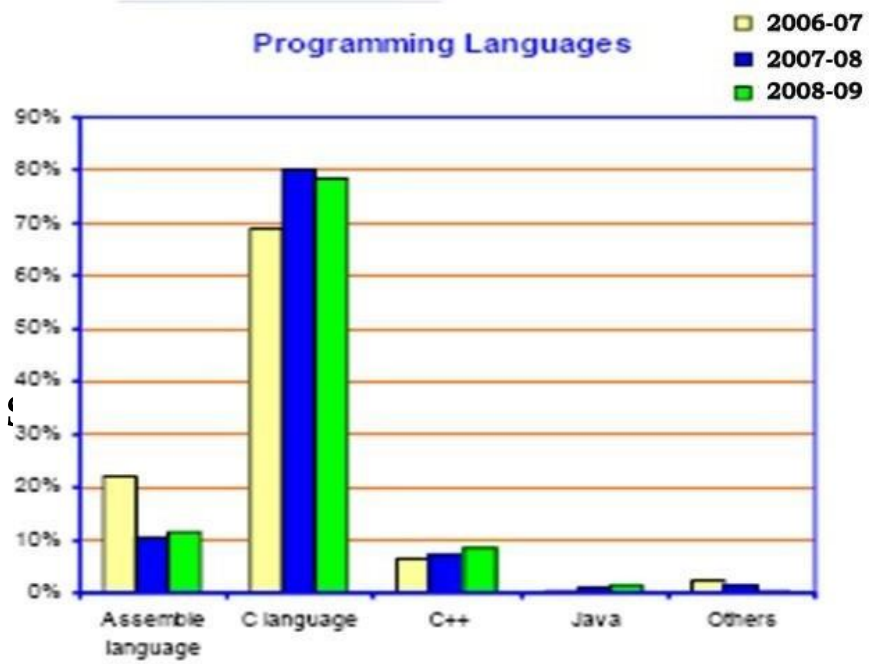
- Microprocessor8085,8086,Pentium
- Microcontroller
- MCS-51,PIC,AVR,MSP430
- Digital Signal Processor
- dsPIC,Blackfin,Sharc,TigerSharc

Development platform

- **The hardware platform.**
- The operating system.
- The programming language.
- The development tools.

Coding the Application

- **Choice of language.**



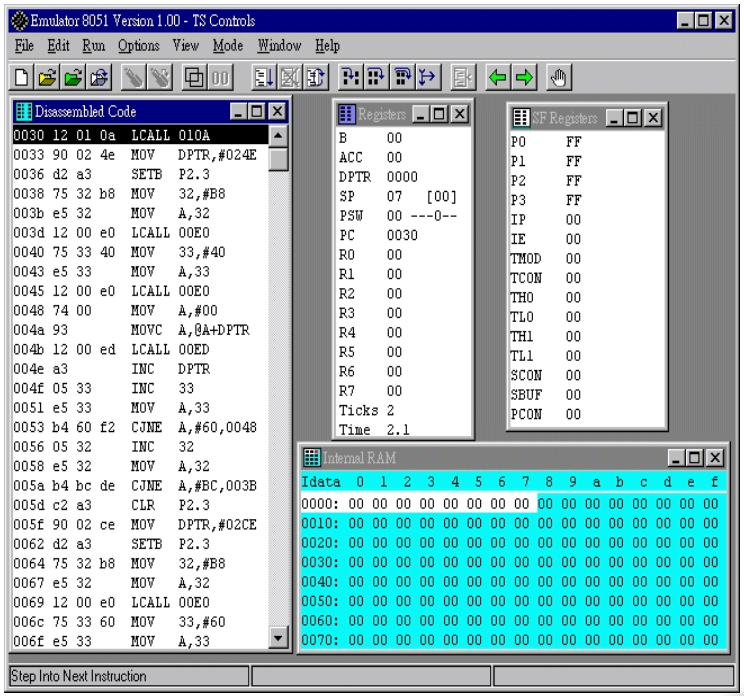
- **Assembly.**
- **C language.**
- **Object Oriented Language (C++, Java etc.).**

Optimizing the code

Verifying the software on the host system

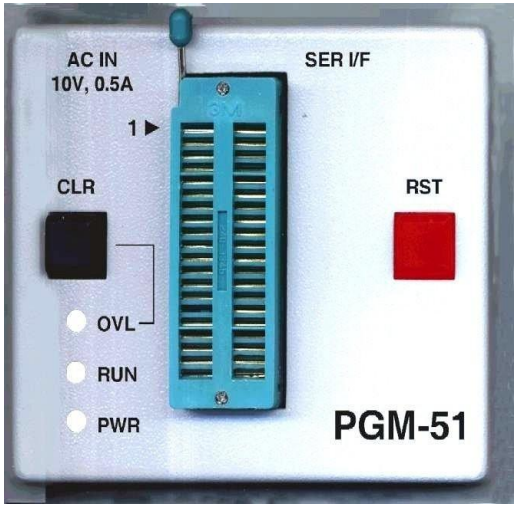
Compile and assemble the source code into object file.

Use a simulator to simulate the working o the system.



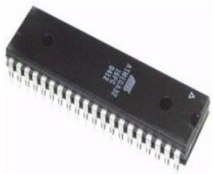
Verifying the software on the target system

- **Download the program using a programmer device.**



- Use an EMULATOR or on chip debugging tools to verify the software.

Recent Trends



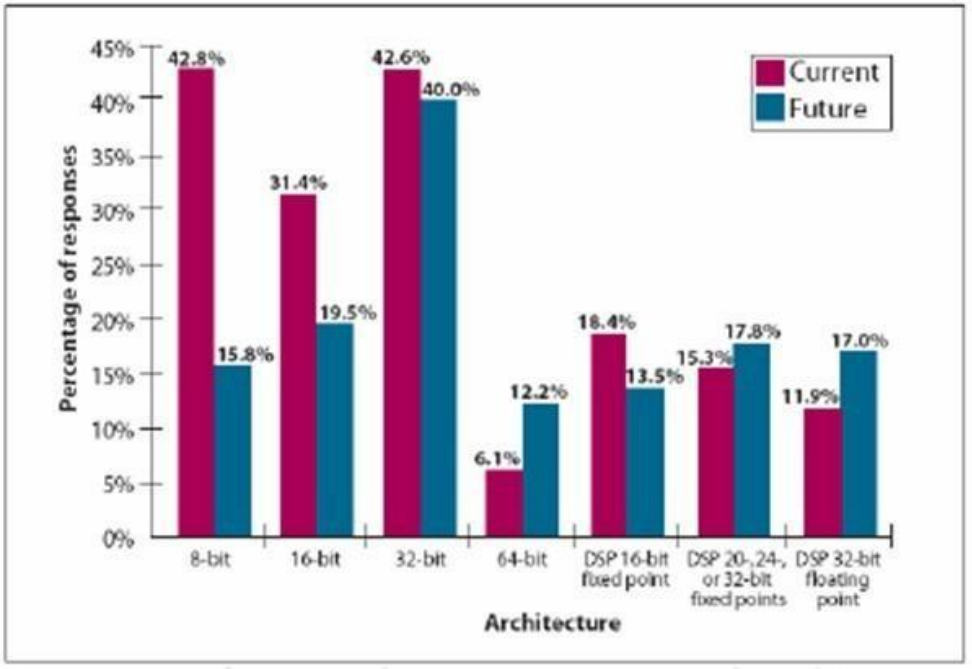
- **Due to the developments in Micro electronics availabilit of processors increased.**

- Reduces cost.

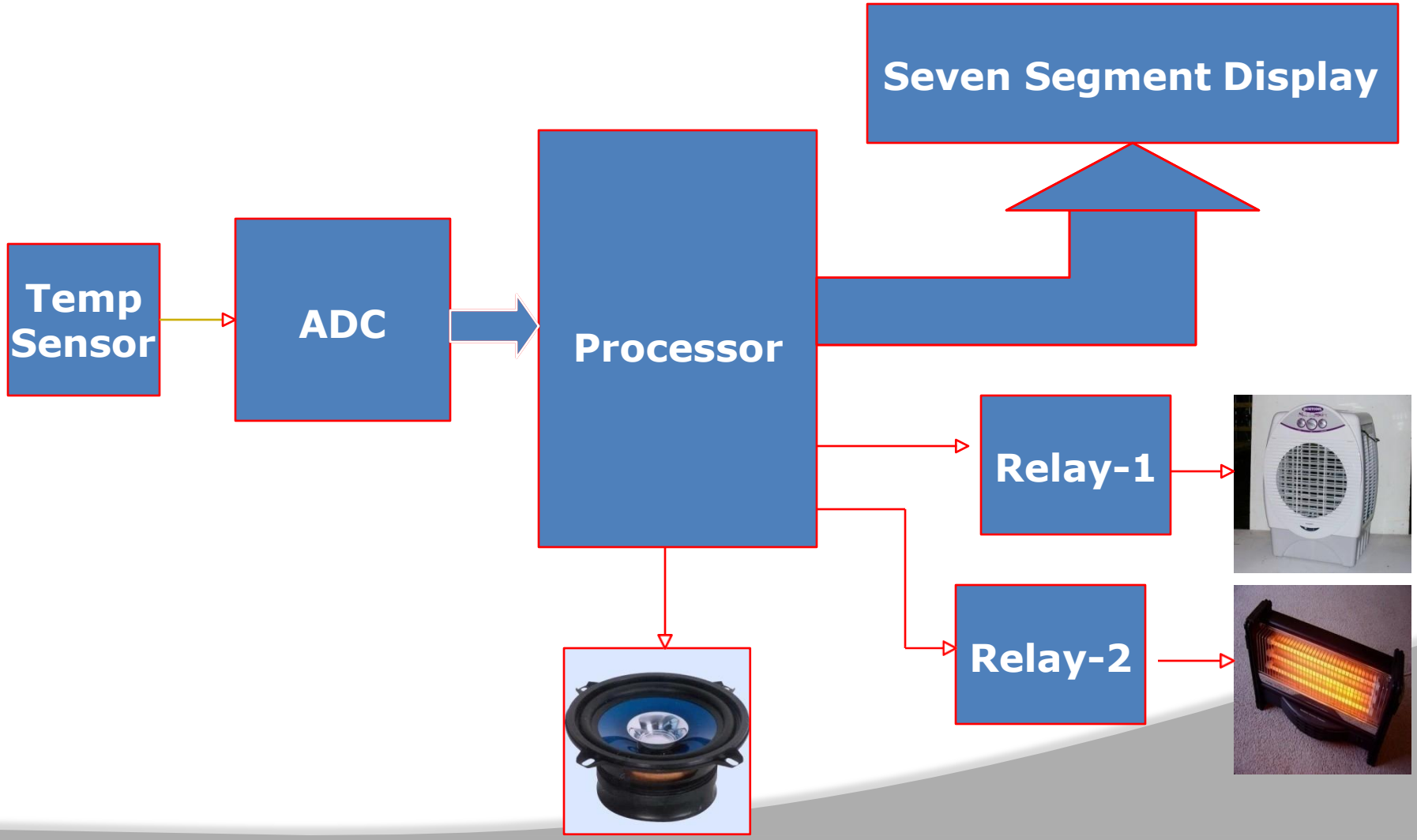
- Increased speed.

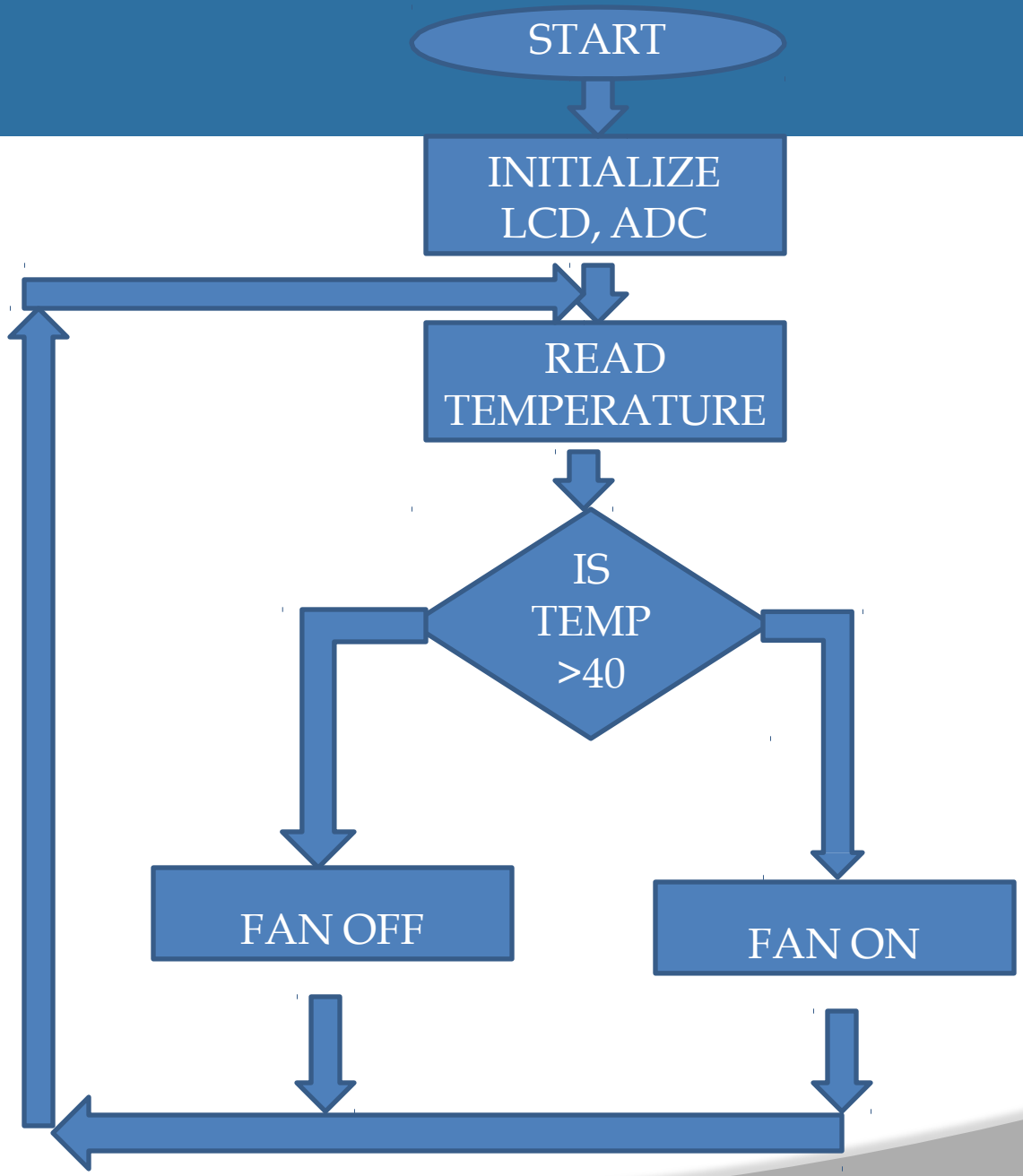
- Reduce Size

- Reduce Power.



Automatic temperature monitoring and control system




START
INITIALIZE LCD, ADC
READ TEMPERATURE
IS TEMP >40
FAN OFF
FAN ON

MODULES AND INDUSTRIAL STANDARD SENSORS USED IN PROJECTS

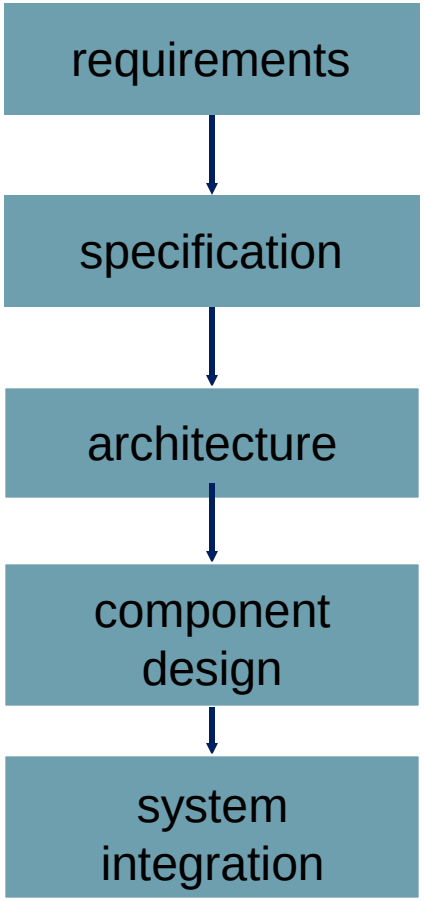
- Pressure Sensors Flow
- Sensors Ultrasonic
- Seansors RF Tx / Rx
- Zigbee Modules EM
- Locks
- Vacuum sensors Digital
- Compass
- CAN IC

- Fire Sensor
- Temperature Sensor
- Speed sensors
- Level sensors
- Industrial proximity sensor
- Vibration sensor
- Water Identifier Sensors
- Acceleration Sensor - 3 Axis
- Glass braking sensor
- Force Sensor

Design goals

- Performance.
  - Overall speed, deadlines.
- Functionality and user interface.
- Manufacturing cost.
- Power consumption.
- Other requirements (physical size, etc.)

Levels of abstraction



Top-down vs. bottom-up

- Top-down design:
  - start from most abstract description;
  - work to most detailed.
- Bottom-up design:
  - work from small components to big system.
- Real design uses both techniques.

Stepwise refinement

- At each level of abstraction, we must:
  - analyze the design to determine characteristics of the current state of the design;
  - refine the design to add detail.

Requirements

- Plain language description of what the user wants and expects to get.
- May be developed in several ways:
  - talking directly to customers;
  - talking to marketing representatives;
  - providing prototypes to users for comment.

Functional vs. non-functional requirements

- Functional requirements:
  - output as a function of input.
- Non-functional requirements:
  - time required to compute output;
  - size, weight, etc.;
  - power consumption;
  - reliability;
  - etc.

Our requirements

name

purpose

inputs

outputs

functions

performance

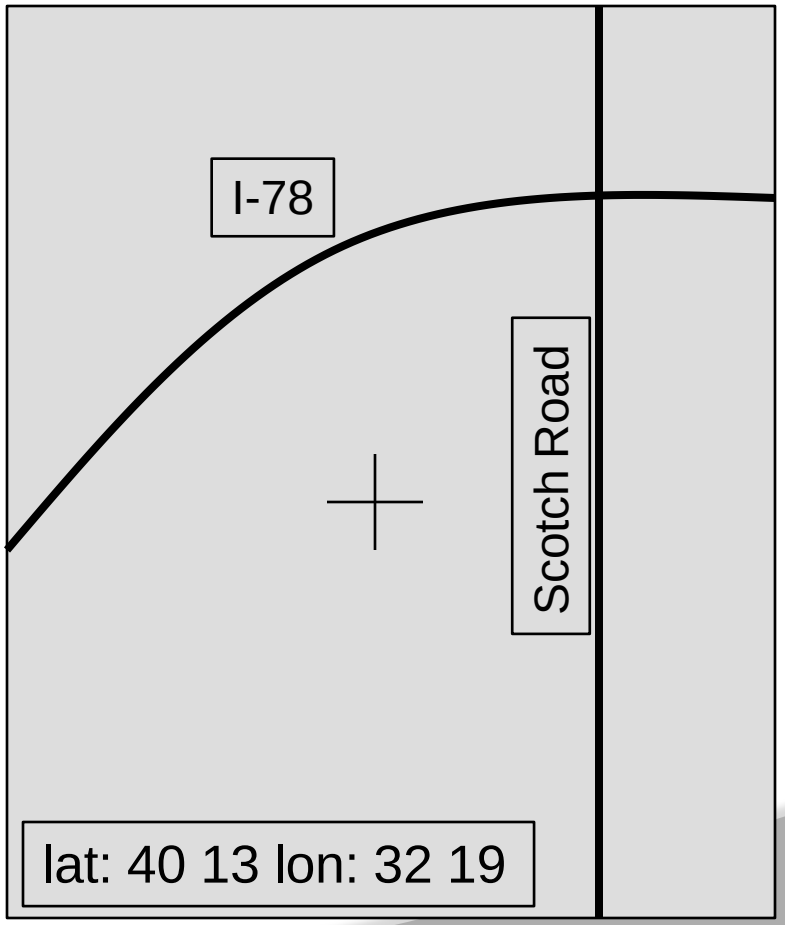
manufacturing cost

power

physical size/weight

Example:
GPS moving map requirements

- Moving map obtains position from GPS, paints map from local database.



GPS moving map needs

- Functionality: For automotive use. Show major roads and landmarks.
- User interface: At least 400 x 600 pixel screen. Three buttons max. Pop-up menu.
- Performance: Map should scroll smoothly. No more than 1 sec power-up. Lock onto GPS within 15 seconds.
- Cost: $120 street price = approx. $30 cost of goods sold.
- Physical size/weight: Should fit in hand.
- Power consumption: Should run for 8 hours on four AA batteries.

Specification

- A more precise description of the system:
  - should not imply a particular architecture;
  - provides input to the architecture design process.
- May include functional and non-functional elements.
- May be executable or may be in mathematical form for proofs.

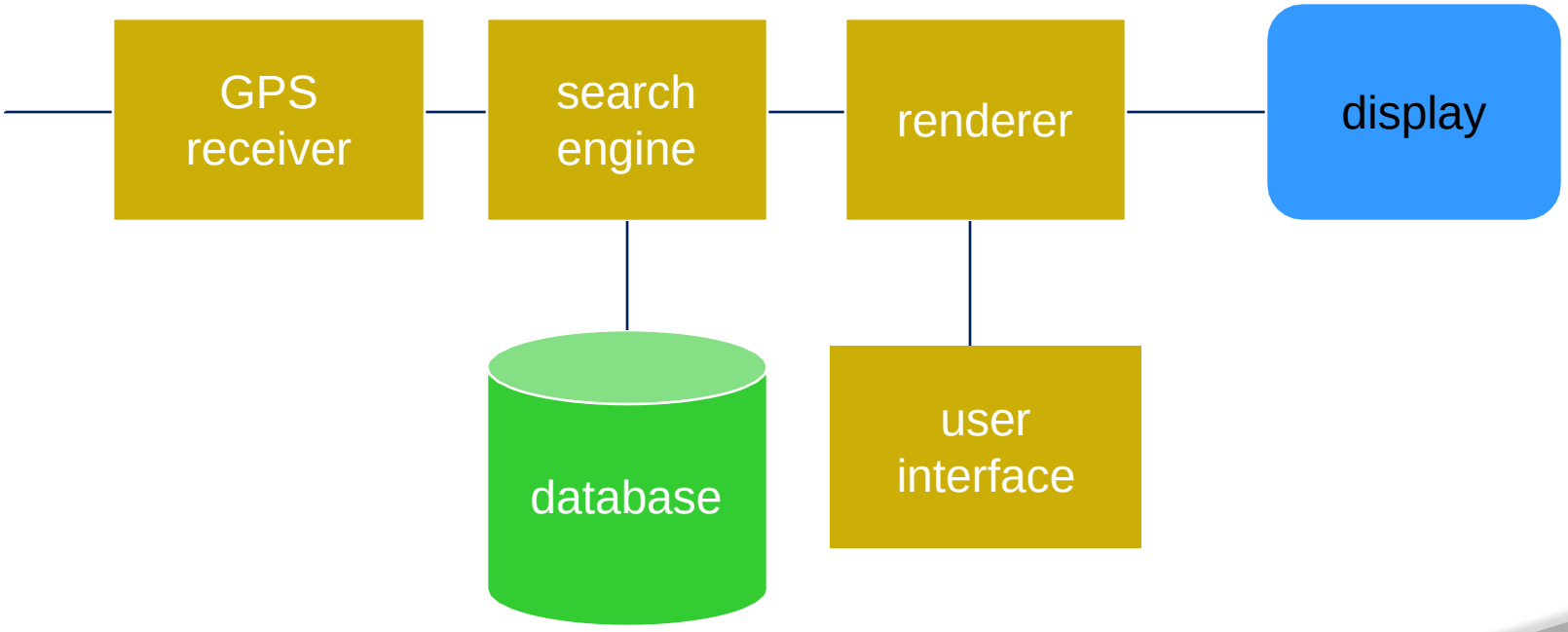
GPS specification

- Should include:
  - What is received from GPS;
  - map data;
  - user interface;
  - operations required to satisfy user requests;
  - background operations needed to keep the system running.

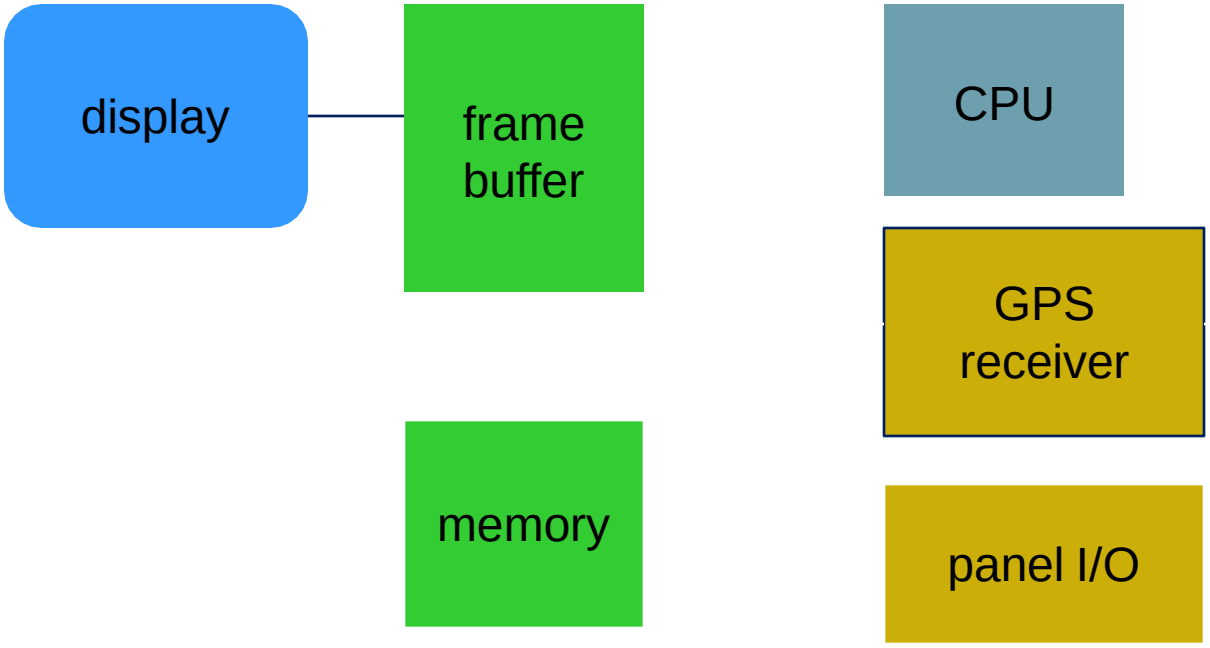
Architecture design

- What major components go satisfying the specification?
- Hardware components:
  - CPUs, peripherals, etc.
- Software components:
  - major programs and their operations.
- Must take into account functional and non-functional specifications.

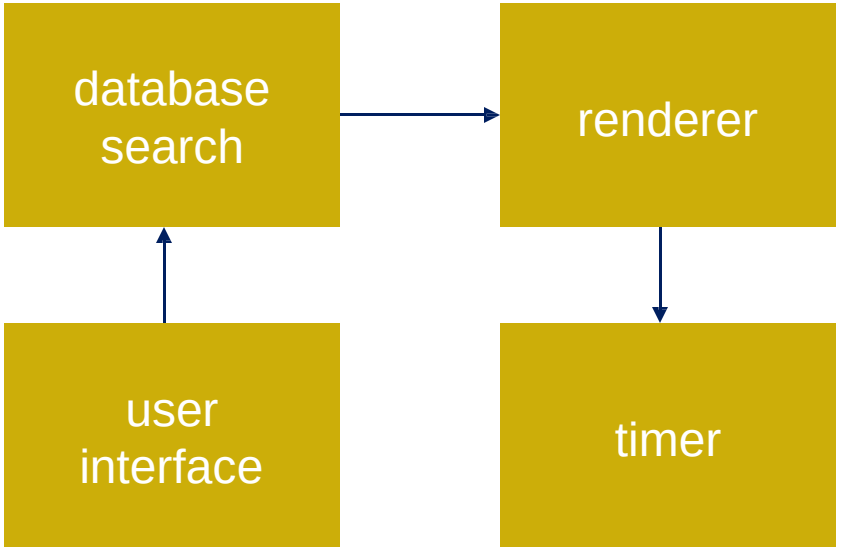
GPS moving map block diagram



GPS moving map hardware



GPS moving map software architecture



Designing hardware and software components

- Must spend time architecting the system before you start coding.
- Some components are ready-made, some can be modified from existing designs, others must be designed from scratch.

QUALITY ATTRIBUTES OF EMBEDDED SYSTEM

- These are the attributes that together form the deciding factor about the quality of an embedded system.
- There are two types of quality attributes are:-

- Operational Quality Attributes.
  1. These are attributes related to operation or functioning of an embedded system. The way an embedded system operates affects its overall quality.

- Non-Operational Quality Attributes.
  1. These are attributes not related to operation or functioning of an embedded system. The way an embedded system operates affects its overall quality.
  2. These are the attributes that are associated with the embedded system before it can be put in operation.

Operational Attributes

- **a)** Response
  - Response is a measure of quickness of the system.
  - It gives you an idea about how fast your system is tracking the input variables.
  - Most of the embedded system demand fast response which should be real-time.

- b) Throughput
  - Throughput deals with the efficiency of system.
  - It can be defined as rate of production or process of a defined process over a stated period of time.
  - In case of card reader like the ones used in buses, throughput means how much transaction the reader can perform in a minute or hour or day.

Operational Attributes

- Reliability
  - Reliability is a measure of how much percentage you rely upon the proper functioning of the system .
  - Mean Time between failures and Mean Time To Repair are terms used in defining system reliability.
  - Mean Time between failures can be defined as the average time the system is functioning before a failure occurs.
  - Mean time to repair can be defined as the average time the system has spent in repairs.
- Maintainability
  - Maintainability deals with support and maintenance to the end user or a client in case of technical issues and product failures or on the basis of a routine system checkup
  - It can be classified into two types

I. Scheduled or Periodic Maintenance
II. Maintenance to unexpected failure

Operational Attributes

Security

- Confidentiality, Integrity and Availability are three corner stones of information security.
- Confidentiality deals with protection data from unauthorized disclosure.
- Integrity gives protection from unauthorized modification.
- Availability gives protection from unauthorized user
- Certain Embedded systems have to make sure they conform to the security measures.
- Ex. An Electronic Safety Deposit Locker can be used only with a pin number like a password.

Safety

- Safety deals with the possible damage that can happen to the operating person and environment due to the breakdown of an embedded system or due to the emission of hazardous materials from the embedded products.

Non Operational Attributes

- Testability and Debug-ability
  - It deals with how easily one can test his/her design, application and by which mean he/she can test it.
  - In hardware testing the peripherals and total hardware function in designed manner
  - Firmware testing is functioning in expected way
  - Debug-ability is means of debugging the product as such for figuring out the probable sources that create unexpected behavior in the total system
- Evolvability
  - For embedded system, the qualitative attribute “Evolvability” refer to ease with which the embedded product can be modified to take advantage of new firmware or hardware technology.

Non Operational Attributes

- Portability
  - Portability is measured of “system Independence”.
  - An embedded product can be called portable if it is capable of performing its operation as it is intended to do in various environments irrespective of different processor and or controller and embedded operating systems.
- Time to prototype and market
  - Time to Market is the time elapsed between the conceptualization of a product and time at which the product is ready for selling or use
  - Product prototyping help in reducing time to market.
  - Prototyping is an informal kind of rapid product development in which important feature of the under consider are develop.
  - In order to shorten the time to prototype, make use of all possible option like use of reuse, off the self component etc.

Non Operational Attributes

- **Per unit and total cost**

  •Cost is an important factor which needs to be carefully monitored. Proper market study and cost benefit analysis should be carried out before taking decision on the per unit cost of the embedded product.

  •When the product is introduced in the market, for the initial period the sales and revenue will be low

  •There won't be much competition when the product sales and revenue increase.

System integration

- Put together the components.
  - Many bugs appear only at this stage.
- Have a plan for integrating components to uncover bugs quickly, test as much functionality as early as possible.

System modeling

- Need languages to describe systems:
  - useful across several levels of abstraction;
  - understandable within and between organizations.
- Block diagrams are a start, but don’t cover everything.

Object-oriented design

- Object-oriented (OO) design: A generalization of object-oriented programming.
- Object = state + methods.
  - State provides each object with its own identity.
  - Methods provide an abstract interface to the object.

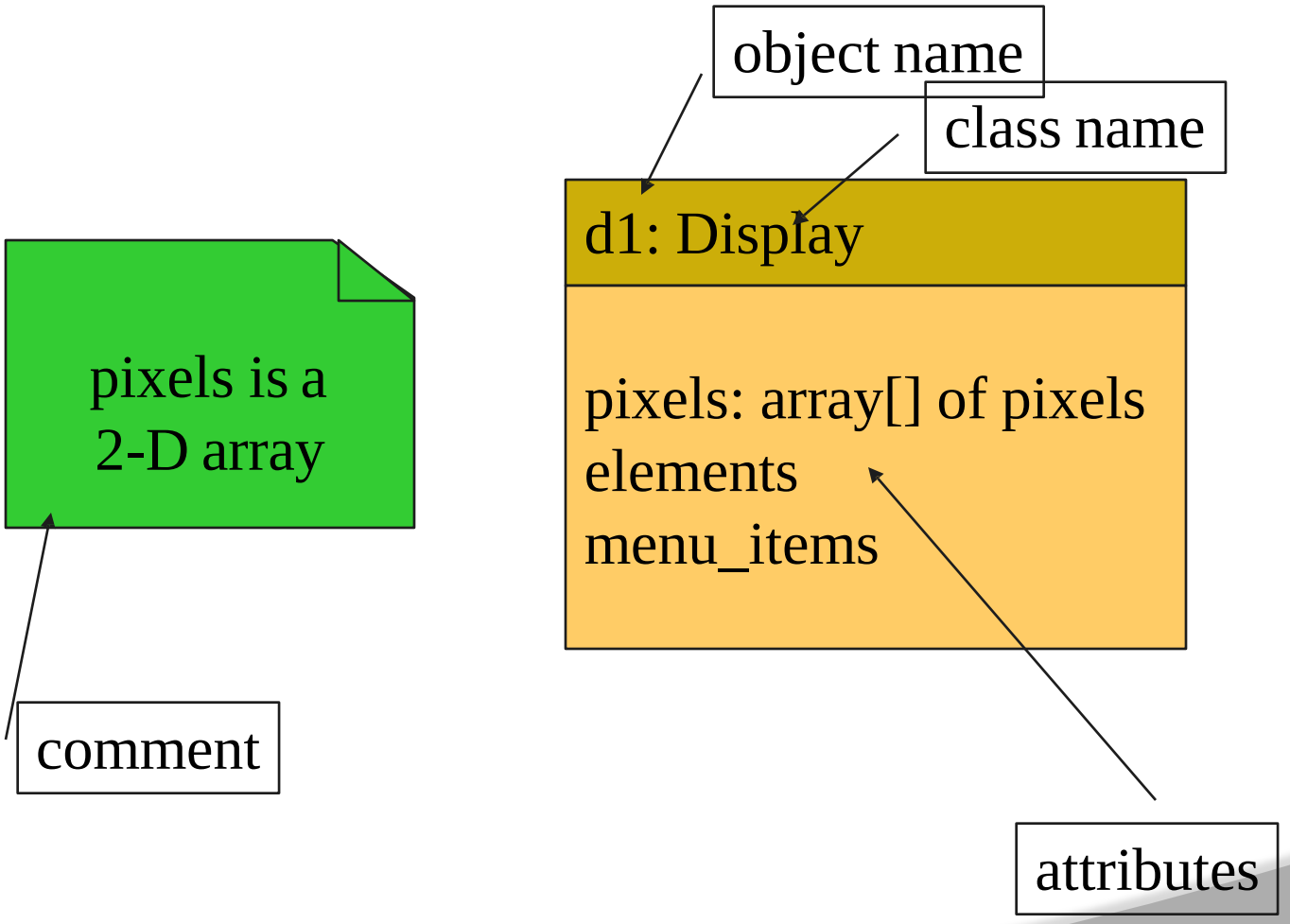
Objects and classes

- Class: object type.
- Class defines the object's state elements but state values may change over time.
- Class defines the methods used to interact with all objects of that type.
  - Each object has its own state.

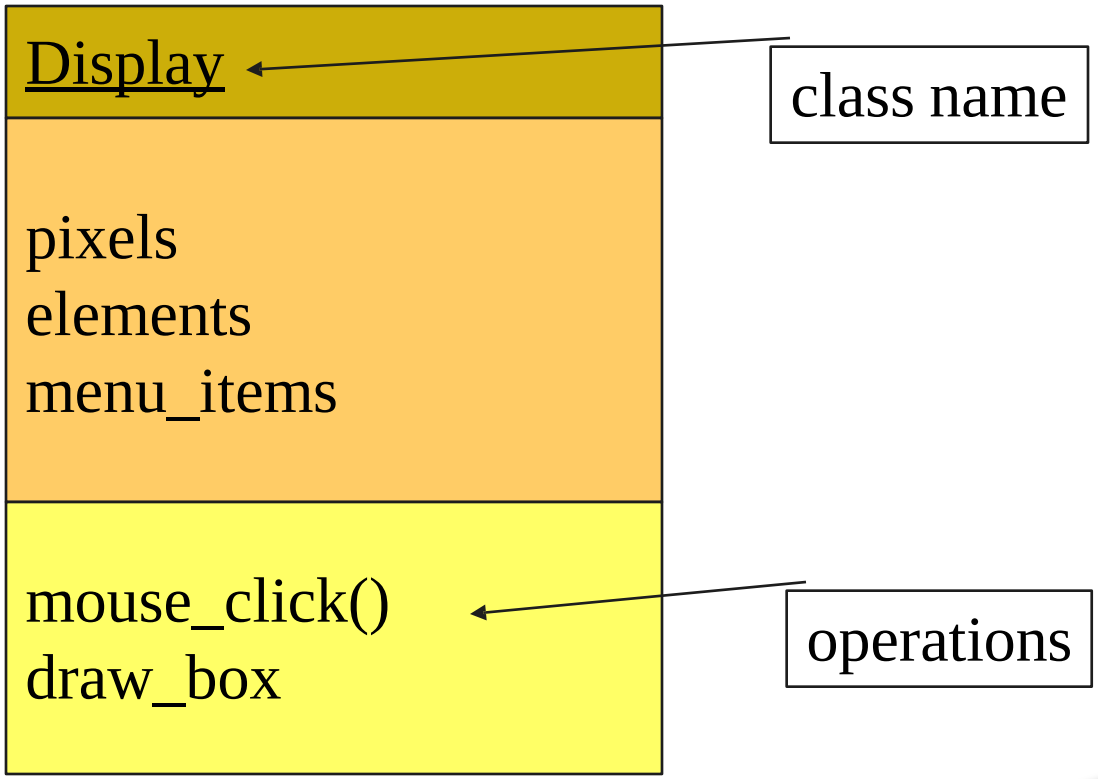
Relationships between objects and classes

- Association: objects communicate but one does not own the other.
- Aggregation: a complex object is made of several smaller objects.
- Composition: aggregation in which owner does not allow access to its components.
- Generalization: define one class in terms of another.

UML object


object name
class name
d1: Display
pixels: array[] of pixels
elements
menu_items
pixels is a
2-D array
comment
attributes

UML class


Display
pixels
elements
menu_items
mouse_click()
draw_box
class name
operations

The class interface

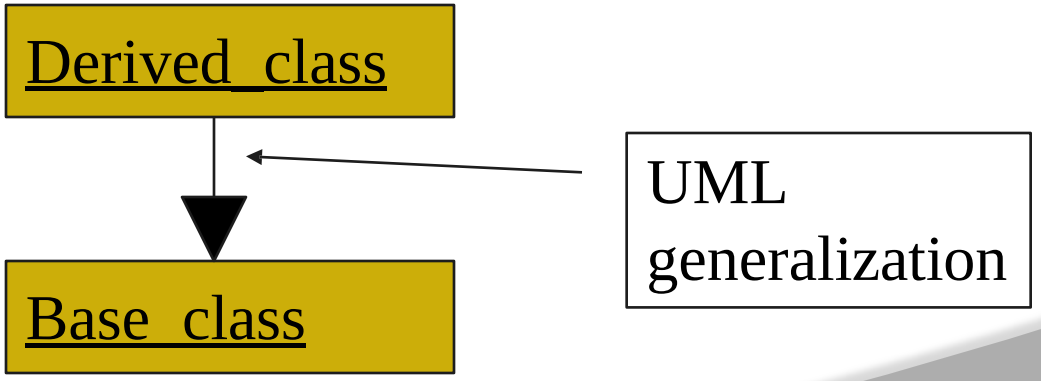
- The operations provide the abstract interface between the class's implementation and other classes.
- Operations may have arguments, return values.
- An operation can examine and/or modify the object's state.

Choose your interface properly

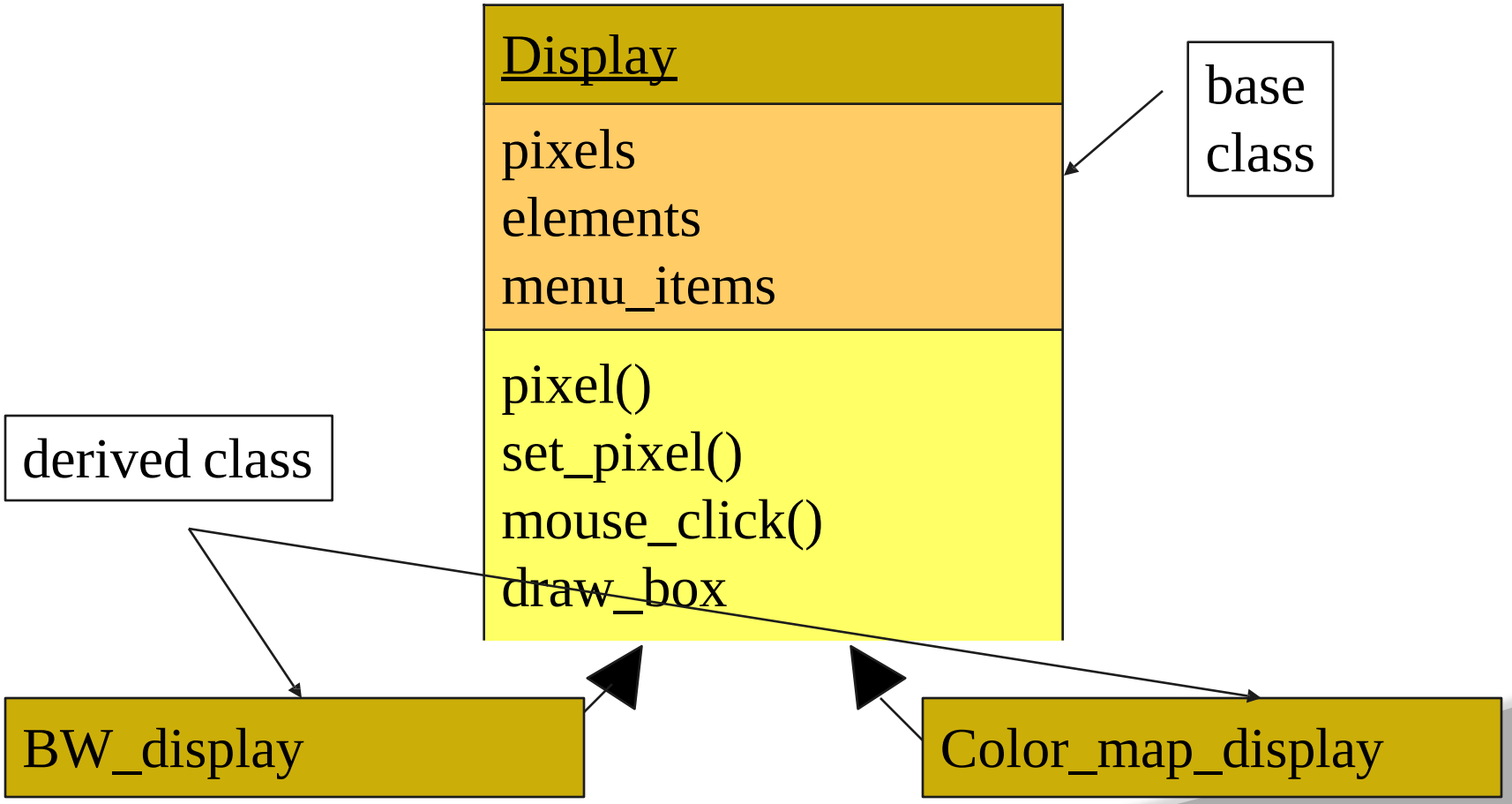
- If the interface is too small/specialized:
  - object is hard to use for even one application;
  - even harder to reuse.
- If the interface is too large:
  - class becomes too cumbersome for designers to understand;
  - implementation may be too slow;
  - spec and implementation are probably buggy.

Class derivation

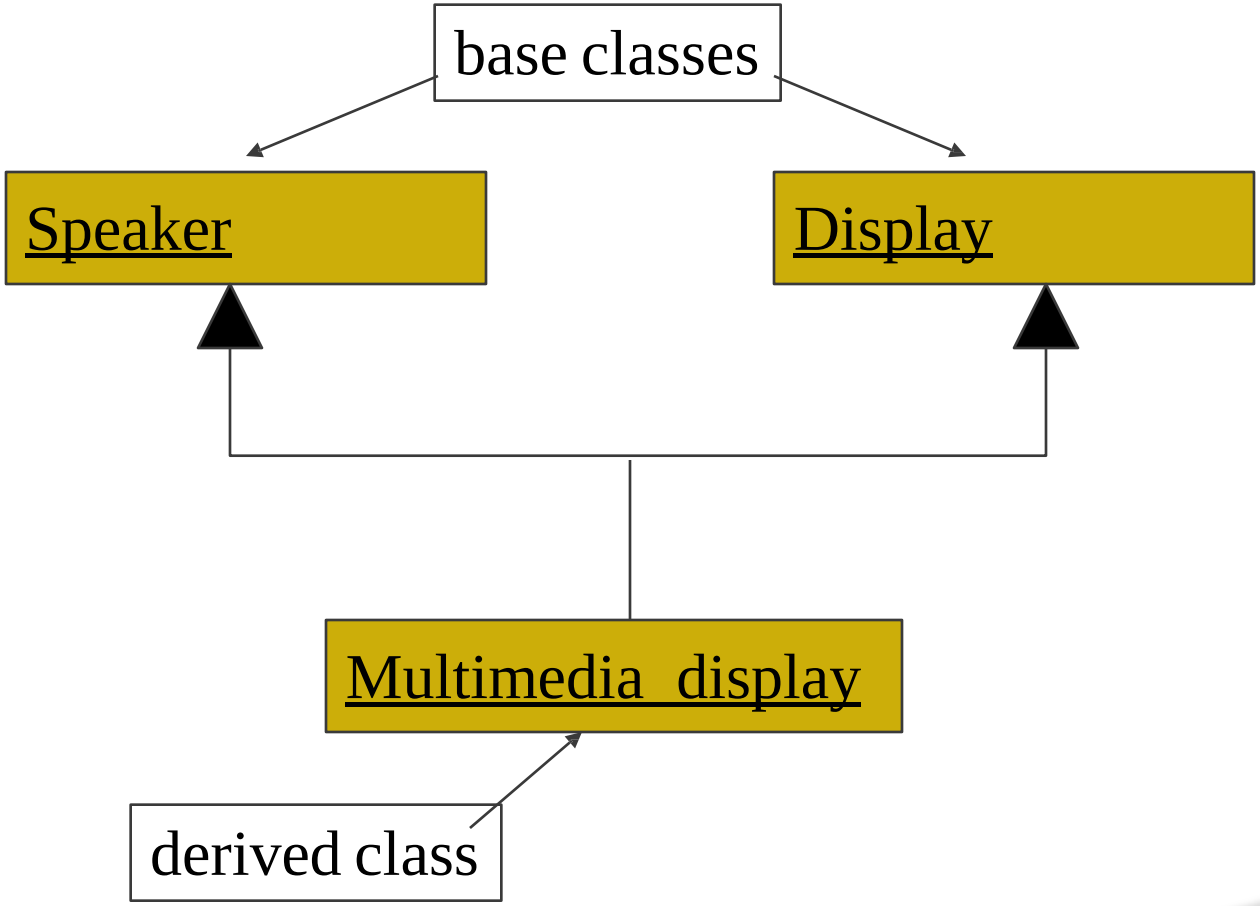
- May want to define one class in terms of another.
  - Derived class inherits attributes, operations of base class.



Class derivation example



Multiple inheritance

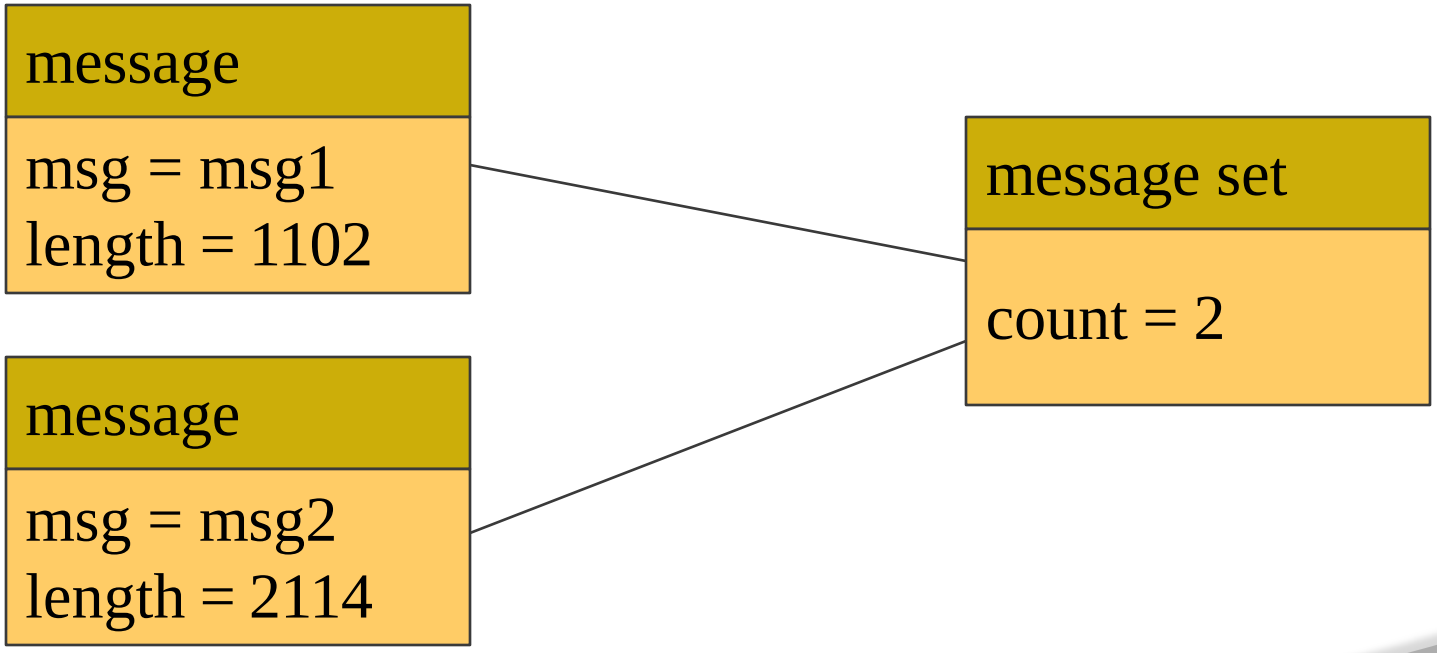


Links and associations

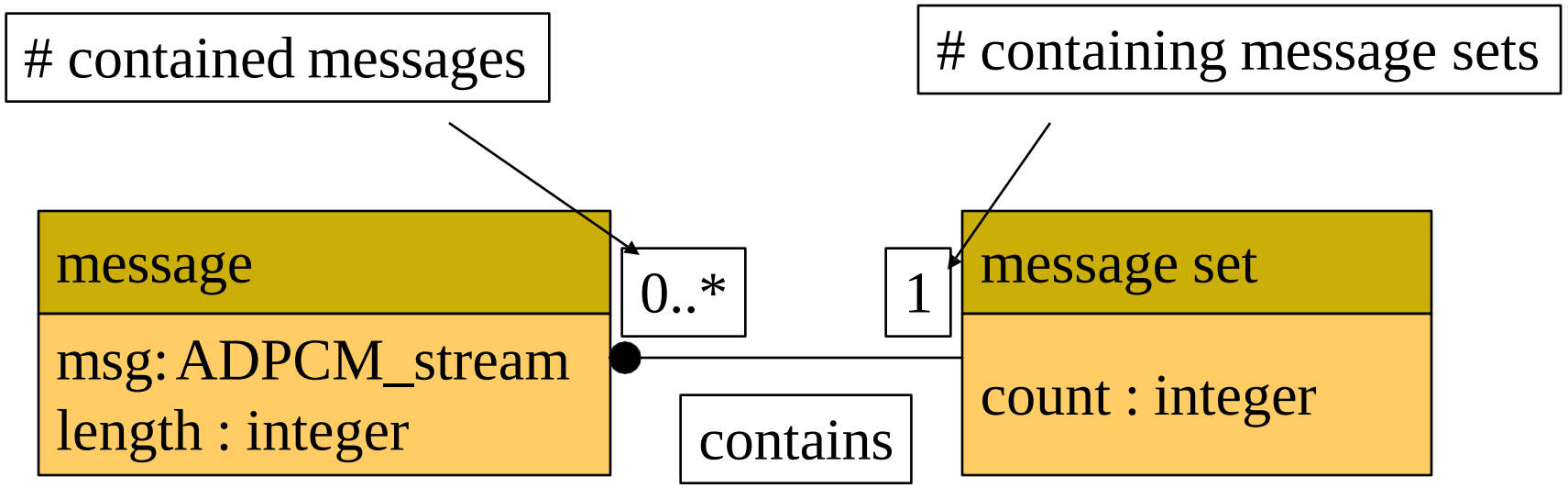
- Link: describes relationships between objects.
- Association: describes relationship between classes.

Link example

- Link defines the contains relationship:



Association example



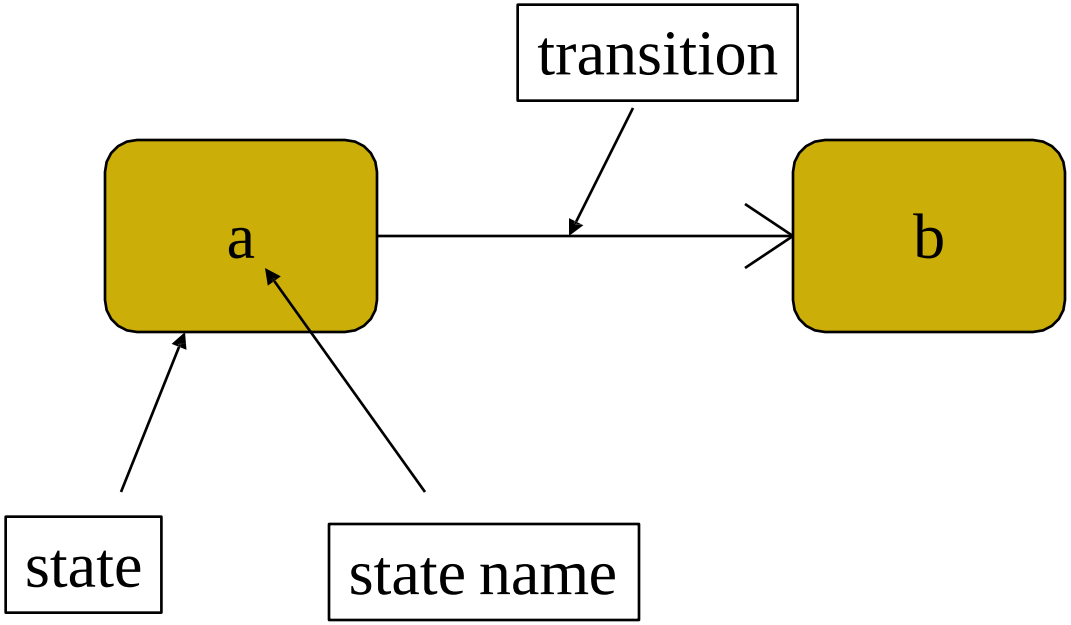
Stereotypes

- Stereotype: recurring combination of elements in an object or class.
- Example:
  - <<foo>>

Behavioral description

- Several ways to describe behavior:
  - internal view;
  - external view.

State machines



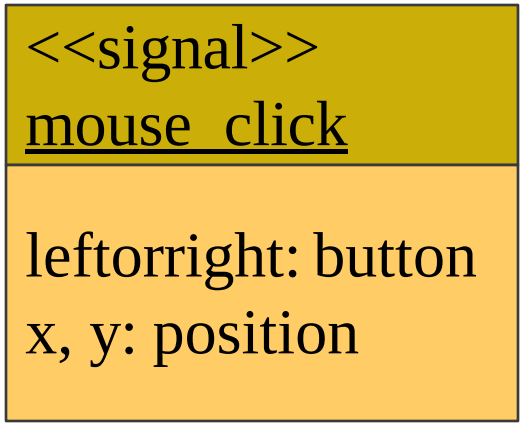
Event-driven state machines

- Behavioral descriptions are written as event-driven state machines.
  - Machine changes state when receiving an input.
- An event may come from inside or outside of the system.

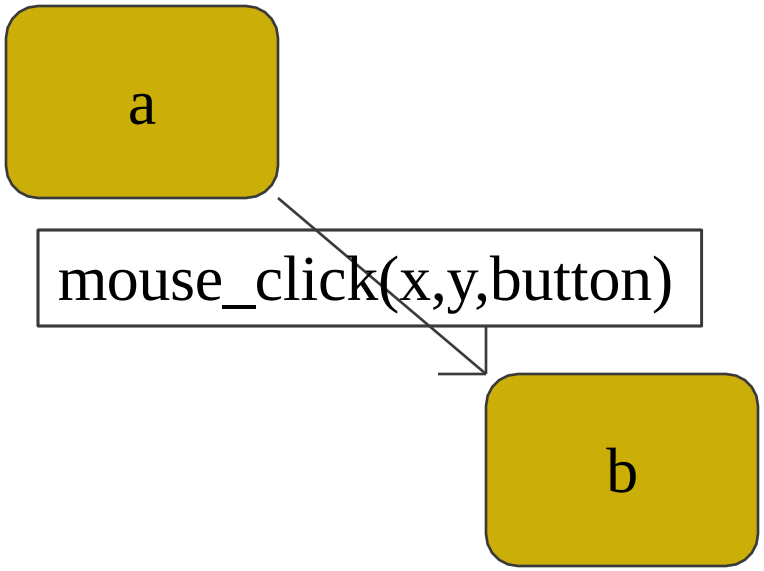
Types of events

- Signal: asynchronous event.
- Call: synchronized communication.
- Timer: activated by time.

Signal event

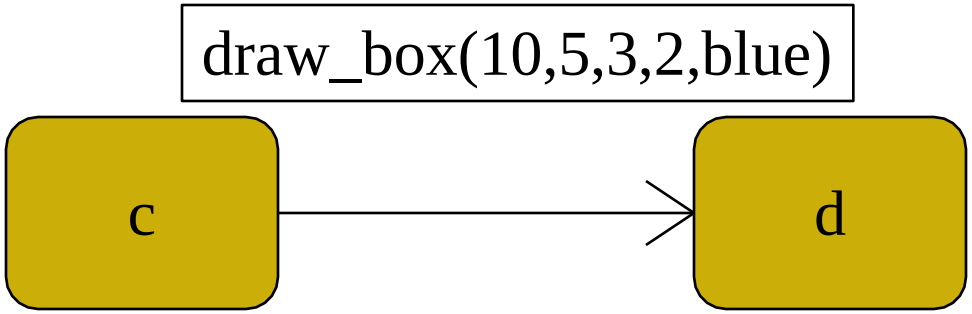


declaration

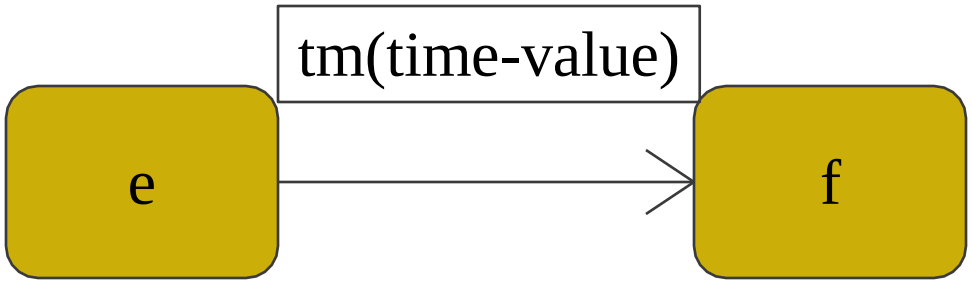


event description

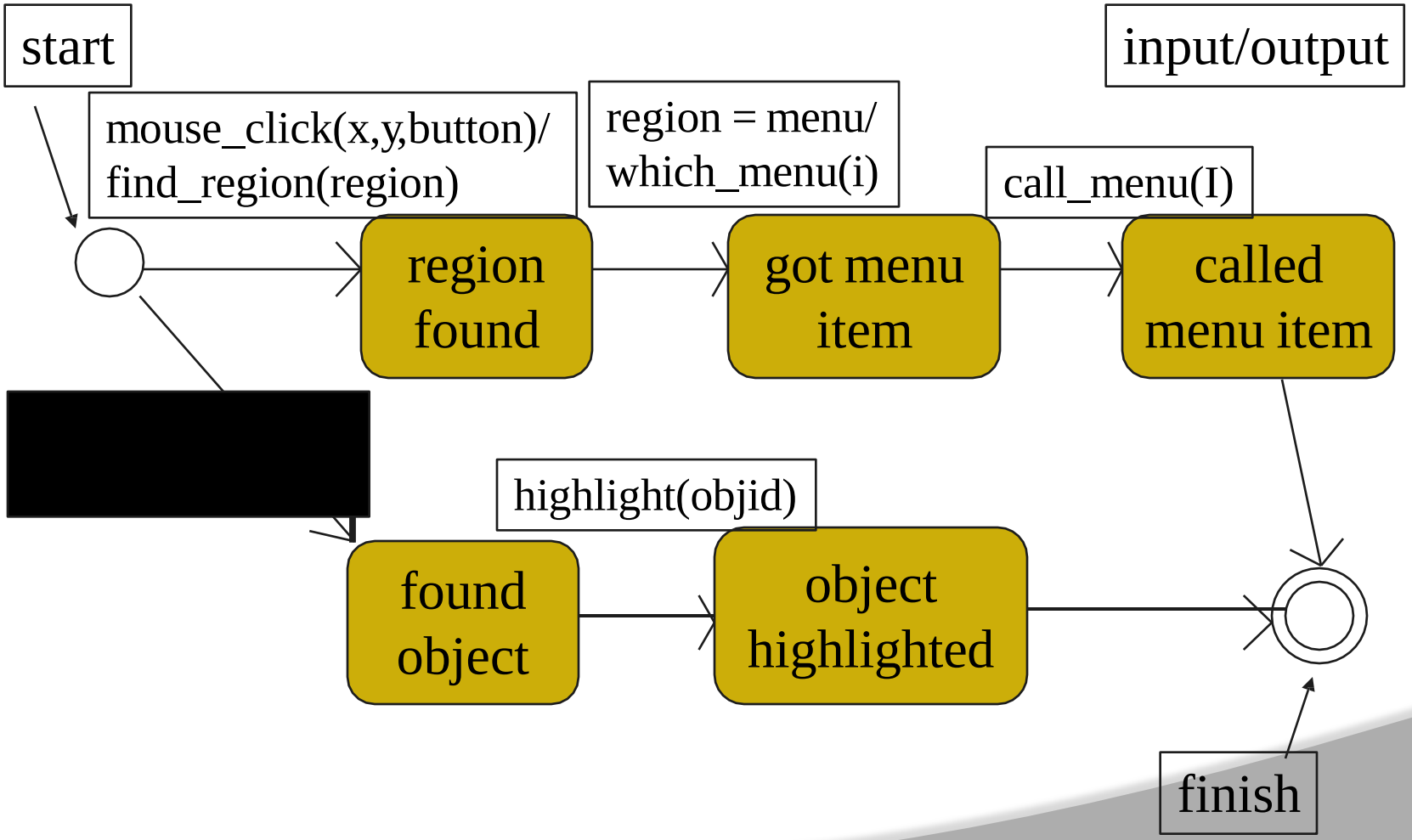
Call event



Timer event



Example state machine



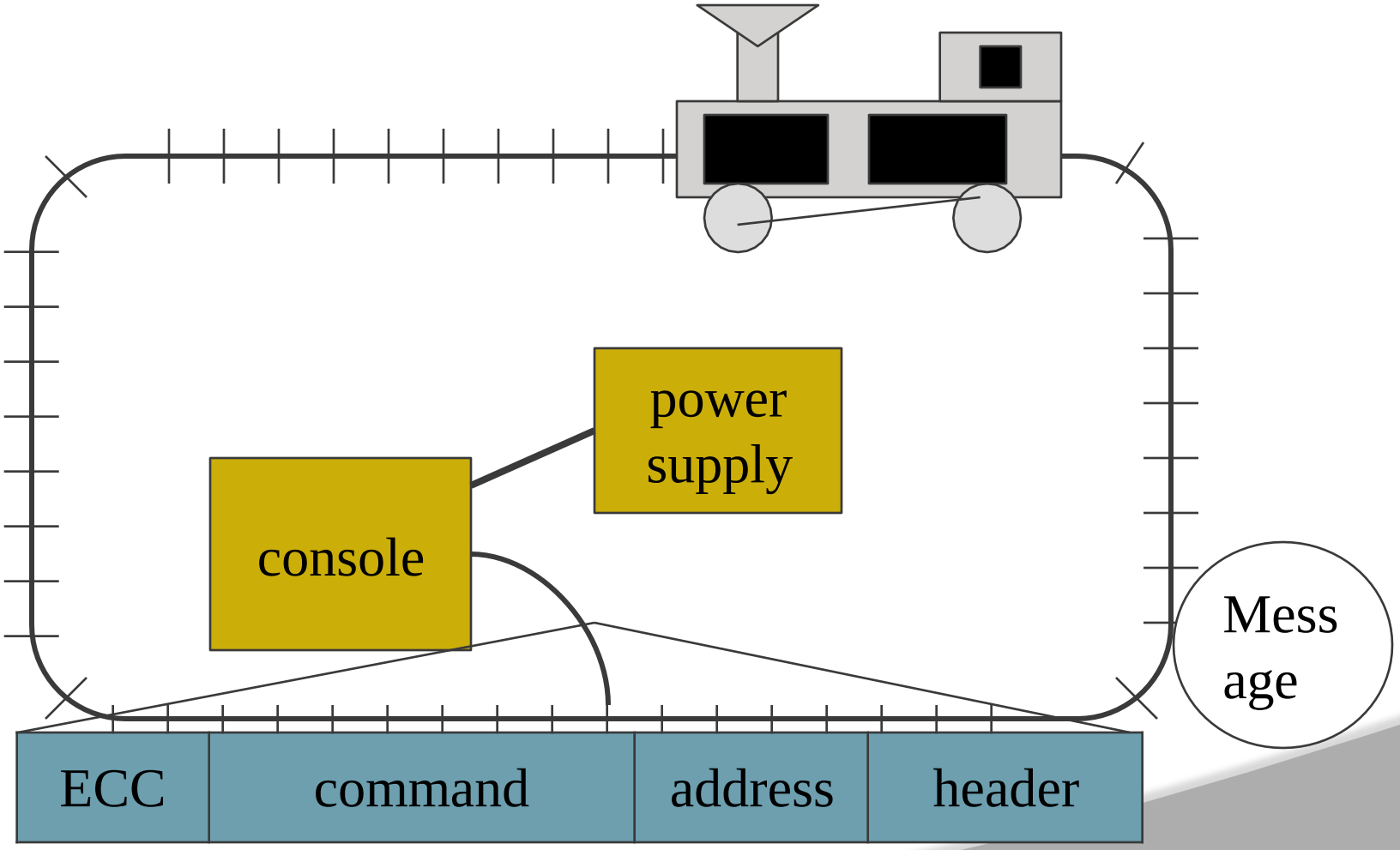
Introduction

- Example: model train controller.

Purposes of example

- Follow a design through several levels of abstraction.
- Gain experience with UML.

Model train setup



Requirements

- Console can control 8 trains on 1 track.
- Throttle has at least 63 levels.
- Inertia control adjusts responsiveness with at least 8 levels.
- Emergency stop button.
- Error detection scheme on messages.

Requirements form

name	model train controller
purpose	control speed of <= 8 model trains
inputs	throttle, inertia, emergency stop, train #
outputs	train control signals
functions	set engine speed w. inertia; emergency stop
performance	can update train speed at least 10 times/sec
manufacturing	$50

Digital Command Control

- DCC created by model railroad hobbyists, picked up by industry.
- Defines way in which model trains, controllers communicate.
  - Leaves many system design aspects open, allowing competition.
- This is a simple example of a big trend:
  - Cell phones, digital TV rely on standards.

DCC documents

- Standard S-9.1, DCC Electrical Standard.
  - Defines how bits are encoded on the rails.
- Standard S-9.2, DCC Communication Standard.
  - Defines packet format and semantics.

UNIT-II

INTRODUCTION TO EMBEDDED C AND APPLICATIONS

CLOs	Course Learning Outcome
CLO5	Understand the basic programming of c and its looping structure.
CLO6	Analyze the embedded C programming in Keil IDE, and compiling and building the hardware.
CLO7	Understand different concepts of display and keyboard interfacing using embedded C.
CLO8	Understand different concepts of serial communication using embedded C and user interfacing

What is an embedded system?

An embedded system is an application that contains at least one programmable computer (typically in the form of a microcontroller, a microprocessor or digital signal processor chip) and which is used by individuals who are, in the main, unaware that the system is computer-based.

Typical examples of embedded applications include:

Mobile phone systems (including both customer handsets and base stations).

Automotive applications (including braking systems, traction control, airbag release systems, engine-management units, steer-by-wire systems and cruise control applications).

Domestic appliances (including dishwashers, televisions, washing machines**,** microwave ovens, video recorders, security systems, garage door controllers).

Contd..

Aerospace applications (including flight control systems, engine controllers, autopilots and passenger in-flight entertainment systems).

Medical equipment (including anaesthesia monitoring systems, ECG monitors, drug delivery systems and MRI scanners).

Defence systems (including radar systems, fighter aircraft flight control systems, radio systems and missile guidance systems).

Which processor should you use?

The 8051 device is very different. It is a well-tested design, introduced in its original form by Intel in 1980 (Figure 1.1). The development costs of this device have now been fully recovered, and prices of modern 8051 devices now start at less than US $1.00. At this price, you get a performance of around 1 million instructions per second, and 256 bytes (not megabytes!) of on-chip RAM. You also get 32 port pins and a serial interface.

The 8051's profile (price, performance, available memory, serial interface) match the needs of many embedded systems very well. As a result, it is now produced in more than 400 different forms by a diverse range of companies including Philips, Infineon, Atmel and Dallas. Sales of this vast family are estimated to have the largest share (around 60%) of the microcontroller market as a whole, and to make up more than 50% of the 8-bit microcontroller market. Versions of the 8051 are currently used in a long list of embedded products, from children's toys to automotive systems.

Contd..

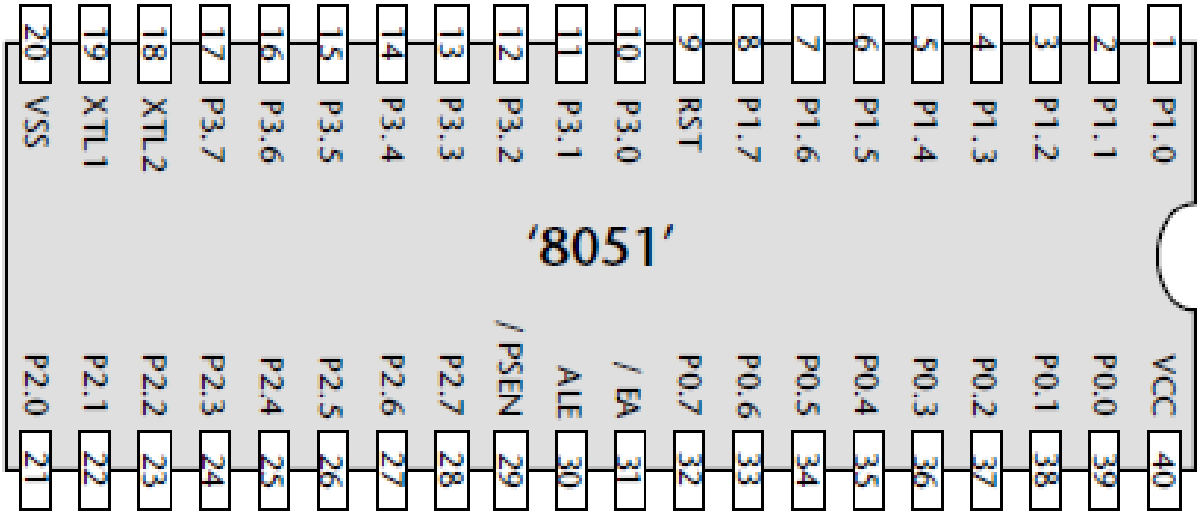
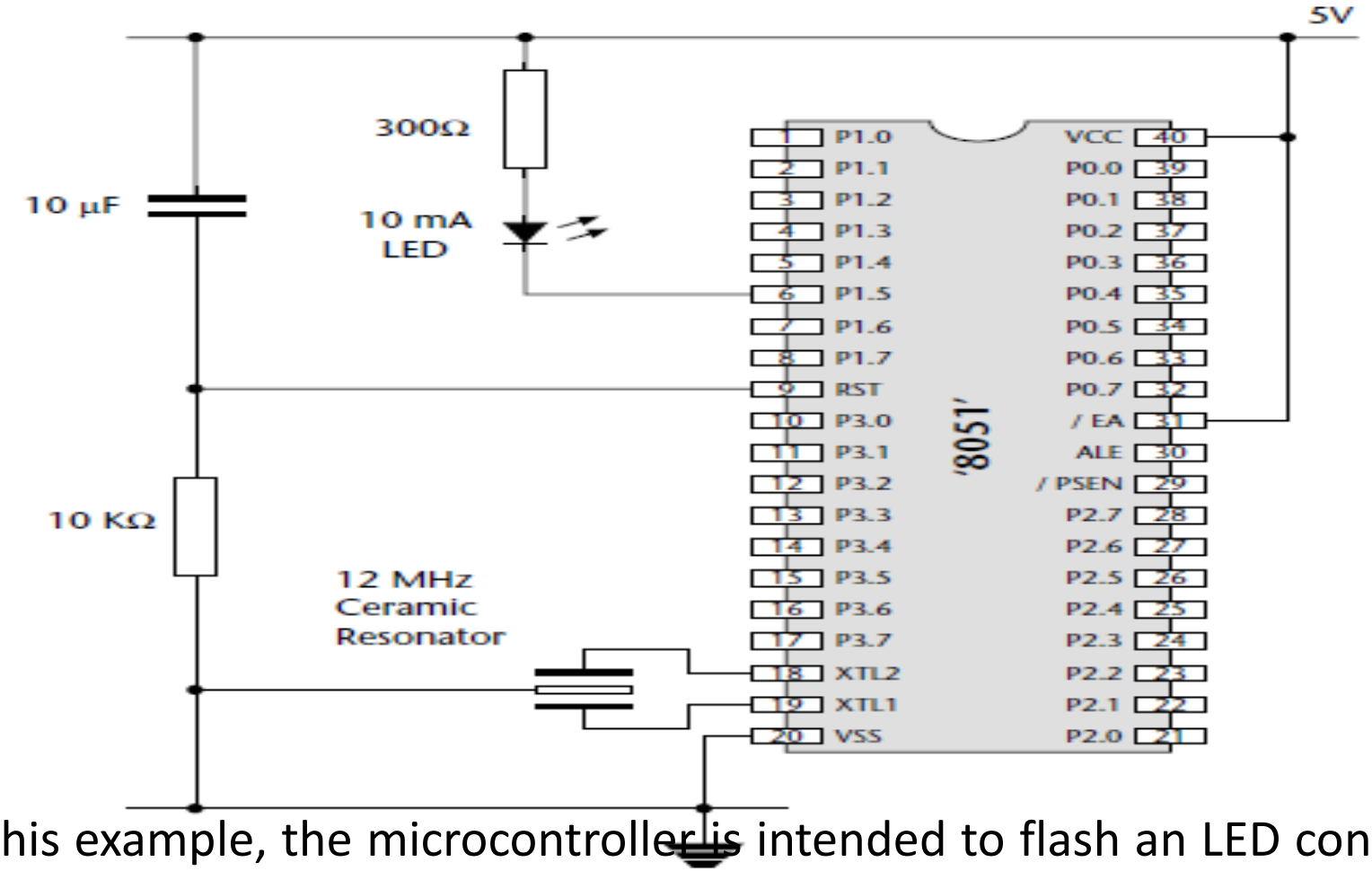


FIGURE 1.1 The external interface of a 'Standard' '8051' microcontroller (40-pin DIP package). Standard 8051s have four ports, and are pin compatible with the original 8051/8052 from Intel.

Below figure shows the circuit diagram for a complete 8051- based application.

Contd..



In this example, the microcontroller is intended to flash an LED connected to Pin 6. In addition to this LED, only a simple 'reset' circuit is required (the capacitor and resistor connected to Pin 9), plus an external oscillator (in this case, a 3-pin ceramic resonator)

Contd..

Overall, the low cost, huge range, easy availability and widespread use of the 8051 architecture makes it an excellent platform for developing embedded systems: these same factors also make it an ideal platform for learning about embedded systems.

Whether you will subsequently use 8-, 16- or 32-bit embedded processors, learning to work within the performance and memory limits of devices such as the 8051 is a crucial requirement in the cost-conscious embedded market.

Which programming language should you use?

Having decided to use an 8051 processor as the basis of your embedded system, the next key decision that needs to be made is the choice of programming language. In order to identify a suitable language for embedded systems, we might begin by making the following observations:

- Computers (such as microcontroller, microprocessor or DSP chips) only accept instructions in 'machine code' ('object code'). Machine code is, by definition, in the language of the computer, rather than that of the programmer. Interpretation of the code by the programmer is difficult and error prone.
- All software, whether in assembly, C, C++, Java or Ada must ultimately be translated into machine code in order to be executed by the computer.
- There is no point in creating 'perfect' source code, if we then make use of a poor translator program (such as an assembler or compiler) and thereby generate executable code that does not operate as we intended.

Contd..

- Embedded processors – like the 8051 – have limited processor power and very limited memory available: the language used must be efficient.
- To program embedded systems, we need low-level access to the hardware: this means, at least, being able to read from and write to particular memory locations (using 'pointers' or an equivalent mechanism).

From one point of view, only machine code is safe, since every other language involves a translator, and any code you create is only as safe as the code written by the manufacturers of the translator. On the other hand, real code needs to be maintained and re-used in new projects, possibly on different hardware: few people would argue that machine code is easy to understand, debug or to port. Inevitably, therefore, we need to make compromises; there is no perfect solution. All we can really say is that we require a language that is efficient, high-level, gives low-level access to hardware, and is well defined. In addition – of course – the language must be available for the platforms we wish to use. Against all of these points, C scores well.

Contd..

We can summarize C's features as follows:

- It is 'mid-level', with 'high-level' features (such as support for functions and modules), and 'low-level' features (such as good access to hardware via pointers).
- It is very efficient.
- It is popular and well understood.
- Even desktop developers who have used only Java or C++ can soon understand C syntax.
- Good, well-proven compilers are available for every embedded processor (8-bit to 32-bit or more).
- Experienced staff are available.
- Books, training courses, code samples and WWW sites discussing the use of the language are all widely available.
- Overall, C's strengths for embedded system development greatly outweigh its weaknesses. It may not be an ideal language for developing embedded systems, but it is unlikely that a 'perfect' language will ever be created.

Which operating system should you use?

Having opted to create our 8051-based applications using C, we can now begin to consider how this language can be used. In doing so, we will begin to probe some of the differences between software development for desktop and embedded systems.

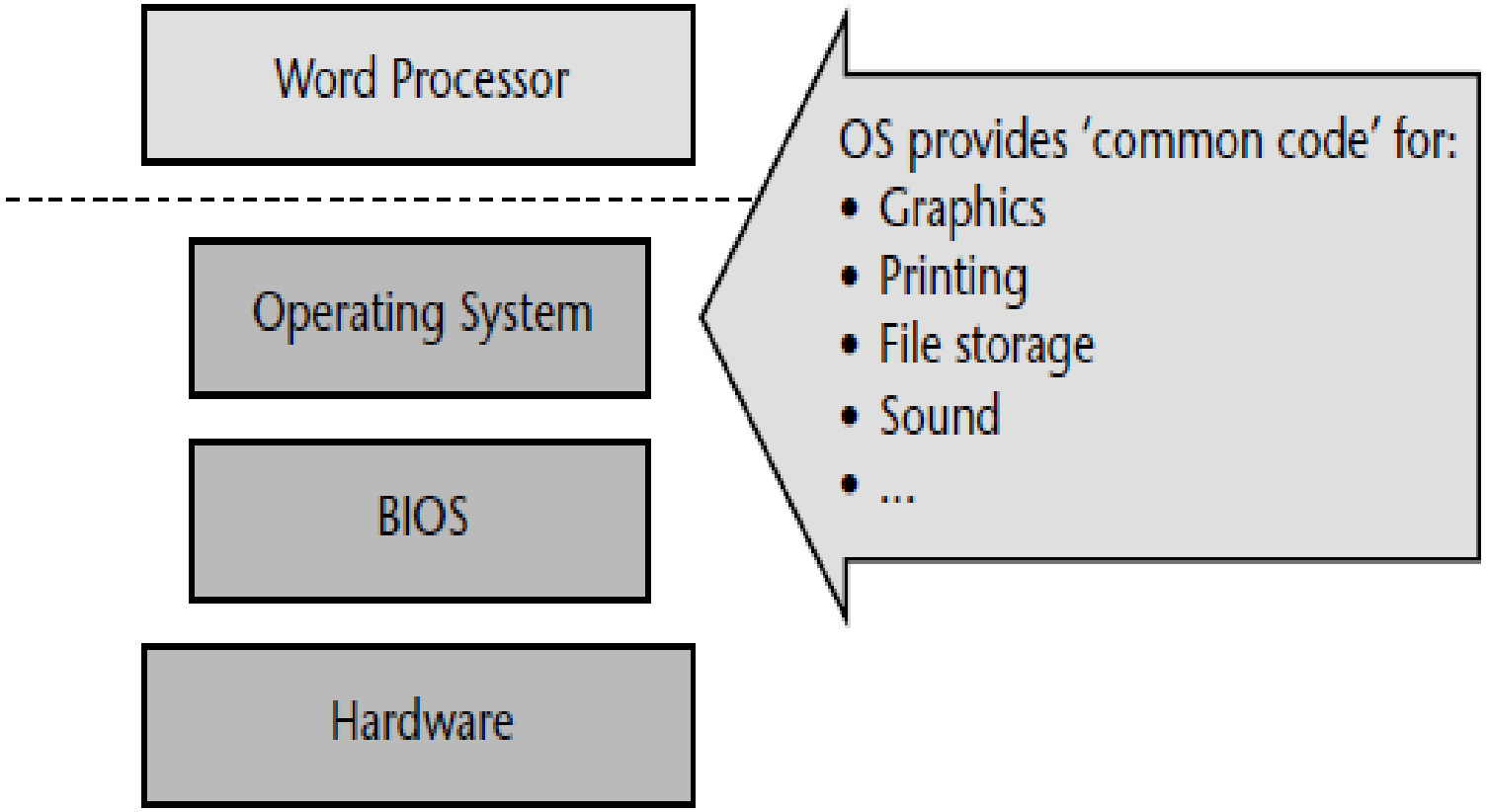
In the desktop environment, the program the user requires (such as a word processor program) is usually loaded from disk on demand, along with any required data (such as a word processor file). Figure 1.5 shows a typical operating environment for such a word processor. Here the system is well insulated from the underlying hardware. For example, when the user wishes to save his or her latest novel on disk, the word processor delegates most of the necessary work to the operating system, which in turn may delegate many of the hardware-specific commands to the BIOS (basic input/output system).

Contd..

The desktop PC does not require an operating system (or BIOS). However, for most users, the main advantage of a personal computer is its flexibility: that is, that the same piece of equipment has the potential to run many thousands of different programs.

If the PC had no operating system, each of these programs would need to be able to carry out all the low-level functions for itself. This would be very inefficient and would tend to make systems more expensive. It would also be likely to lead to errors, as many simple functions would have to be duplicated in even the smallest of programs. One way of viewing this is that a desktop PC is used to run multiple programs, and the operating system provides the 'common code' (for printing, file storage, graphics, and so forth) that is required by this set of programs

Contd..



A schematic representation of the BIOS/OS sandwich from a desk-bound computer system running some word processor software

Contd..

There are two fundamental differences between the embedded systems and desktop computer systems:

1 The vast majority of embedded systems are required to run only one program: this program will start running when the microcontroller is powered up, and will stop running when the power is removed.

2 Many of the facilities provided by the modern desktop OS – such as the ability to display high-resolution graphics, printing facilities and efficient disk access – are of little value in embedded systems, where sophisticated graphics screens, printers and disks are generally unavailable.

As a consequence, the simplest architecture in an embedded system is typically a form of 'Super Loop'

Contd..

Part of a simple Super Loop demonstration

```
void main(void)
    {
    // Prepare run function X
    X_Init();

    while(1) // 'for ever' (Super Loop)
        {
        X(); // Run function X()
        }
    }
```

It is important to appreciate that there is no operating system in use here. When power is applied to the system, the function main() will be called: having performed the initializations, the function X() will be called, repeatedly, until the system is disconnected from the power supply (or a serious error occurs).

Contd..

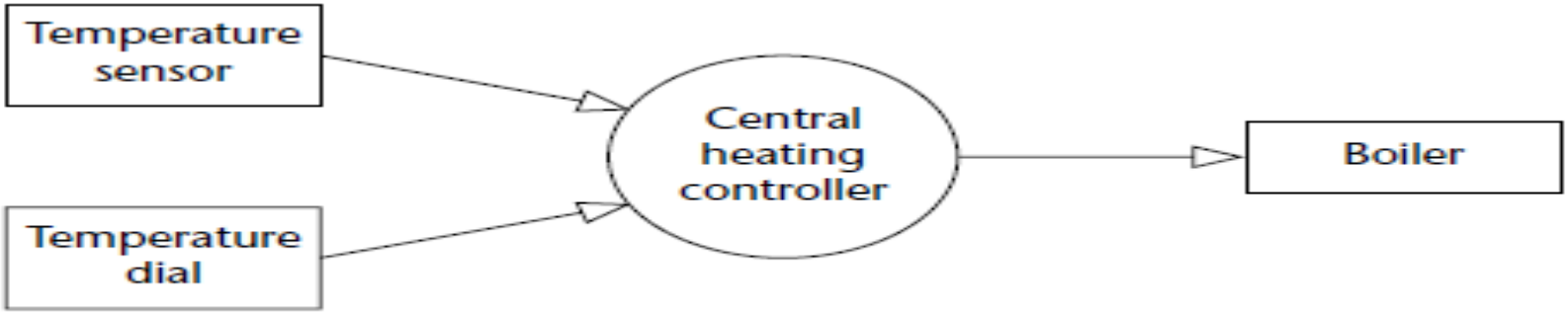
It is important to appreciate that there is no operating system in use here. When power is applied to the system, the function main() will be called: having performed the initializations, the function X() will be called, repeatedly, until the system is disconnected from the power supply (or a serious error occurs).

For example, suppose we wish to develop a microcontroller-based control system to be used as part of the central-heating system in a building. The simplest version of this system might consist of a gas-fired boiler (which we wish to control), a sensor (measuring room temperature), a temperature dial (through which the desired temperature is specified) and the controller itself

We assume that the boiler, temperature sensor and temperature dial are connected to the system via appropriate ports.

Here, precise timing is not required, and a Super Loop framework similar to that shown in Listing may be appropriate.

Contd..



An overview of a central heating controller

Part of the code for a simple central-heating system

```
/*-------------------------------------------------------*-
   Main.C
  -------------------------------------------------------
   Framework for a central heating system using a Super Loop.
   [Compiles and runs but does nothing useful]
-*-------------------------------------------------------*/
#include "Cen_Heat.h"
/*-------------------------------------------------------*/
void main(void)
   {
   // Init the system
   C_HEAT_Init();
```

Contd..

```
   while(1) // 'for ever' (Super Loop)
      {
      // Find out what temperature the user requires
      // (via the user interface)
      C_HEAT_Get_Required_Temperature();

      // Find out what the current room temperature is
      // (via temperature sensor)
      C_HEAT_Get_Actual_Temperature();

      // Adjust the gas burner, as required
      C_HEAT_Control_Boiler();
      }
   }
/*-----------------------------------------------------------*-
  ---- END OF FILE ----------------------------------------
-*-----------------------------------------------------------*/
```

It should be noted that the Super Loop architecture employed in this central heating system is not appropriate for all embedded applications.

How do you develop embedded software?

The process of compiling, linking and executing the program on a desktop PC is straightforward. In this environment, the executable code we create will, in almost all cases, be intended to run on a desktop computer similar to the one on which the code development is carried out. In the embedded environment this is rarely the case. For example, the 8051 devices we will use throughout this book do not have sufficient memory resources to allow them to be used for compiling programs, and they will not support a keyboard or graphics display. As a result, the code will be 'cross-compiled' on a desktop PC, generating machine code that is compatible with the 8051 family

Contd..

Having created the required executable code, we need to test it and refine it. To do this, we need to do the following:

1 Build the hardware for the embedded system.

2 Transfer the executable code to the embedded hardware and test the system.

For programmers without experience of electronics, the process of building embedded hardware is a daunting one. A typical approach used to prototype small embedded applications is the ‘breadboard’. This allows the microcontroller and associated components to be connected together, without soldering, in order to test and refine the hardware and software design.

Contd..

1 Create the executable code for the embedded system on a desktop PC using an appropriate cross-compiler and related tools.

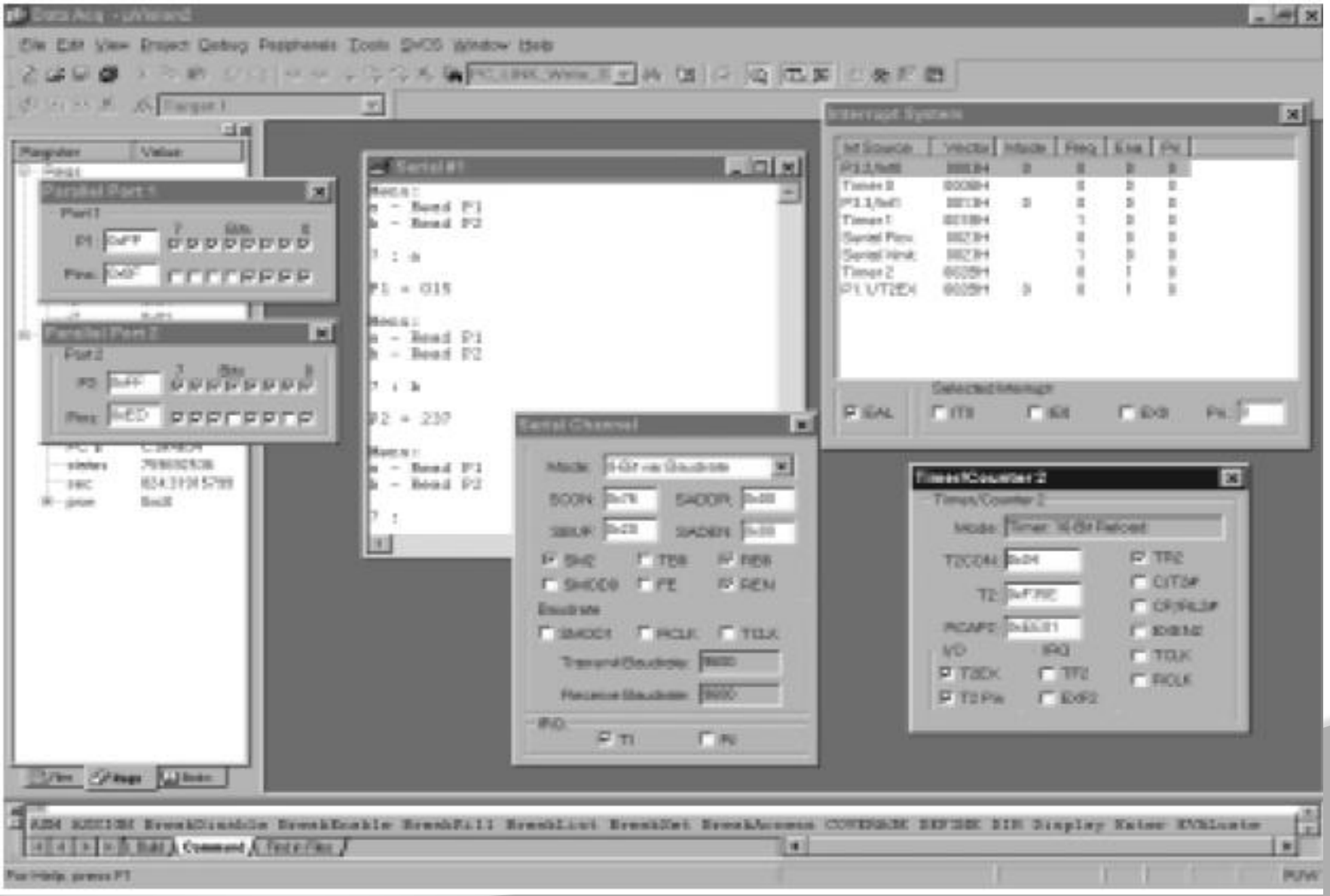
2 Use a software simulator (running on the desktop PC) to test the code.

3 Repeat Step 1 and Step 2, as necessary, until the software operates as required.

We will use such a simulator, produced by Keil Software. This provides you with a very flexible 'hardware' platform, on which you can gain experience of embedded software without simultaneously having to learn how to solder.

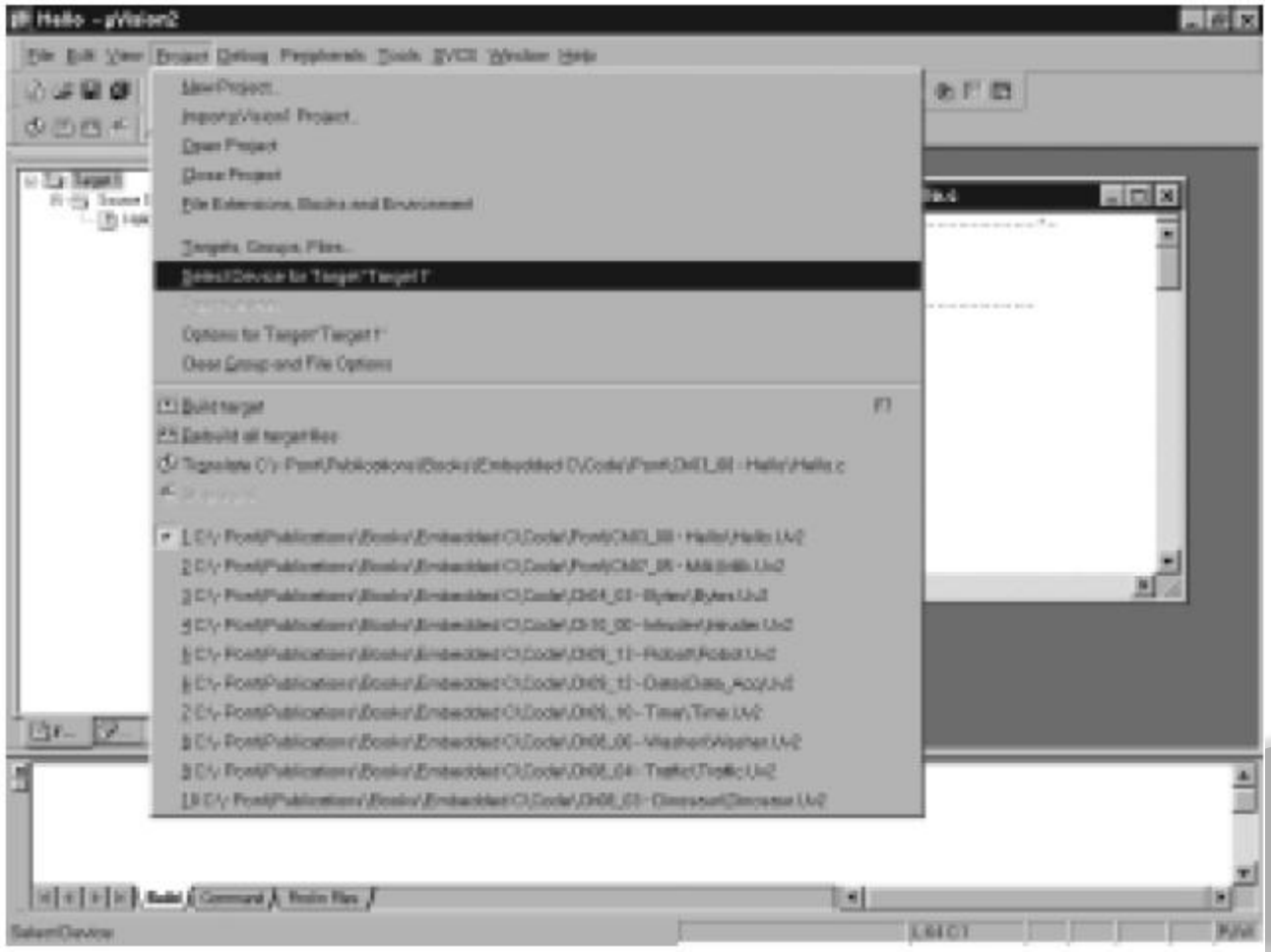
An example of the simulator in use is given in Figure.

Contd..



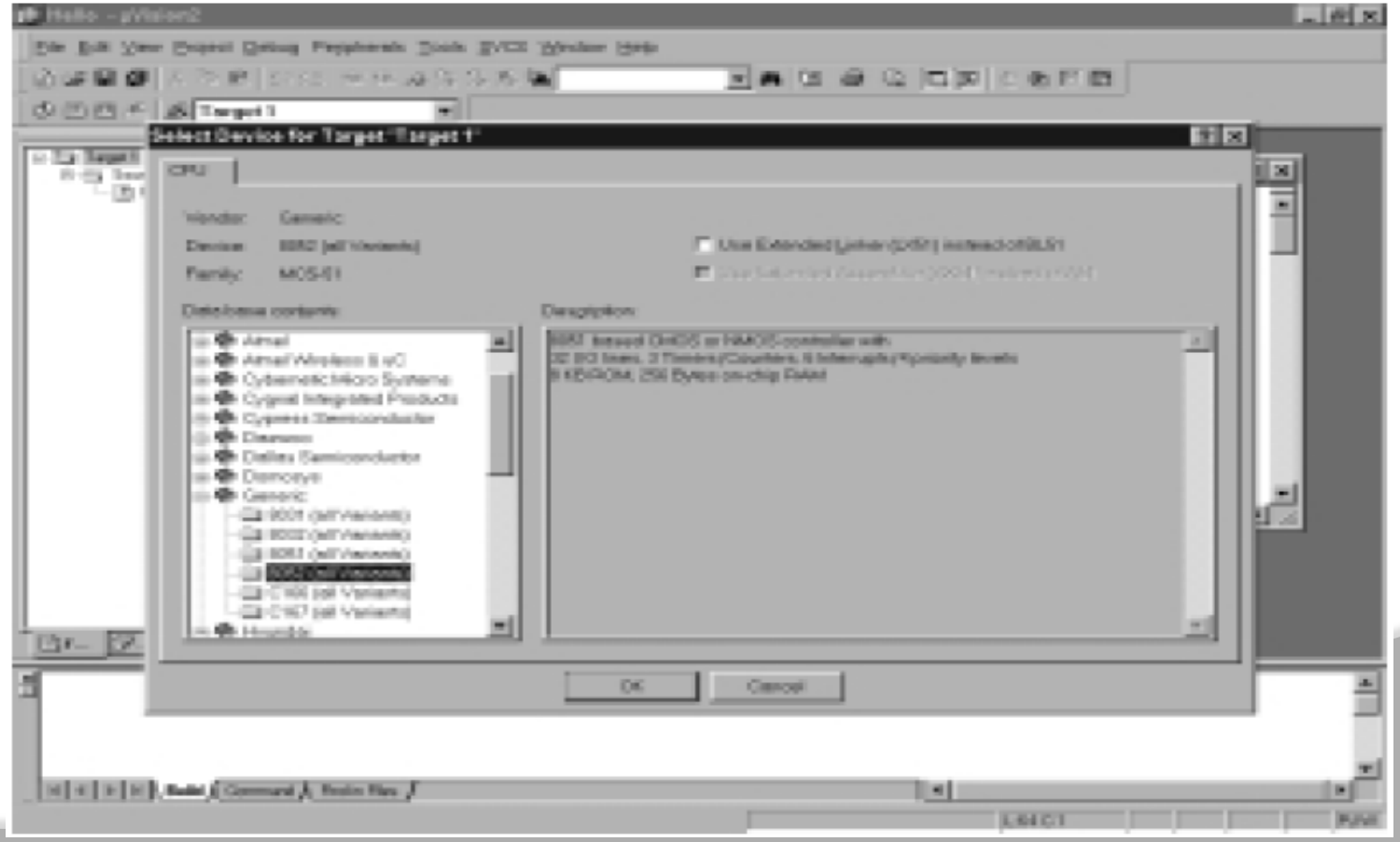
Installing the Keil software and loading the project

When you have copied the files onto your hard disk, please run the Keil µVision application, and use the ‘Open Project’ option (from the ‘Project’ menu) to load the ‘Hello’ example.



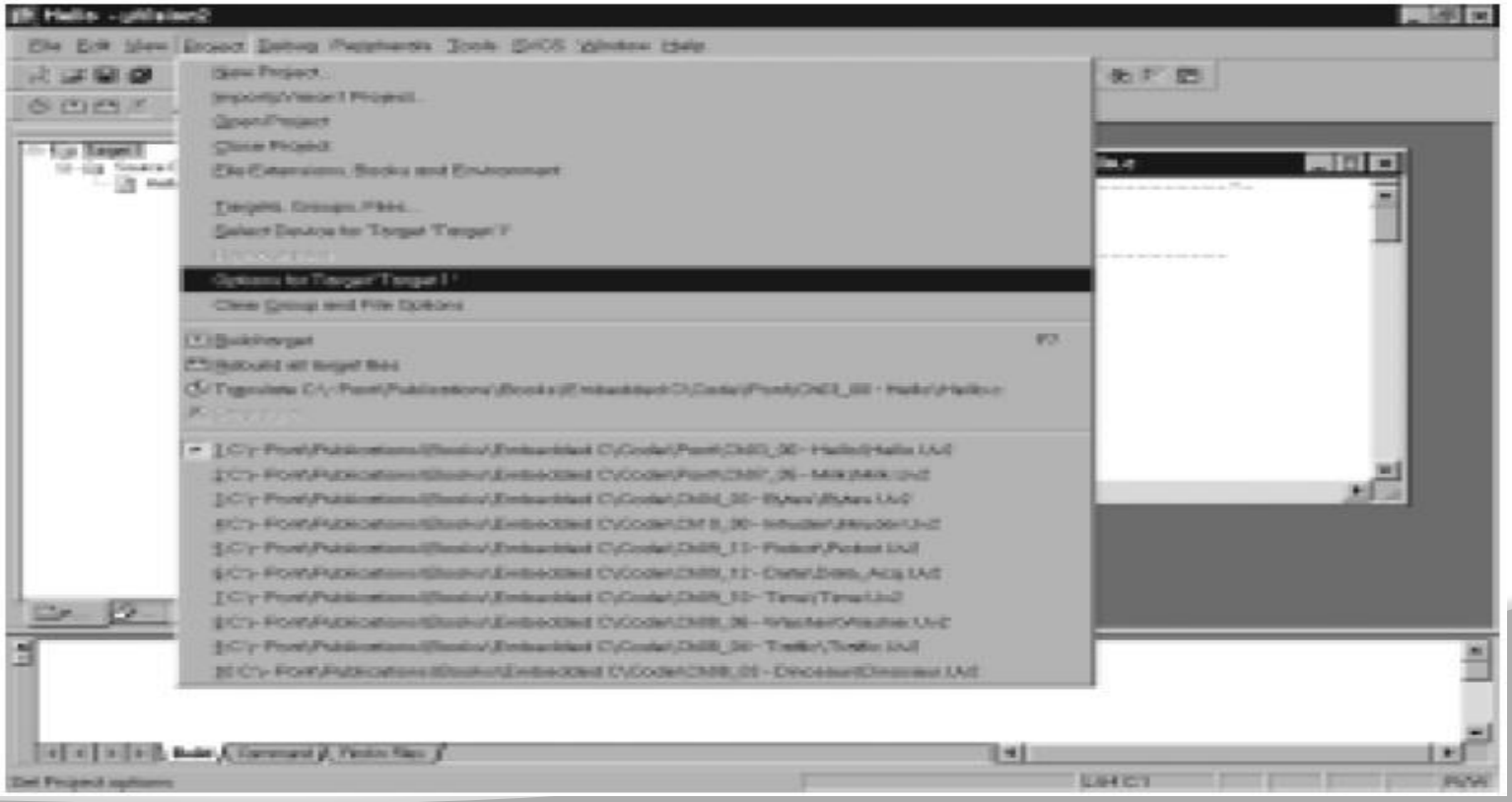
Configuring the simulator

Having loaded the ‘Hello’ project in the Keil µVision environment, we will begin by exploring the project settings. First, using the Project menu, we will look at the 8051 device which we are intending to use for this application.



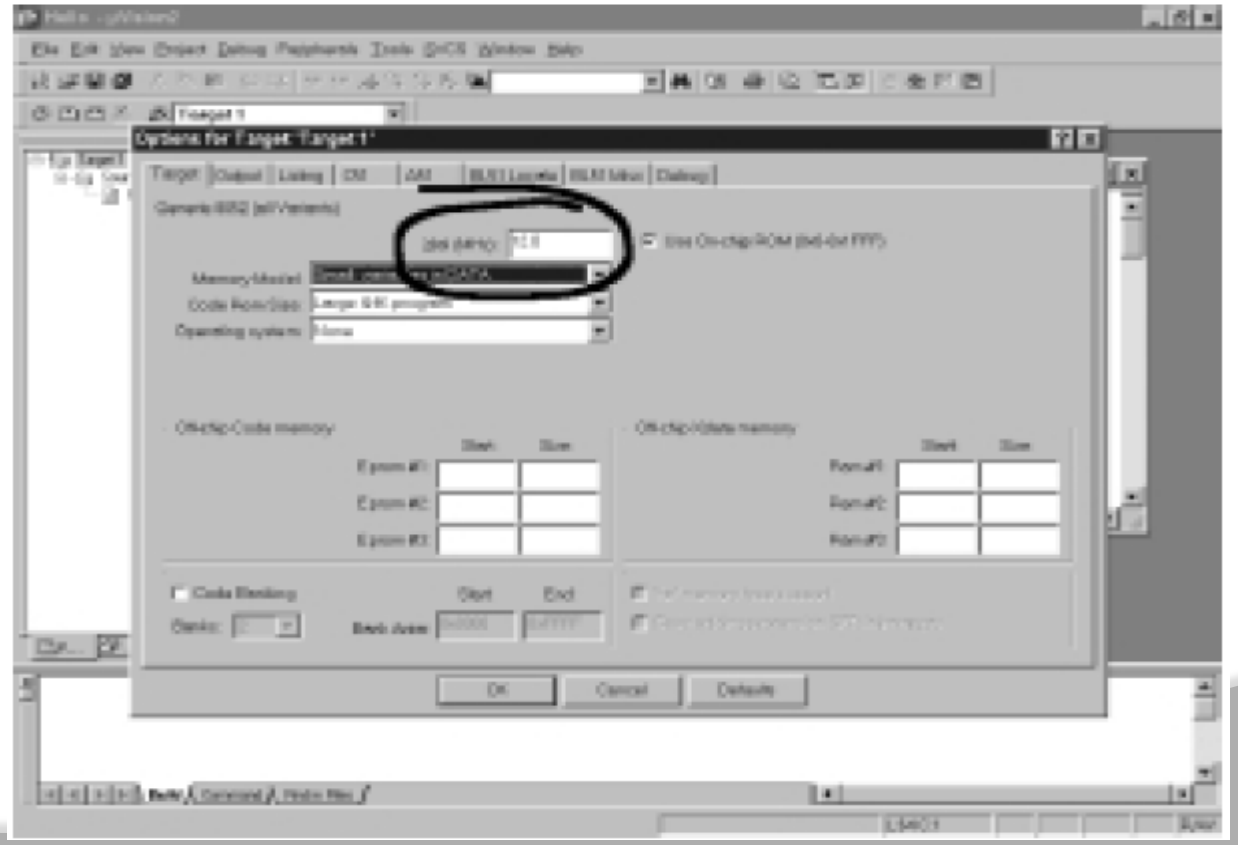
Contd..

In this case, we will use a generic ‘8052’ driver. The next thing we need to check is the oscillator frequency. The choice of oscillator frequency has a large impact on 8051-based applications. In most examples in this book, we will assume that the oscillator frequency is 12 MHz. Figure shows how to inspect and if required alter this frequency.



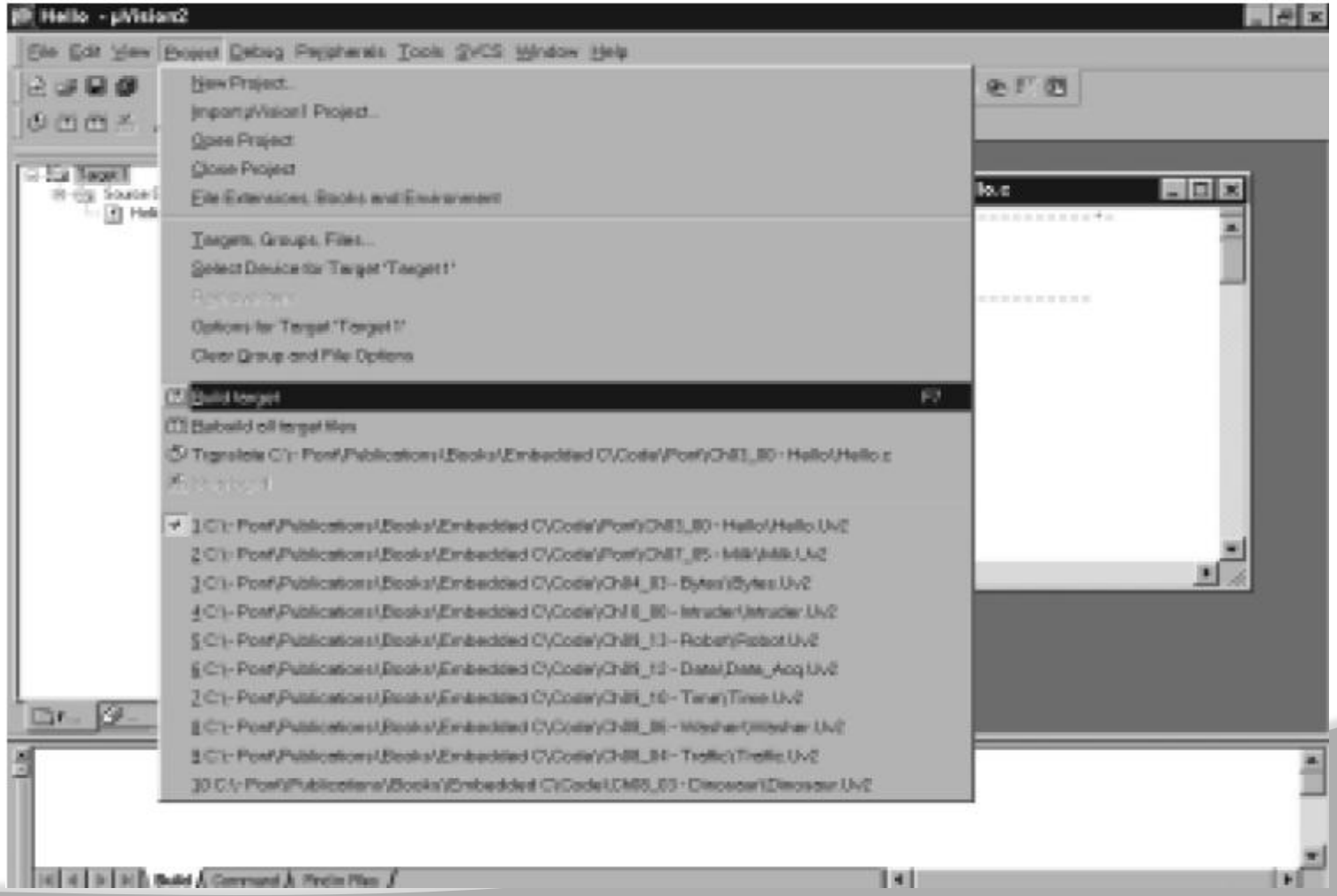
Contd..

Note: one of the key reasons for setting the oscillator frequency in the simulator is that any attempting at ‘profiling’ the application (for example, measuring function durations) will only be successful if the oscillator frequency in the simulator matches the frequency that will be used in the real system hardware.



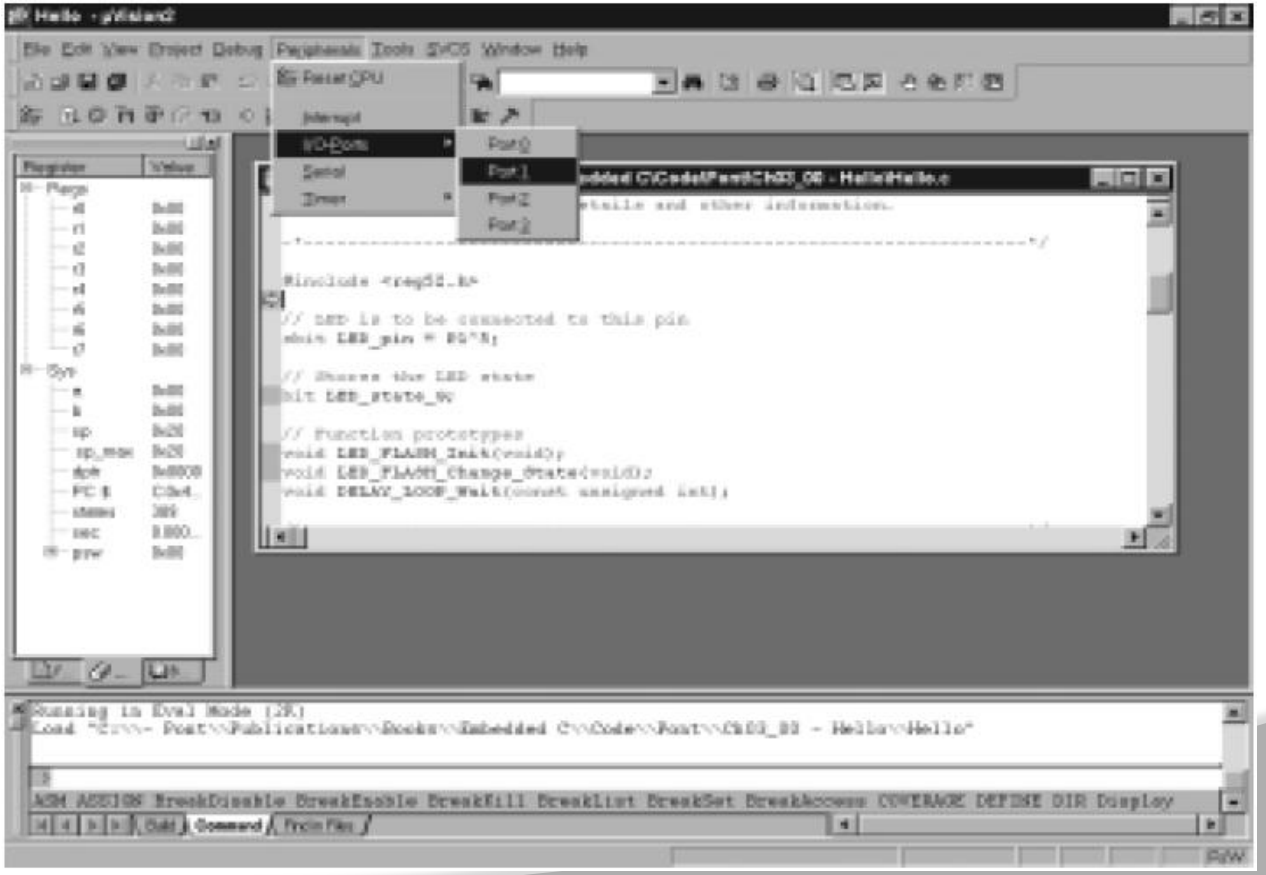
Building the target

We next need to build the ‘target’, as illustrated in Figure. Building the target (compiling and linking your source files) in the Keil environment.



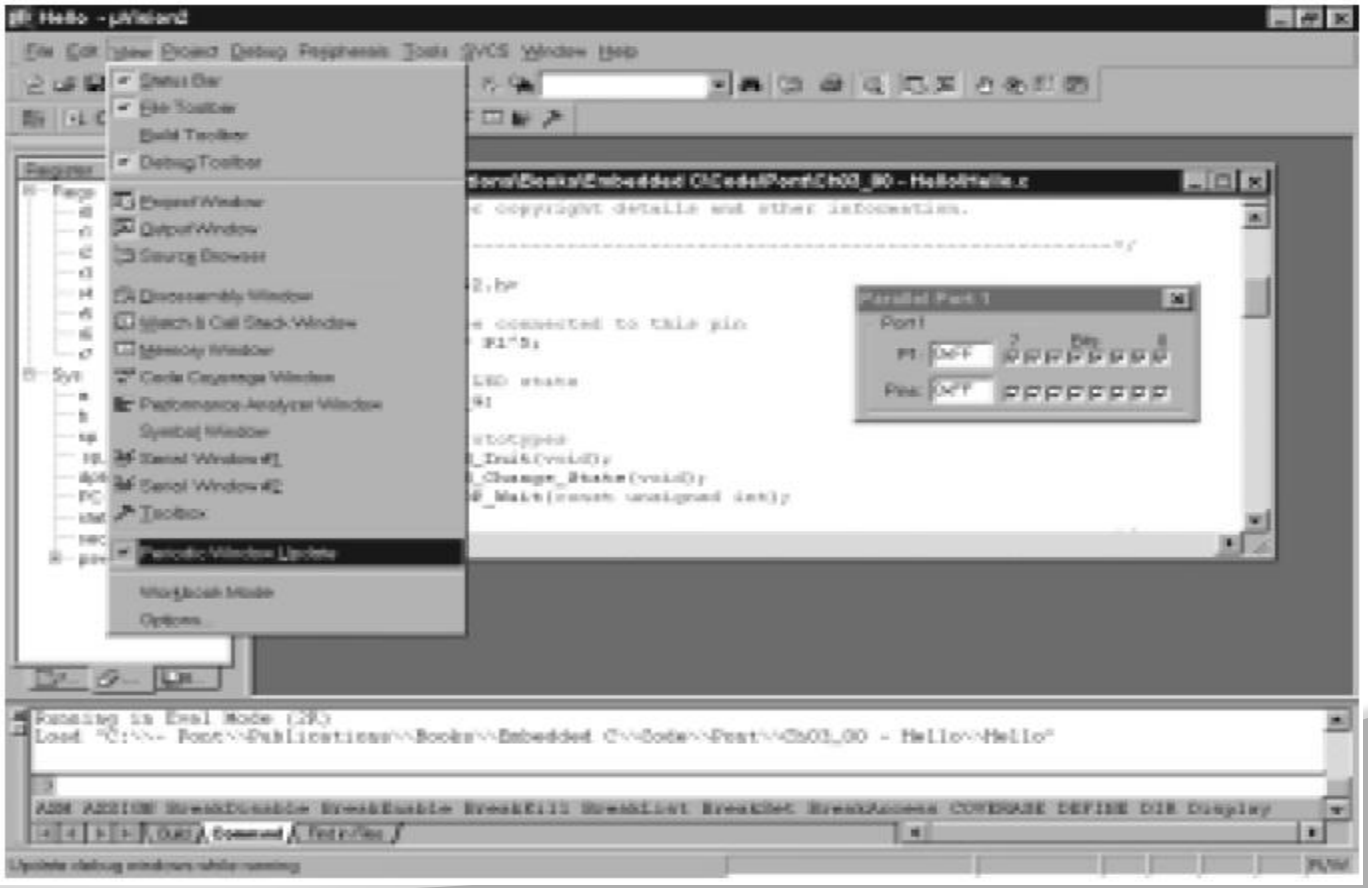
Running the simulation

Having successfully built the target, we are now ready to start the debug session and run the simulator. First, start a debug session. The ‘flashing LED’ we will view will be connected to Port 1. We therefore want to observe the activity on this port.



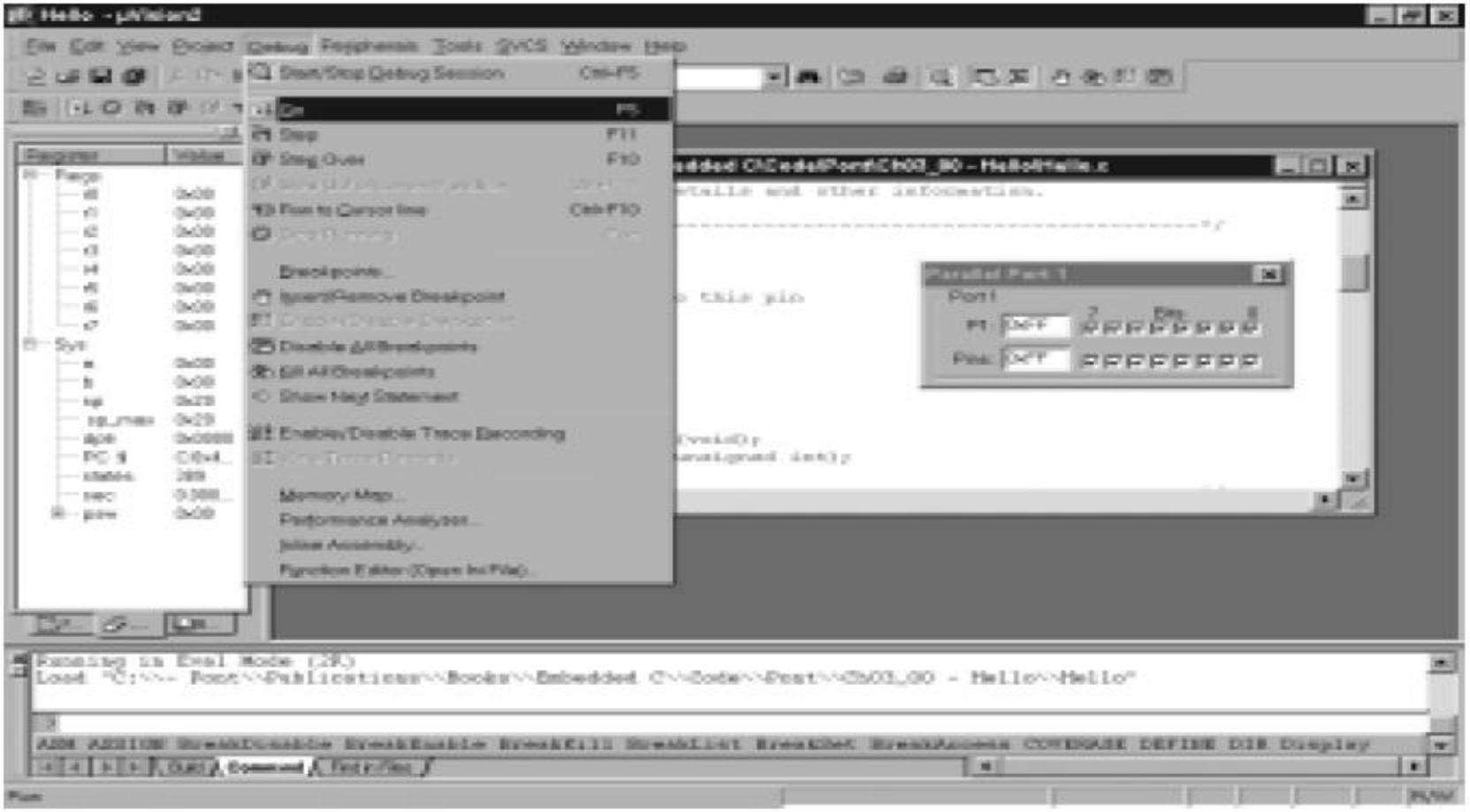
Contd..

By default, to speed up the simulation, updates to the various components are carried out only on demand. For our purposes, we want to ensure that the simulator regularly updates the screen: we do this by ticking the ‘Periodic Window. Update’ option in the ‘View’ menu.



Contd..

Finally, we are ready to start running our ‘Hello, Embedded World’ program in the simulator. As the program runs, observe that Pin 1.5 flashes ‘on’ and ‘off’, as required.



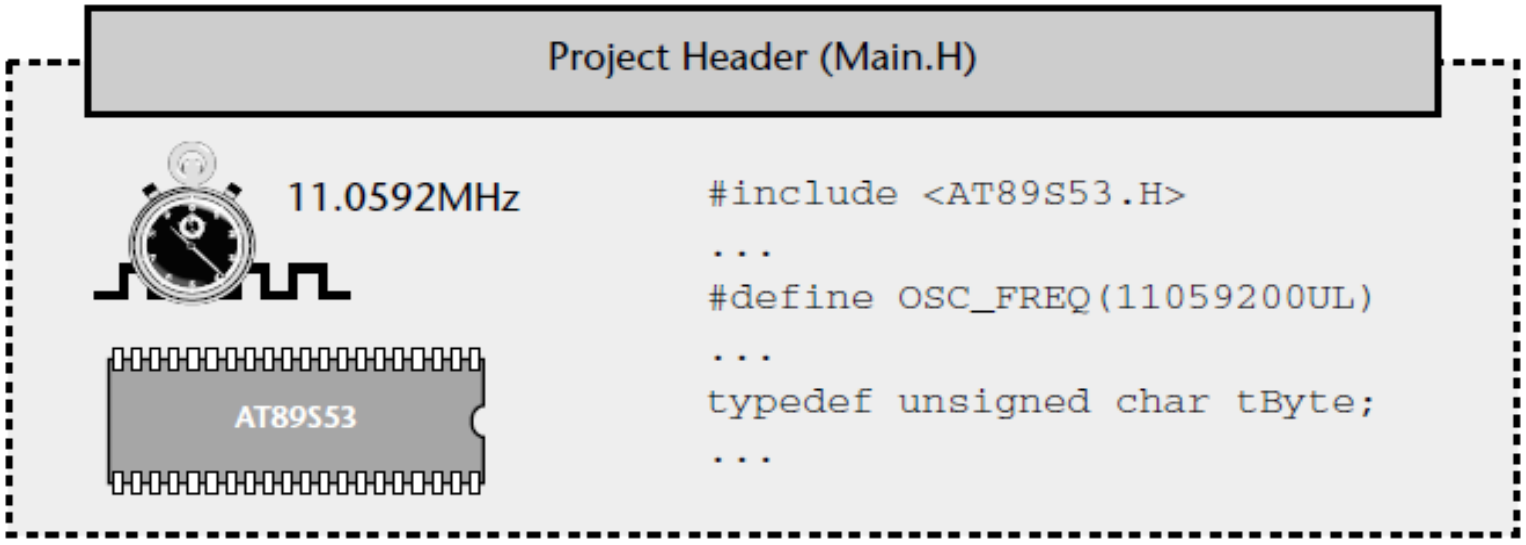
Building the hardware

You should base your design around an 8051 device with flash memory: for example, the Atmel AT89C52 is widely available, at low cost.

The required hardware schematic is taken. This may be assembled on a breadboard of the type. You will also require a suitable programmer with which to program your chosen microcontroller. Various companies produce suitable devices. Alternatively, if you have some experience in electronic construction, you will find an Application Note from Atmel which describes how to construct a suitable programmer.

The Project Header (Main.H)

The ‘Project Header’ is simply a header file, included in all projects, that groups the key information about the 8051 device you have used, along with other key parameters – such as the oscillator frequency – in one file. As such, it is a practical implementation of a standard software design guideline: ‘Do not duplicate information in numerous files; place the information in one place, and refer to it where necessary. We use a Project Header file. This is always called Main.H.



Contd..

An example of a typical project header file (Main.H)

```
/*-------------------------------------------------------------*-

   Main.H (v1.00)

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

   'Project Header' for project HELLO2 (see Chap 5)

-*-------------------------------------------------------------*/

#ifndef _MAIN_H
#define _MAIN_H

//-------------------------------------------------------------
// WILL NEED TO EDIT THIS SECTION FOR EVERY PROJECT
//-------------------------------------------------------------

// Must include the appropriate microcontroller header file here
#include <reg52.h>

// Oscillator / resonator frequency (in Hz) e.g. (11059200UL)
#define OSC_FREQ (12000000UL)

// Number of oscillations per instruction (12, etc)
//  12 - Original 8051 / 8052 and numerous modern versions
//   6 - Various Infineon and Philips devices, etc.
//   4 - Dallas 320, 520 etc.
//   1 - Dallas 420, etc.

#define OSC_PER_INST (12)
```

Contd..

```
//-----------------------------------------------------------
// SHOULD NOT NEED TO EDIT THE SECTIONS BELOW
//-----------------------------------------------------------

// Typedefs (see Chap 5)
typedef unsigned char tByte;
typedef unsigned int tWord;
typedef unsigned long tLong;

// Interrupts (see Chap 7)
#define INTERRUPT_Timer_0_Overflow 1
#define INTERRUPT_Timer_1_Overflow 3
#define INTERRUPT_Timer_2_Overflow 5

#endif

/*-----------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------
-*-----------------------------------------------------------*/
```

We consider the various components of this file in the sub-sections below

The device header

The first entry in the project header is the link to the appropriate 'device header' file. These files will, in most cases, have been produced by your compiler manufacturer, and will include the addresses of the special function registers (SFRs) used for port access, plus similar details for other on-chip components such as analog-to-digital converters.

For example, Listing shows part of the device header for an Extended 8051, the Infineon C515C. This device has eight ports, a watchdog unit, analog-to-digital converter and other components, all made accessible through the device header.

Part of a device header file (Infineon C515C). Copyright Keil Elektronik GmbH

```
/*-------------------------------------------------------------
  REG515C.H

  Header file for the Infineon C515C

  Copyright (c) 1995-1999 Keil Elektronik GmbH All rights
  reserved.
-------------------------------------------------------------*/
```

Contd..

```
. . .

/* A/D Converter */
sfr ADCON0 = 0xD8;

. . .

/* Interrupt System */
sfr  IEN0   = 0xA8;

. . .

/* Ports */
sfr  P0     = 0x80;
sfr  P1     = 0x90;
sfr  P2     = 0xA0;
sfr  P3     = 0xB0;
sfr  P4     = 0xE8;
sfr  P5     = 0xF8;
sfr  P6     = 0xDB;
sfr  P7     = 0xFA;

. . .
```

Contd..

```
/*  Serial Channel    */
sfr   SCON    = 0x98;
...

/*  Timer0 / Timer1 */
sfr   TCON    = 0x88;
...

/*  CAP/COM Unit / Timer2 */
sfr   CCEN    = 0xC1;
...

/*  Watchdog */
sfr   WDTREL = 0x86;

/*  Power Save Modes */
sfr   PCON    = 0x87;
sfr   PCON1   = 0x88;
```

Oscillator frequency and oscillations per instruction

If you create an application using a particular 8051 device operating at a particular oscillator frequency, with a particular number of oscillations per instruction, this information will be required when compiling many of the different source files in your project. For example – in many cases – we can create code for generating delays (and similar purposes) if we store information about the oscillator frequency and number of oscillations-per-instruction in an appropriate form. This is done in the Main.H file as follows:

```
// Oscillator / resonator frequency (in Hz) e.g. (11059200UL)
#define OSC_FREQ (12000000UL)

// Number of oscillations per instruction (12, etc)
// 12 - Original 8051 / 8052 and numerous modern versions
//  6 - Various Infineon and Philips devices, etc.
//  4 - Dallas 320, 520 etc.
//  1 - Dallas 420, etc.

#define OSC_PER_INST (12)
```

Common data types

The next part of the Project Header file in Listing 5.3 includes three typedef statements:

typedef unsigned char tByte;

typedef unsigned int tWord;

typedef unsigned long tLong;

In C, the typedef keyword allows us to provide aliases for data types: we can then use these aliases in place of the original types. Thus, in the projects you will see code like this:

tWord Temperature;

Rather than:

unsigned int Temperature;

The main reason for using these typedef statements is to simplify – and promote – the use of unsigned data types. This is a good idea for two main reasons:

Contd..

- The 8051 does not support signed arithmetic and extra code is required to manipulate signed data: this reduces your program speed and increases the program size. Wherever possible, it makes sense to use unsigned data, and these typedef statements make this easier.
- Use of bitwise operators generally makes sense only with unsigned data types: use of ‘typedef’ variables reduces the likelihood that programmers will inadvertently apply these operators to signed data.

Finally, as in desktop programming, use of the typedef keyword in this way can make it easier to adapt your code for use on a different processor (for example, when you move your 8051 code to a 32-bit environment). In many circumstances, you will simply be able to change the typedef statements in Main.H, rather than editing every source file in your project.

Interrupts

As we noted in Chapter 2, interrupts are a key component of most embedded systems.
The following lines in the Project Header are intended to make it easier for you to use (timer-based) interrupts in your projects:

```
#define INTERRUPT_Timer_0_Overflow 1
#define INTERRUPT_Timer_1_Overflow 3
#define INTERRUPT_Timer_2_Overflow 5
```

Use of Project Header can help to make your code more readable, not least because anyone using your projects knows where to find key information, such as the model of microcontroller and the oscillator frequency required to execute the software. The use of a Project Header can help to make your code more easily portable, by placing some of the key microcontroller-dependent data in one place: if you change the processor or the oscillator used then – in many cases – you will need to make changes only to the Project Header.

The Port Header (Port.H)

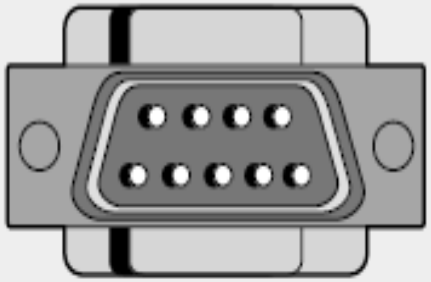
In a typical embedded project, you may have a user interface created using an LCD, a keypad, and one or more single LEDs. There may be a serial (RS-485) link to another microcontroller board. There may be one or more high-power devices (say 3-phase industrial motors) to be controlled. Each of these (software) components in your application will require exclusive access to one or more port pins.

How do you ensure that changes to port access in one component does not impact on another? How do you ensure that it is easy to adapt the application to an environment where different port pins must be used? These issues are addressed through the use of a simple Port Header file. Using a Port Header, you pull together the different port access features for the whole project into a single (header) file. Use of this technique can ease project development, maintenance and porting.

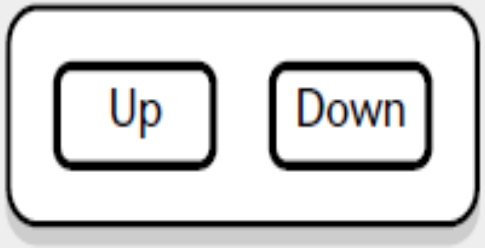
Contd..

Port Header (Port.H)

```
// Pins 3.0 and 3.1 used
// for RS-232 interface
```



```
// Switches
sbit Sw_up = P1^2;
sbit Sw_down = P1^3;
```



The Port Header file is simple to understand and easy to apply. Consider, for example, that we have three C files in a project (A, B, C), each of which require access to one or more port pins, or to a complete port.

Contd..

File A may include the following:

```
// File A
sbit Pin_A = P3^2;
. . .
```

File B may include the following:

```
// File B
#define Port_B P0
. . .
```

File C may include the following:

```
// File C
sbit Pin_C = P2^7;
. . .
```

In this version of the code, all of the port access requirements are spread over multiple files.

Contd..

Instead of this, there are many advantages obtained by integrating all port access in a single Port.H header file:

```
// ----- Port.H -----
// Port access for File B
#define Port_B P0
// Port access for File A
sbit Pin_A = P3^2;
// Port access for File C
sbit Pin_C = P2^7;
…
```

Each of the remaining project files will then '#include' the file 'Port.H'. Listing shows a complete example of a Port.H file from a real application.

Contd..

Listing : An example of a real Port Header file (Port.H) from a project using an interface consisting of a keypad and liquid crystal display

```
/*-------------------------------------------------------*-
Port.H (v1.00)
-*-------------------------------------------------------*/
#ifndef _PORT_H
#define _PORT_H
#include 'Main.H'
// ------ Menu_A.C ----------------------------------------
// Uses whole of Port 1 and Port 2 for data acquisition
#define Data_Port1 P1
#define Data_Port2 P2
// ------ PC_IO.C -----------------------------------------
// Pins 3.0 and 3.1 used for RS-232 interface
#endif
/*-------------------------------------------------------*-
---- END OF FILE ----------------------------------------*/
```

Contd..

Despite its simplicity, use of a Port Header file can improve reliability and safety, because it avoids potential conflicts between port pins, particularly during the maintenance phase of the project when developers (who may not have been involved in the original design) are required to make code changes.

A Port Header is itself portable: it can be used with any microcontroller, and is not linked to the 8051 family. Use of a Port Header also improves portability, by making accessible, in one location, all of the port access requirements of the application.

Restructuring the Hello, Embedded World example

Part of the 'Hello, Embedded World' example (restructured version)

```
/*------------------------------------------------------------*-
Main.H (v1.00)
------------------------------------------------------------
'Project Header'
-*------------------------------------------------------------*/
#ifndef _MAIN_H
#define _MAIN_H
//------------------------------------------------------------
// WILL NEED TO EDIT THIS SECTION FOR EVERY PROJECT
//------------------------------------------------------------
// Must include the appropriate microcontroller header file here
#include <reg52.h>
// Oscillator / resonator frequency (in Hz) e.g. (11059200UL)
#define OSC_FREQ (12000000UL)
// Number of oscillations per instruction (12, etc)
// 12 – Original 8051 / 8052 and numerous modern versions
```

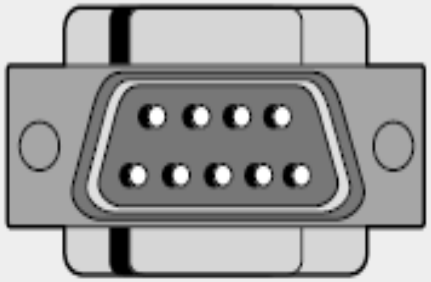
Contd..

```
// 6 – Various Infineon and Philips devices, etc.
// 4 – Dallas 320, 520 etc.
// 1 – Dallas 420, etc.
#define OSC_PER_INST (12)
//-----------------------------------------------------------
// SHOULD NOT NEED TO EDIT THE SECTIONS BELOW
//-----------------------------------------------------------
// Typedefs
typedef unsigned char tByte;
typedef unsigned int tWord;
typedef unsigned long tLong;
// Interrupts (see Chap 7)
#define INTERRUPT_Timer_0_Overflow 1
#define INTERRUPT_Timer_1_Overflow 3
#define INTERRUPT_Timer_2_Overflow 5
#endif
/*-----------------------------------------------------------*-
---- END OF FILE ----------------------------------------------*/
```

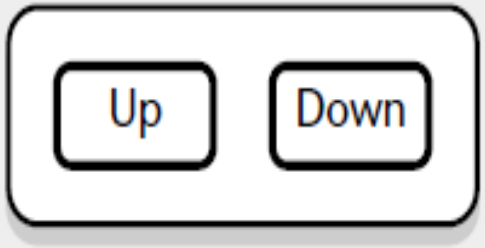
Contd..

Port Header (Port.H)

```
// Pins 3.0 and 3.1 used
// for RS-232 interface
```



```
// Switches
sbit Sw_up = P1^2;
sbit Sw_down = P1^3;
```



The Port Header file is simple to understand and easy to apply. Consider, for example, that we have three C files in a project (A, B, C), each of which require access to one or more port pins, or to a complete port.

Part of the ‘Hello, Embedded World’ example (restructured version)

```
/*-------------------------------------------------------*-
Port.H (v1.00)
-------------------------------------------------------
'Port Header' for project HELLO2 (see Chap 5)
-*-------------------------------------------------------*/
#ifndef _PORT_H
#define _PORT_H
// ------ LED_Flash.C ----------------------------------------
// Connect LED to this pin, via appropriate resistor
sbit LED_pin = P1^5;
#endif
/*-------------------------------------------------------*-
---- END OF FILE ----------------------------------------
-*-------------------------------------------------------*/
```

Part of the ‘Hello, Embedded World’ example (restructured version)

```
/*-----------------------------------------------------------*-
Main.C (v1.00)
-----------------------------------------------------------
A "Hello Embedded World" test program for 8051. (Re-structured version
– multiple source files)
-*-----------------------------------------------------------*/
#include "Main.H"
#include "Port.H"
#include "Delay_Loop.h"
#include "LED_Flash.h"
void main(void)
{
LED_FLASH_Init();
while(1)
{
// Change the LED state (OFF to ON, or vice versa)
LED_FLASH_Change_State();
```

Contd..

```
// Delay for *approx* 1000 ms
DELAY_LOOP_Wait(1000);
}
}
/*-------------------------------------------------------*-
---- END OF FILE ----------------------------------------
-*-------------------------------------------------------*/
```

Part of the 'Hello, Embedded World' example (restructured version)

```
/*-------------------------------------------------------*-
Main.C (v1.00)
-------------------------------------------------------
A "Hello Embedded World" test program for 8051. (Re-structured version –
multiple source files)
-*-------------------------------------------------------*/
#include "Main.H"
#include "Port.H")
```

Contd..

```
#include "Delay_Loop.h"
#include "LED_Flash.h"
void main(void)
{
LED_FLASH_Init();
while(1)
{
// Change the LED state (OFF to ON, or vice versa)
LED_FLASH_Change_State();
// Delay for *approx* 1000 ms
DELAY_LOOP_Wait(1000);
}
}
/*-------------------------------------------------------*-
---- END OF FILE ----------------------------------------
-*-------------------------------------------------------*/
```

Part of the 'Hello, Embedded World' example (restructured version)

```
/*------------------------------------------------------------*-
LED_flash.H (v1.00)
------------------------------------------------------------
– See LED_flash.C for details.
-*------------------------------------------------------------*/
#ifndef _LED_FLASH_H
#define _LED_FLASH_H
// ------ Public function prototypes ------------------------
void LED_FLASH_Init(void);
void LED_FLASH_Change_State(void);
#endif
/*------------------------------------------------------------*-
---- END OF FILE ----------------------------------------------
-*------------------------------------------------------------*/
```

Part of the 'Hello, Embedded World' example (restructured version)

```
/*-------------------------------------------------------*-
LED_flash.C (v1.00)
Simple 'Flash LED' test function.
-*-------------------------------------------------------*/
#include "Main.H"
#include "Port.H"
#include "LED_flash.H"
// ------ Private variable definitions ----------------------
static bit LED_state_G;
/*-------------------------------------------------------*-
LED_FLASH_Init()
Prepare for LED_Change_State() function – see below.
-*-------------------------------------------------------*/
void LED_FLASH_Init(void)
{
LED_state_G = 0;
}
/*-------------------------------------------------------*-
```

Contd..

LED_FLASH_Change_State()
Changes the state of an LED (or pulses a buzzer, etc) on a specified port pin. Must call at twice the required flash rate: thus, for 1 Hz flash (on for 0.5 seconds, off for 0.5 seconds) must call every 0.5 seconds.

```
-*-------------------------------------------------------*/
void LED_FLASH_Change_State(void)
{
// Change the LED from OFF to ON (or vice versa)
if (LED_state_G == 1)
{
LED_state_G = 0;
LED_pin = 0;
}
```

Contd..

```
else
{
LED_state_G = 1;
LED_pin = 1;
}
}
/*-----------------------------------------------------*-
---- END OF FILE ----------------------------------------
-*-----------------------------------------------------*/
```

Part of the ‘Hello, Embedded World’ example (restructured version)

```
/*-------------------------------------------------------*-
Delay_Loop.H (v1.00)
-------------------------------------------------------
– See Delay_Loop.C for details.
-*-------------------------------------------------------*/
#ifndef _DELAY_LOOP_H
#define _DELAY_LOOP_H
// ------ Public function prototype -------------------------
void DELAY_LOOP_Wait(const tWord);
#endif
/*-------------------------------------------------------*-
---- END OF FILE ----------------------------------------
-*-------------------------------------------------------*/
```

Part of the ‘Hello, Embedded World’ example (restructured version)

```
/*-----------------------------------------------------------*-
Delay_Loop.C (v1.00)
-----------------------------------------------------------
Create a simple software delay using a loop.
-*-----------------------------------------------------------*/
#include "Main.H"
#include "Port.H"
#include "Delay_loop.h"
/*-----------------------------------------------------------*-
DELAY_LOOP_Wait()
Delay duration varies with parameter.
Parameter is, *ROUGHLY*, the delay, in milliseconds,
on 12MHz 8051 (12 osc cycles).
You need to adjust the timing for your application!
-*-----------------------------------------------------------*/
```

Contd..

```
void DELAY_LOOP_Wait(const tWord DELAY_MS)
{
tWord x, y;
for (x = 0; x <= DELAY_MS; x++)
{
for (y = 0; y <= 120; y++);
}
}
/*-------------------------------------------------------*-
---- END OF FILE ----------------------------------------
-*-------------------------------------------------------*/
```

UNIT-III
EMBEDDED C APPLICATIONS

CLOs	Course Learning Outcome
CLO9	Analyse the programming on switches
CLO10	Understanding the programming language tools.
CLO11	Understand different concepts of display and keyboard interfacing using embedded C.
CLO12	Understand different concepts of stepper motor interfacing.

Part of the ‘Hello, Embedded World’ example (restructured version)

```
/*-------------------------------------------------------*-
Port.H (v1.00)
-------------------------------------------------------
'Port Header' for project HELLO2 (see Chap 5)
-*-------------------------------------------------------*/
#ifndef _PORT_H
#define _PORT_H
// ------ LED_Flash.C ----------------------------------------
// Connect LED to this pin, via appropriate resistor
sbit LED_pin = P1^5;
#endif
/*-------------------------------------------------------*-
---- END OF FILE ------------------------------------------
-*-------------------------------------------------------*/
```

Part of the ‘Hello, Embedded World’ example (restructured version)

```
/*-----------------------------------------------------------*-
Main.C (v1.00)
-----------------------------------------------------------
A "Hello Embedded World" test program for 8051. (Re-structured version
– multiple source files)
-*-----------------------------------------------------------*/
#include "Main.H"
#include "Port.H"
#include "Delay_Loop.h"
#include "LED_Flash.h"
void main(void)
{
LED_FLASH_Init();
while(1)
{
// Change the LED state (OFF to ON, or vice versa)
LED_FLASH_Change_State();
```

Contd..

```
// Delay for *approx* 1000 ms
DELAY_LOOP_Wait(1000);
}
}
/*-------------------------------------------------------*-
---- END OF FILE ----------------------------------------
-*-------------------------------------------------------*/
```

Part of the 'Hello, Embedded World' example (restructured version)

```
/*-------------------------------------------------------*-
Main.C (v1.00)
-------------------------------------------------------
A "Hello Embedded World" test program for 8051. (Re-structured version – multiple source files)
-*-------------------------------------------------------*/
#include "Main.H"
#include "Port.H")
```

Contd..

```
#include "Delay_Loop.h"
#include "LED_Flash.h"
void main(void)
{
LED_FLASH_Init();
while(1)
{
// Change the LED state (OFF to ON, or vice versa)
LED_FLASH_Change_State();
// Delay for *approx* 1000 ms
DELAY_LOOP_Wait(1000);
}
}
/*-------------------------------------------------------*-
---- END OF FILE ----------------------------------------
-*-------------------------------------------------------*/
```

Part of the 'Hello, Embedded World' example (restructured version)

```
/*------------------------------------------------------------*-
LED_flash.H (v1.00)
------------------------------------------------------------
– See LED_flash.C for details.
-*------------------------------------------------------------*/
#ifndef _LED_FLASH_H
#define _LED_FLASH_H
// ------ Public function prototypes ------------------------
void LED_FLASH_Init(void);
void LED_FLASH_Change_State(void);
#endif
/*------------------------------------------------------------*-
---- END OF FILE ----------------------------------------------
-*------------------------------------------------------------*/
```

Part of the ‘Hello, Embedded World’ example (restructured version)

```
/*-----------------------------------------------------*-
LED_flash.C (v1.00)
Simple 'Flash LED' test function.
-*-----------------------------------------------------*/
#include "Main.H"
#include "Port.H"
#include "LED_flash.H"
// ------ Private variable definitions ----------------------
static bit LED_state_G;
/*-----------------------------------------------------*-
LED_FLASH_Init()
Prepare for LED_Change_State() function – see below.
-*-----------------------------------------------------*/
void LED_FLASH_Init(void)
{
LED_state_G = 0;
}
/*-----------------------------------------------------*-
```

Contd..

LED_FLASH_Change_State()
Changes the state of an LED (or pulses a buzzer, etc) on a specified port pin. Must call at twice the required flash rate: thus, for 1 Hz flash (on for 0.5 seconds, off for 0.5 seconds) must call every 0.5 seconds.

```
-*-------------------------------------------------------*/
void LED_FLASH_Change_State(void)
{
// Change the LED from OFF to ON (or vice versa)
if (LED_state_G == 1)
{
LED_state_G = 0;
LED_pin = 0;
}
```

Contd..

```
else
{
LED_state_G = 1;
LED_pin = 1;
}
}
/*-----------------------------------------------------*-
---- END OF FILE ----------------------------------------
-*-----------------------------------------------------*/
```

Part of the ‘Hello, Embedded World’ example (restructured version)

```
/*-------------------------------------------------------*-
Delay_Loop.H (v1.00)
-------------------------------------------------------
– See Delay_Loop.C for details.
-*-------------------------------------------------------*/
#ifndef _DELAY_LOOP_H
#define _DELAY_LOOP_H
// ------ Public function prototype -------------------------
void DELAY_LOOP_Wait(const tWord);
#endif
/*-------------------------------------------------------*-
---- END OF FILE ------------------------------------------
-*-------------------------------------------------------*/
```

Part of the ‘Hello, Embedded World’ example (restructured version)

```
/*-----------------------------------------------------------*-
Delay_Loop.C (v1.00)
-----------------------------------------------------------
Create a simple software delay using a loop.
-*-----------------------------------------------------------*/
#include "Main.H"
#include "Port.H"
#include "Delay_loop.h"
/*-----------------------------------------------------------*-
DELAY_LOOP_Wait()
Delay duration varies with parameter.
Parameter is, *ROUGHLY*, the delay, in milliseconds,
on 12MHz 8051 (12 osc cycles).
You need to adjust the timing for your application!
-*-----------------------------------------------------------*/
```

Contd..

```
void DELAY_LOOP_Wait(const tWord DELAY_MS)
{
tWord x, y;
for (x = 0; x <= DELAY_MS; x++)
{
for (y = 0; y <= 120; y++);
}
}
/*-------------------------------------------------------*-
---- END OF FILE ----------------------------------------
-*-------------------------------------------------------*/
```

Basic techniques for reading from port pins

Control of the 8051 ports is carried out using 8-bit latches (SFRs). We can send some data to Port 1 as follows:

```
sfr P1 = 0x90; // Usually in header file
P1 = 0x0F; // Write 00001111 to Port 1
```

In exactly the same way, we can read from Port 1 as follows:

```
unsigned char Port_data;
P1 = 0xFF; // Set the port to 'read mode'
Port_data = P1; // Read from the port
```

After the 8051 microcontroller is reset, the port latches all have the value 0xFF(11111111 in binary): that is, all the port-pin latches are set to values of '1'. It is tempting to assume that writing data to the port is therefore unnecessary, and that we can get away with the following version:

```
unsigned char Port_data;
// Assume nothing written to port since reset
// – DANGEROUS!!!
Port_data = P1;
```

Contd..

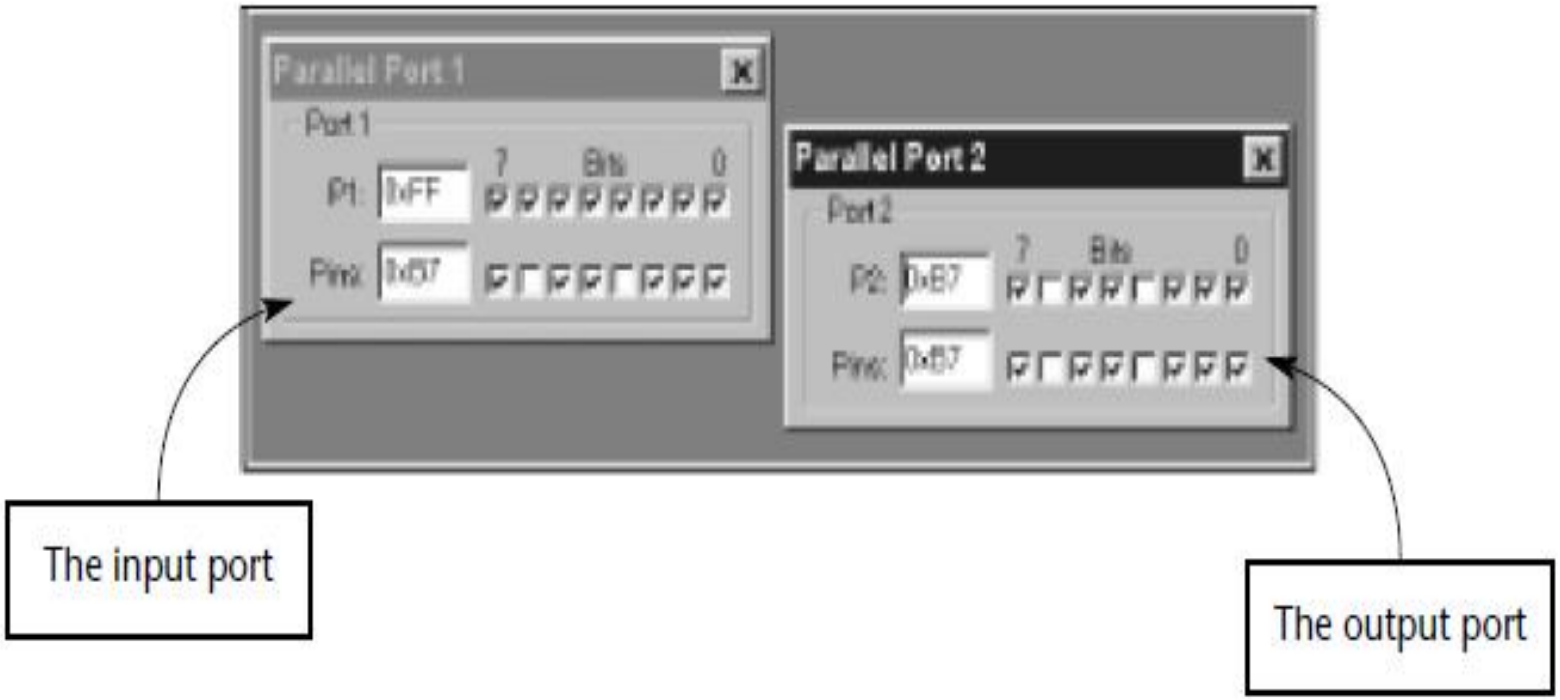
The problem with this code is that, in simple test programs it works: this can Pull the developer into a false sense of security. If, at a later date, someone modifies the program to include a routine for writing to all or part of the same port, this code will not generally work as required:

```
unsigned char Port_data;
P1 = 0x00;
. . .
// Assumes nothing written to port since reset
// – WON'T WORK
Port_data = P1;
```

In most cases, initialization functions are used to set the port pins to a known state at the start of the program. Where this is not possible, it is safer to always write '1' to any port pin before reading from it.

Reading and writing bytes

Listing is a simple example which illustrates how we can read from one port on an 8051 microcontroller and ‘echo’ the result on another port. Figure shows the output from one such simulation.



Contd..

A simple 'Super Loop' application which copies the values from P1 to P2.

```
/*-------------------------------------------------------------*-

   Bytes.C (v1.00)

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

   Reads from P1 and copies the value to P2.

-*-------------------------------------------------------------*/

#include <Reg52.H>

/* ..............................................................*/

void main (void)
   {
   unsigned char Port1_value;

   // Must set up P1 for reading
   P1 = 0xFF;
```

Contd..

```
  while(1)
    {
    // Read the value of P1
    Port1_value = P1;

    // Copy the value to P2
    P2 = Port1_value;
    }
  }

/*-------------------------------------------------------------*-
 ---- END OF FILE ------------------------------------------
-*-------------------------------------------------------------*/
```

Reading and writing bits (simple version)

However, suppose we have a switch connected to Pin 1.0 and an LED connected to Pin 1.1. We might also have input and output devices connected to the other pins on Port 1. These pins may be used by totally different parts of the same system, and the code to access them may be produced by other team members, or other companies. It is therefore essential that we are able to read-from or write-to individual port pins without altering the values of other pins on the same port.

Listing 4.2 goes one step further and illustrates how we can read from Pin 1.0, and write to Pin 1.1, without disrupting any other pins on this (or any other) port.

Reading and writing bits (simple version)

```
/*-------------------------------------------------------*-
Bits1.C (v1.00)
-------------------------------------------------------
Reading and writing individual port pins.
NOTE: Both pins on the same port
-*-------------------------------------------------------*/
```

Contd..

```
#include <Reg52.H>

sbit Switch_pin = P1^0;
sbit LED_pin = P1^1;

/*  ..................................................  */

void main (void)
   {
   bit x;

   // Set switch pin for reading
   Switch_pin = 1;

   while(1)
     {
     x = Switch_pin; // Read Pin 1.0
     LED_pin = x;    // Write to Pin 1.1
     }
   }

/*-------------------------------------------------------*-
  ---- END OF FILE ----------------------------------------
-*-------------------------------------------------------*/
```

Contd..

Please note these lines:

sbit Switch_pin = P1^0;

sbit LED_pin = P1^1;

Here we gain access to two port pins through the use of an sbit variable declaration. The symbol '^' is used, but the XOR bitwise operator is NOT involved.

Reading and writing bits (generic version). See text for details

```
/*-------------------------------------------------------------*-

   Bits2.C (v1.00)

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

   Reading and writing individual port pins.

   NOTE: Both pins on the same port

   --- Generic version ---

-*-------------------------------------------------------------*/
```

Contd..

```
#include <reg52.H>

// Function prototypes
void Write_Bit_P1(const unsigned char, const bit);
bit Read_Bit_P1(const unsigned char);

/* ..................................................... */

void main (void)
   {
   bit x;

   while(1)
     {
     x = Read_Bit_P1(0); // Read Port 1, Pin 0
     Write_Bit_P1(1,x);  // Write to Port 1, Pin 1
     }
   }

/* ------------------------------------------------------ */
```

Contd..

```
void Write_Bit_P1(const unsigned char PIN, const bit VALUE)
   {
   unsigned char p = 0x01; // 00000001

   // Left shift appropriate number of places
   p <<= PIN;

   // If we want 1 output at this pin
   if (VALUE == 1)

     {
     P1 |= p; // Bitwise OR
     return;
     }

  // If we want 0 output at this pin
  p = ~p; // Complement
  P1 &= p; // Bitwise AND
  }
```

Contd..

```
/* ------------------------------------------------------------ */

bit Read_Bit_P1(const unsigned char PIN)
   {
   unsigned char p = 0x01; // 00000001

   // Left shift appropriate number of places
   p <<= PIN;

   // Write a 1 to the pin (to set up for reading)
   Write_Bit_P1(PIN, 1);

   // Read the pin (bitwise AND) and return
   return (P1 & p);
   }

/*------------------------------------------------------------*-
  ---- END OF FILE ----------------------------------------
-*------------------------------------------------------------*/
```

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….

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.

Multitasking….

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.

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.

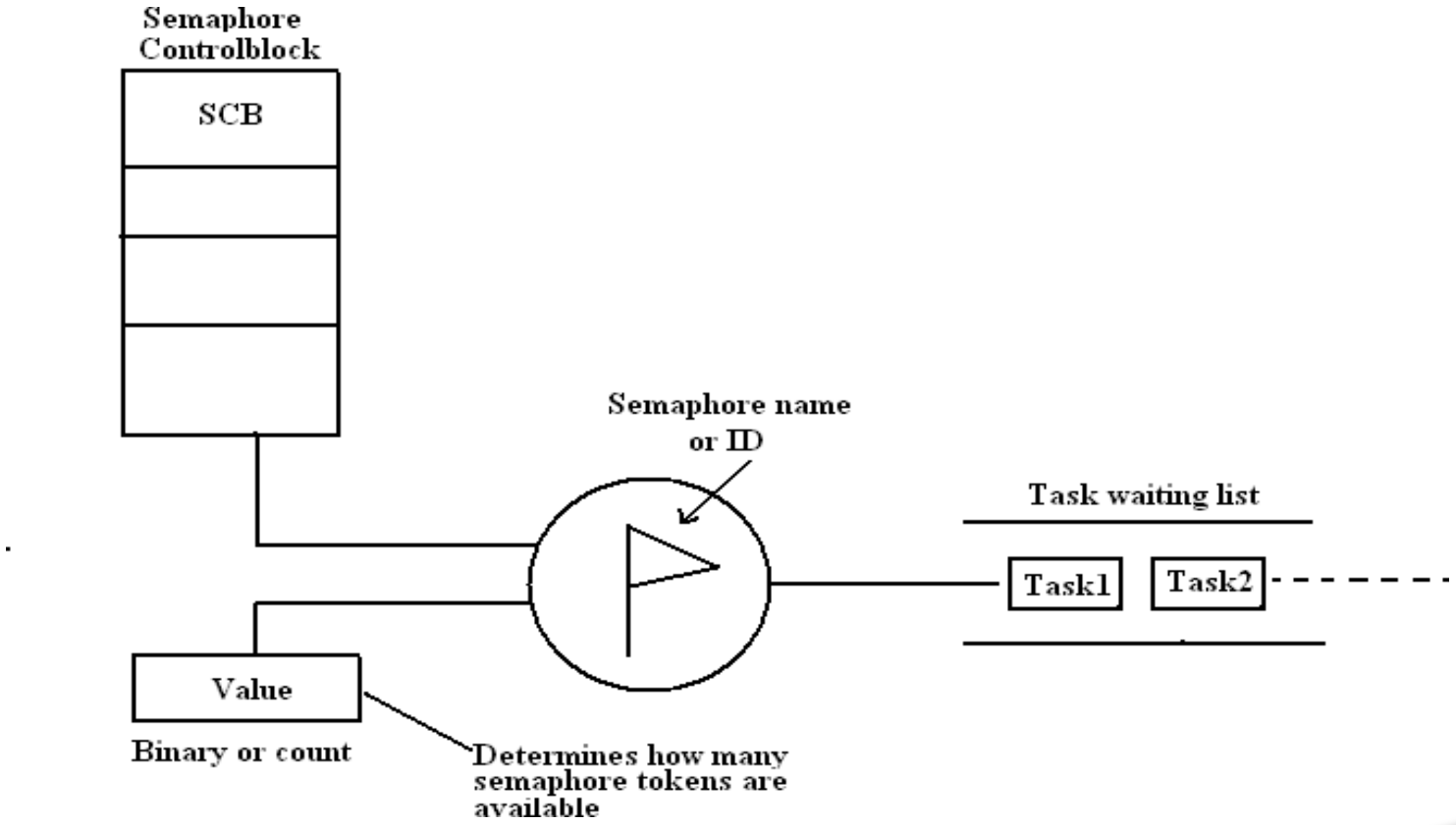
Semaphores which allow an arbitrary resource count are called counting semaphores, whilst semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores.

UNIT-IV

EMBEDDED SOFTWARE DEVELOPMENT TOOLS

CLOs	Course Learning Outcome
CLO13	Understand and analyze the RTOS concepts for firmware development.
CLO14	Remember how to choose an RTOS, task scheduling, semaphores and queues, hard real-time scheduling considerations.
CLO15	Understand the task communication, its programming and Task synchronization with its issues and techniques.
CLO16	Develop host and target machines for linking to embedded software.
CLO17	Develop debugging techniques for testing on host machine with examples.

Semaphores…


Semaphore
Controlblock
SCB
Semaphore name
or ID
Task waiting list
Task1
Task2
Value
Binary or count
Determines how many
semaphore tokens are
available

Semaphores…

Types of Semaphores: There are three types of semaphores

- Binary Semaphores,
- Counting Semaphores and
- Mutexes.

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.

Message Queues…

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

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

Saving Memory and Power…

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

Saving Memory and Power…

Data Memory

Consider using more static variables instead of stack variables

On 8-bit processors, use char instead of int when possible.

Saving Memory and Power…

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.

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.

Saving Memory and Power…

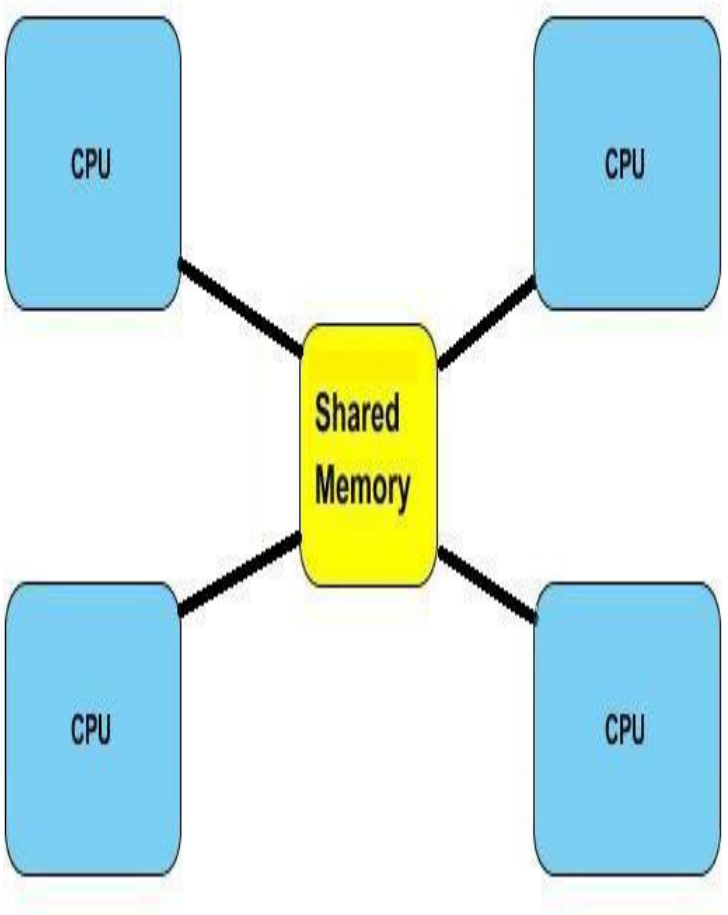
Shared memory:

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.

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.

Remote Procedure Call (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.

Sockets…

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.

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.

HOST AND TARGET MACHINES

Host:

Where the embedded software is developed, compiled, tested, debugged, optimized, and prior to its translation into target device. (Because the host has keyboards, editors, monitors, printers, more memory, etc. for development, while the target may have not of these capabilities for developing the software.)

Target:

After development, the code is cross-compiled, translated – cross-assembled, linked (into target processor instruction set) and ***located*** into the target

HOST AND TARGET MACHINES

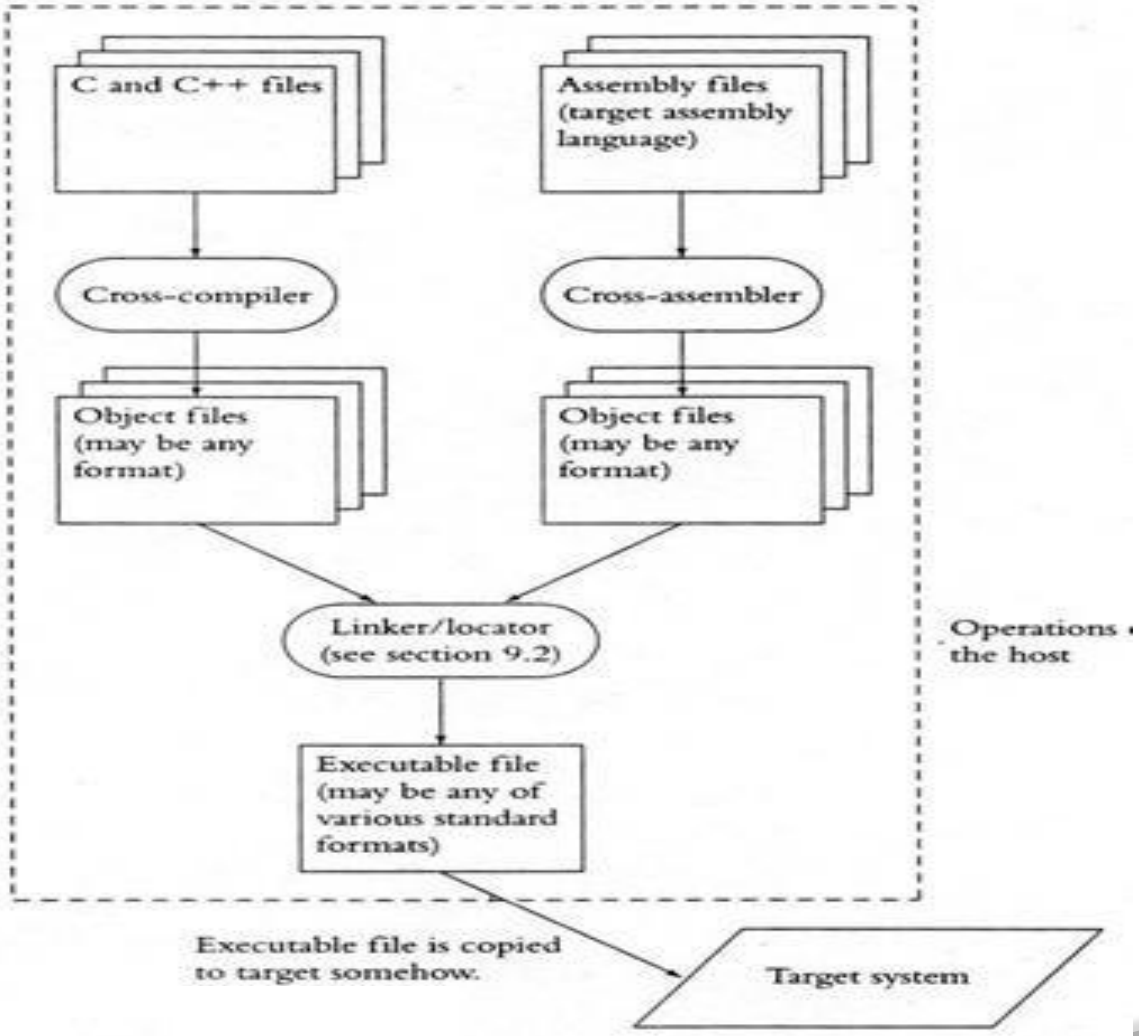
- **Cross-Compilers :**

  - Native tools are good for host, but to port/locate embedded code to target, the host must have a tool-chain that includes a cross-compiler, one which runs on the host but produces code for the target processor
  - Cross-compiling doesn't guarantee correct target code due to (e.g., differences in word sizes, instruction sizes, variable declarations, library functions)

HOST AND TARGET MACHINES

- **Cross-Assemblers and Tool Chain:**
  - Host uses cross-assembler to assemble code in target's instruction syntax for the target
  - Tool chain is a collection of compatible, translation tools, which are 'pipelined' to produce a complete binary/machine code that can be linked and located into the target processor

HOST AND TARGET MACHINES



LINKERS AND LOCATORS

Linker/Locators for Embedded Software:

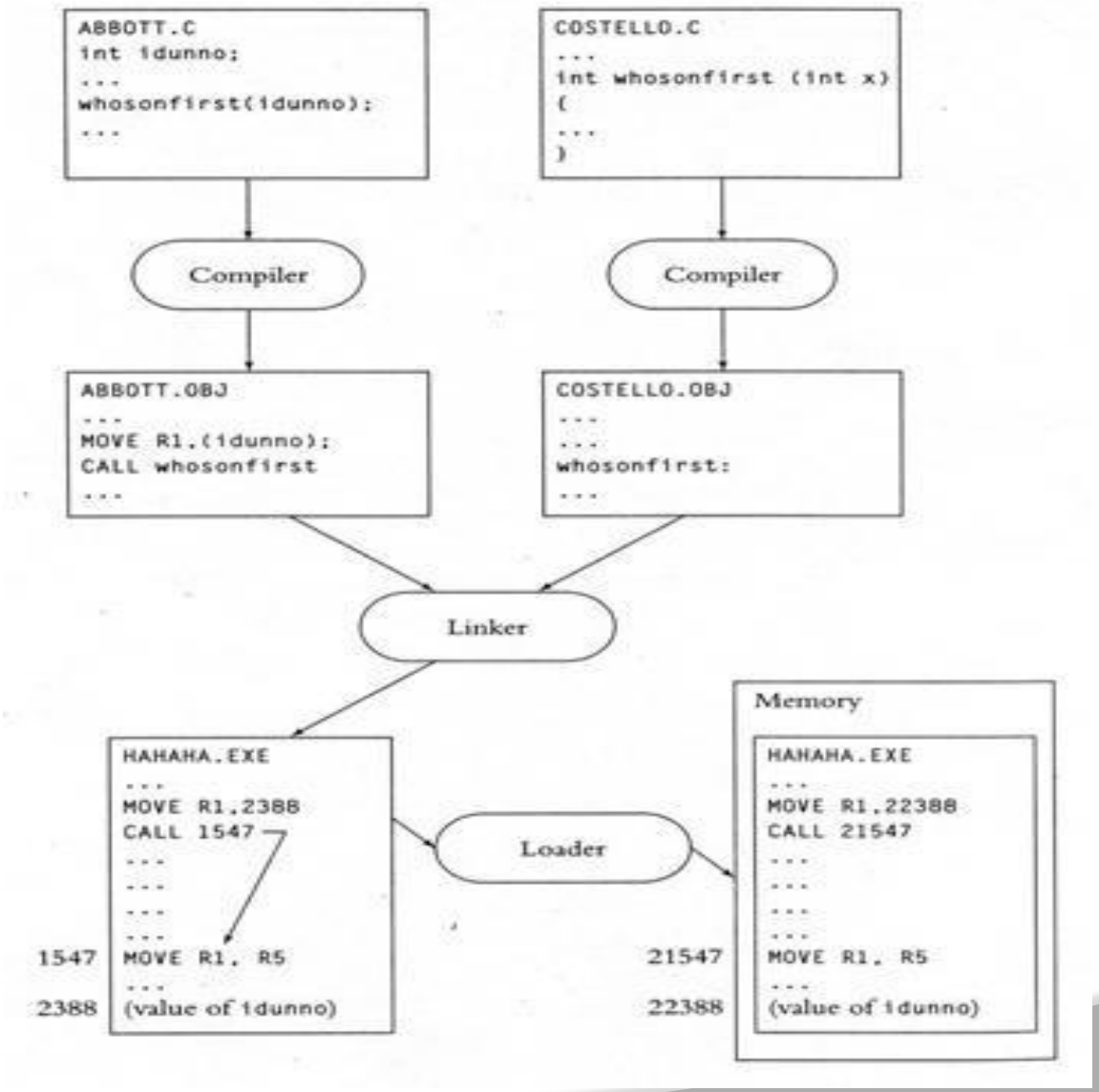
- Native linkers are different from cross-linkers (or locators) that perform additional tasks to ***locate*** embedded binary code into target processors
- Address Resolution –
  - Native Linker: produces host machine code on the hard-drive (in a named file), which the loader loads into RAM, and then schedules (under the OS control) the program to go to the CPU.

LINKERS AND LOCATORS

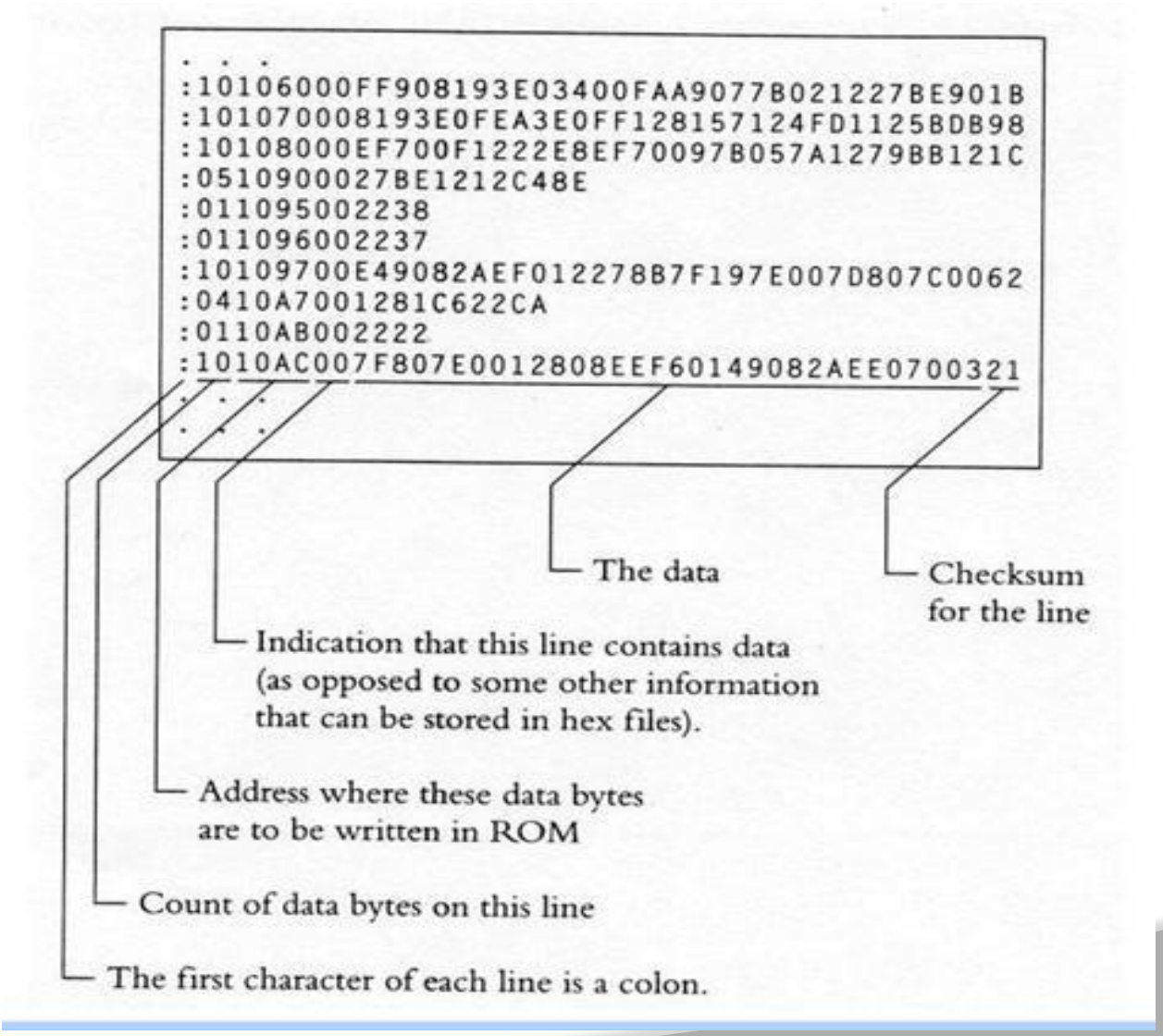
Linker/Locators for Embedded Software:

- Function calls, are ordered or organized by the linker. The loader then maps the logical addresses into physical addresses a process called **address resolution**. The loader then loads the code accordingly into RAM . In the process the loader also resolves the addresses for calls to the native OS routines
- Locator: produces target machine code (which the locator glues into the RTOS) and the combined code (called **map**) gets copied into the target ROM. The locator doesn't stay in the target environment, hence all addresses are resolved, guided by locating-tools and directives, prior to running the code.

LINKERS AND LOCATORS



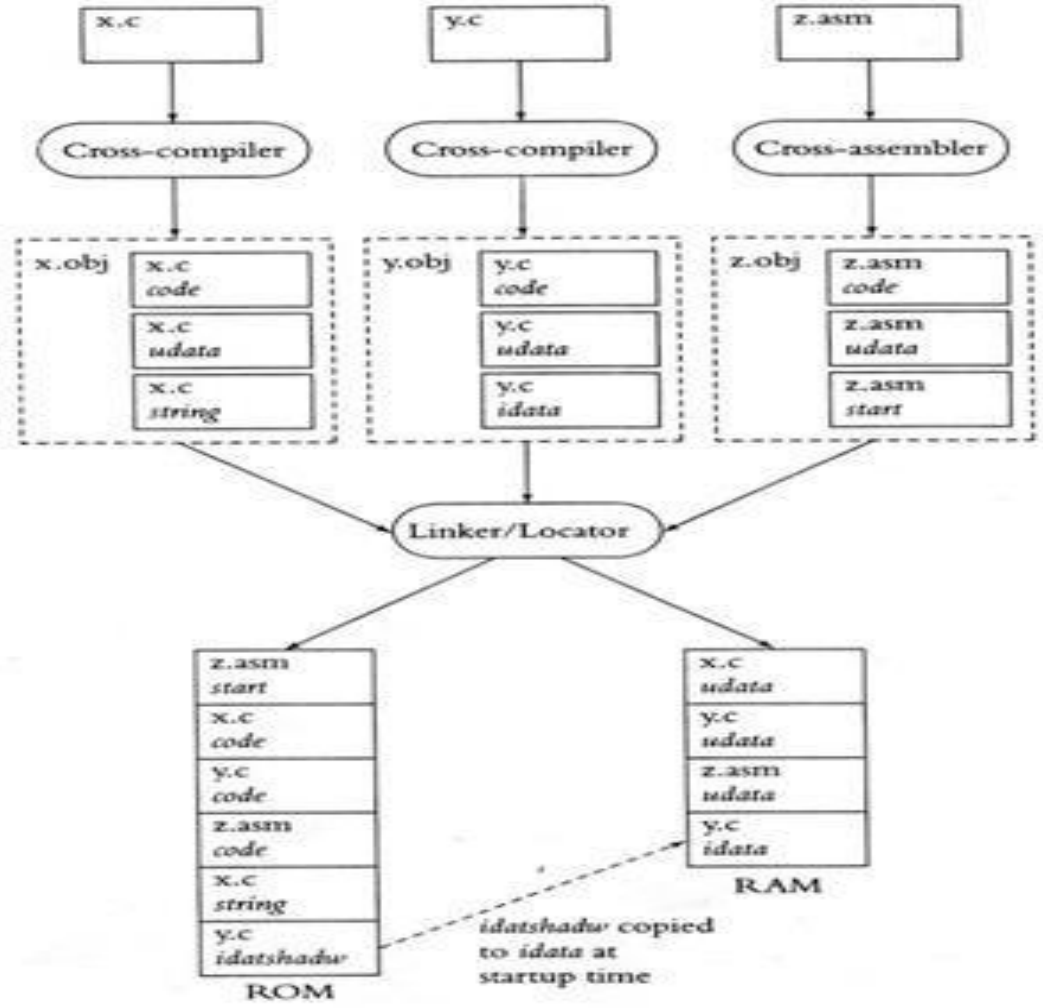
HOST AND TARGET MACHINES



LINKERS AND LOCATORS

- Locating Program Components – Segments
- Unchanging embedded program (binary code) and constants must be kept in ROM to be remembered even on power-off
- Changing program segments (e.g., variables) must be kept in RAM
- Chain tools separate program parts using **segments** concept
- Chain tools (for embedded systems) also require a ‘start-up’ code to be in a separate segment and ‘located’ at a microprocessor-defined location where the program starts execution
- Some cross-compilers have default or allow programmer to specify segments for program parts, but cross-assemblers have no default behavior and programmer must specify segments for program parts

HOST AND TARGET MACHINES

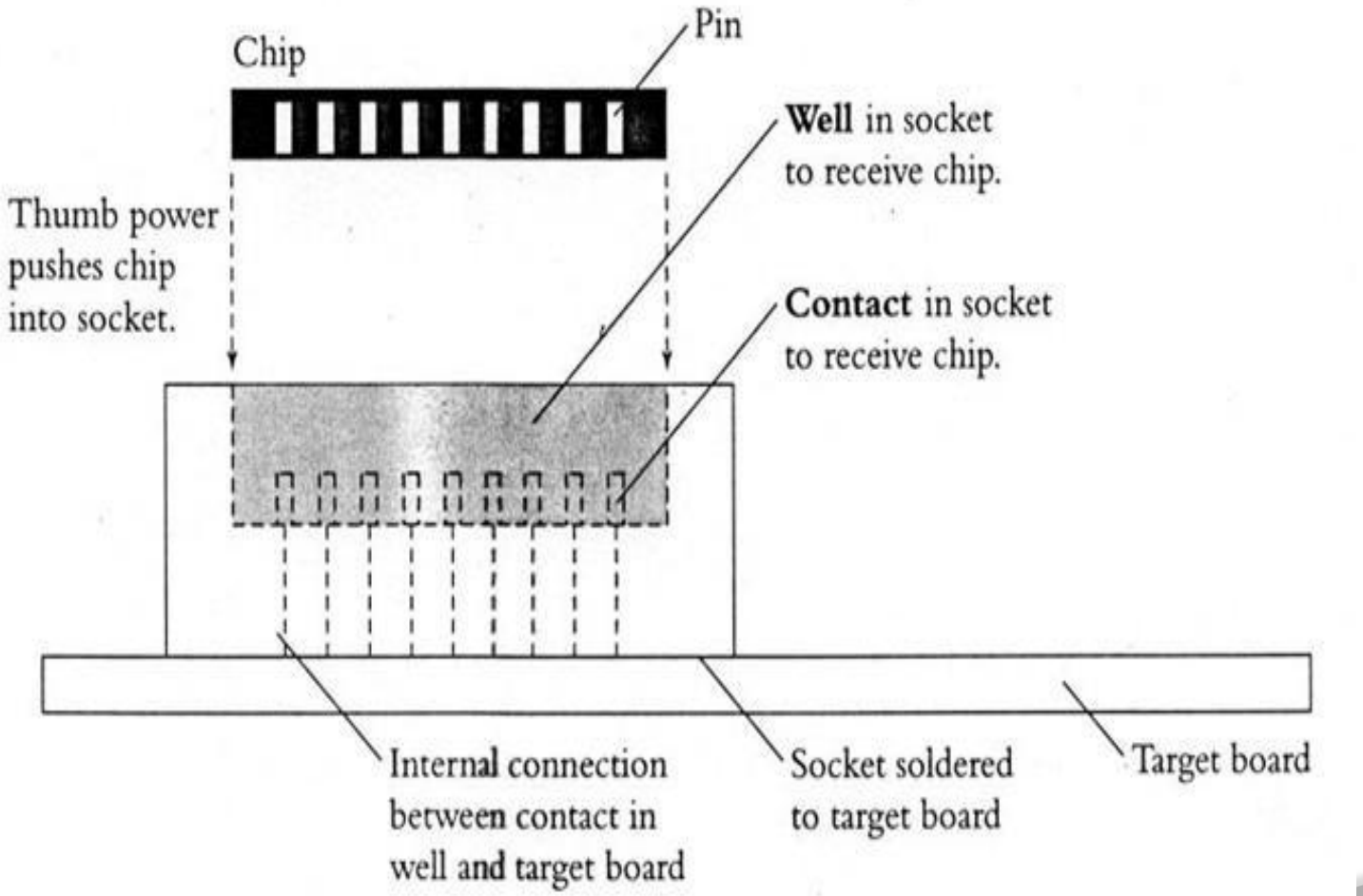


GETTING EMBEDDED SOFTWARE INTO TARGET SYSTEM

Getting Embedded Software into Target System

- Moving maps into ROM or PROM, is to create a ROM using hardware tools or a PROM programmer (for small and changeable software, during debugging)
- If PROM programmer is used (for changing or debugging software), place PROM in a **socket** (which makes it erasable – for EPROM, or removable/replaceable) rather than 'burnt' into circuitry
- PROM's can be pushed into sockets by hand, and pulled using a chip puller
- The PROM programmer must be compatible with the format (syntax/semantics) of the Map

GETTING EMBEDDED SOFTWARE INTO TARGET SYSTEM



Getting Embedded Software into Target System – 1

- ROM Emulators – Another approach is using a ROM emulator (hardware) which emulates the target system, has all the ROM circuitry, and a serial or network interface to the host system. The locator loads the Map into the emulator, especially, for debugging purposes.

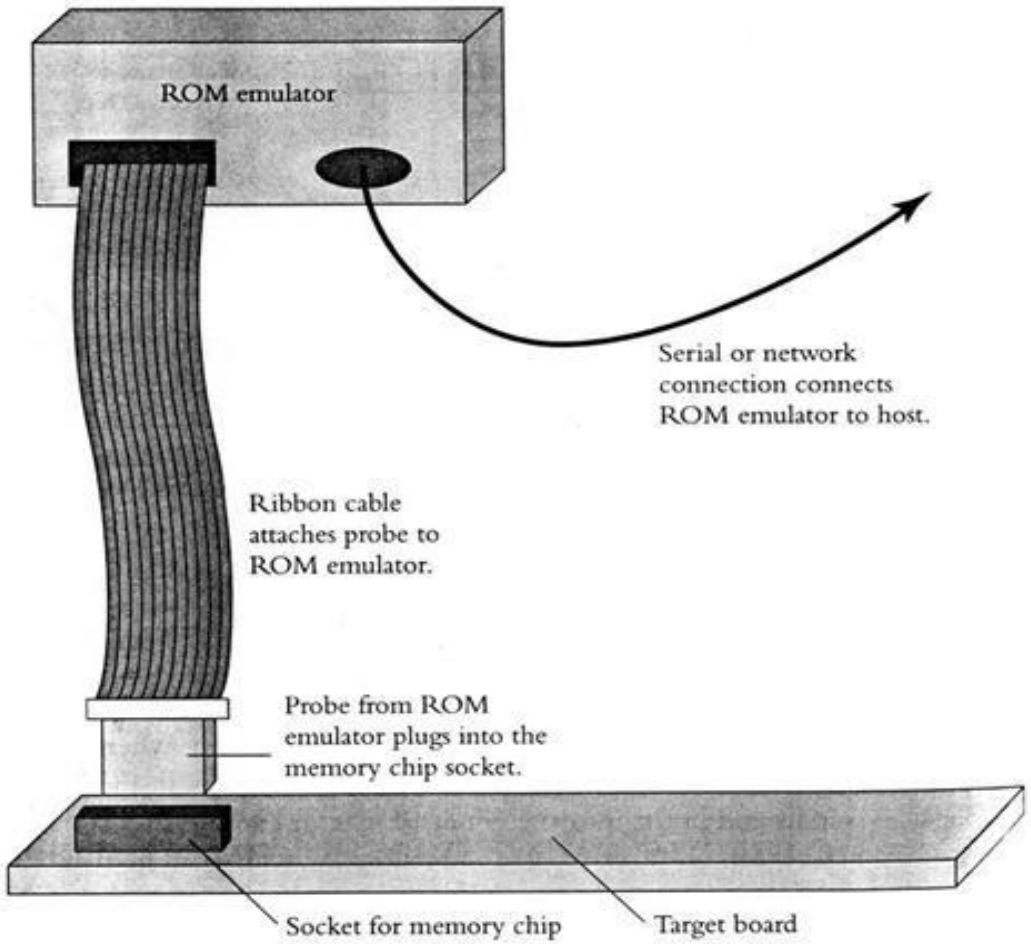
- Software on the host that loads the Map file into the emulator must understand (be compatible with) the Map's syntax/semantics

DEBUGGING TECHNIQUES

- **Getting Embedded Software into Target System – 1**

- Using Flash Memory
- For debugging, a flash memory can be loaded with target Map code using a software on the host over a serial port or network connection (just like using an EPROM)

DEBUGGING TECHNIQUES



DEBUGGING TECHNIQUES

Advantages:

- No need to pull the flash (unlike PROM) for debugging different embedded code
- Transferring code into flash (over a network) is faster and hassle-free
- Modifying and/or debugging the flash programming software requires moving it into RAM, modify/debug, and reloading it into target flash memory using above methods

DEBUGGING TOOLS

Advantages:

New versions of embedded software (supplied by vendor) can be loaded into flash memory by customers over a network - Requires a) protecting the flash programmer, saving it in RAM and executing from there, and reloading into flash after new version is written and b) the ability to complete loading new version even if there are crashes and protecting the startup code as in (a)

DEBUGGING TECHNIQUES

Advantages:

- No need to pull the flash (unlike PROM) for debugging different embedded code
- Transferring code into flash (over a network) is faster and hassle-free
- Modifying and/or debugging the flash programming software requires moving it into RAM, modify/debug, and reloading it into target flash memory using above methods

SIMPLE VOLT/OHM METER

- Simple volt-ohm meter can be used to test the target hardware.
- It has two leds red and black
- One end is connected to meter and other is connected to point between which the voltage or resistance is to be measured
- The meter is set for volt for checking the power supply voltage at sorce and voltage level at chips and port pins.
- The meter is set for ohm for checking thebroken connections,improper ground connections,burn out resistance and diods.

SIMPLE LED TESTS AND LOGIC PROBE

- A logic probe is a hand-held test probe used for analyzing and troubleshooting the logical states (boolean 0 or 1) of a digital circuit.

- Most modern logic probes typically have one or more LEDs on the body of the probe:
  - an LED to indicate a high (1) logic state.
  - an LED to indicate a low (0) logic state.
  - an LED to indicate changing back and forth between low and high states.

OSCILLOSCOPE

- An 'oscilloscope', previously called an 'oscillograph', and informally known as a scope or o-scope, CRO (for cathode-ray oscilloscope), or DSO (for the more modern digital storage oscilloscope), is a type of electronic test instrument that graphically displays varying signal voltage, usually as a two-dimensional plot of one or more signals as a function of time. Other signals (such as sound or vibration) can be converted to voltages and displayed.
- Oscilloscopes display the change of an electrical signal over time, with voltage and time as the Y- and X-axes, respectively, on a calibrated scale.

OSCILLOSCOPE

- The waveform can then be analyzed for properties such as amplitute, f requency, rise time , time interval, distortion, and others.
- The oscilloscope can be adjusted so that repetitive signals can be observed as a continuous shape on the screen.
- A storage oscilloscope can capture a single event and display it continuously, so the user can observe events that would otherwise appear too briefly to see directly.
- Oscilloscopes are used in the sciences, medicine, engineer

BIT RATE METER

- In telecommunications and computng , bit rate (bit rate or as a variable *R*) is the number of bits that are conveyed or processed per unit of time.
- The bit rate is quantified using the bits per second unit (symbol: "bit/s"), often in conjunction with an SI prefix such as "kilo" (1 kbit/s = 1,000 bit/s), "mega" (1 Mbit/s = 1,000 kbit/s), "giga" (1 Gbit/s = 1,000 Mbit/s) or "tera" (1 Tbit/s = 1000 Gbit/s). The non-standard abbreviation "bps" is often used to replace the standard symbol "bit/s", so that, for example, "1 Mbps" is used to mean one million bits per second.

BIT RATE METER

The bit rate is calculated using the formula:

1.Frequency × bit depth × channels = bit rate.

2.44,100 samples per second × 16 bits per sample × 2 channels = 1,411,200 bits per second (or 1,411.2 kbps)

3.14,411,200 × 240 = 338,688,000 bits (or 40.37 megabytes)

LOGIC ANALYZER

- A **logic analyzer** is an electronic instrument that captures and displays multiple signals from a digital system or digital circuit.

- A logic analyzer may convert the captured data into timing diagrams, protocol decodes, state machine traces, assembly language, or may correlate assembly with source-level software.

- Logic analyzers have advanced triggering capabilities, and are useful when a user needs to see the timing relationships between many signals in a digital system

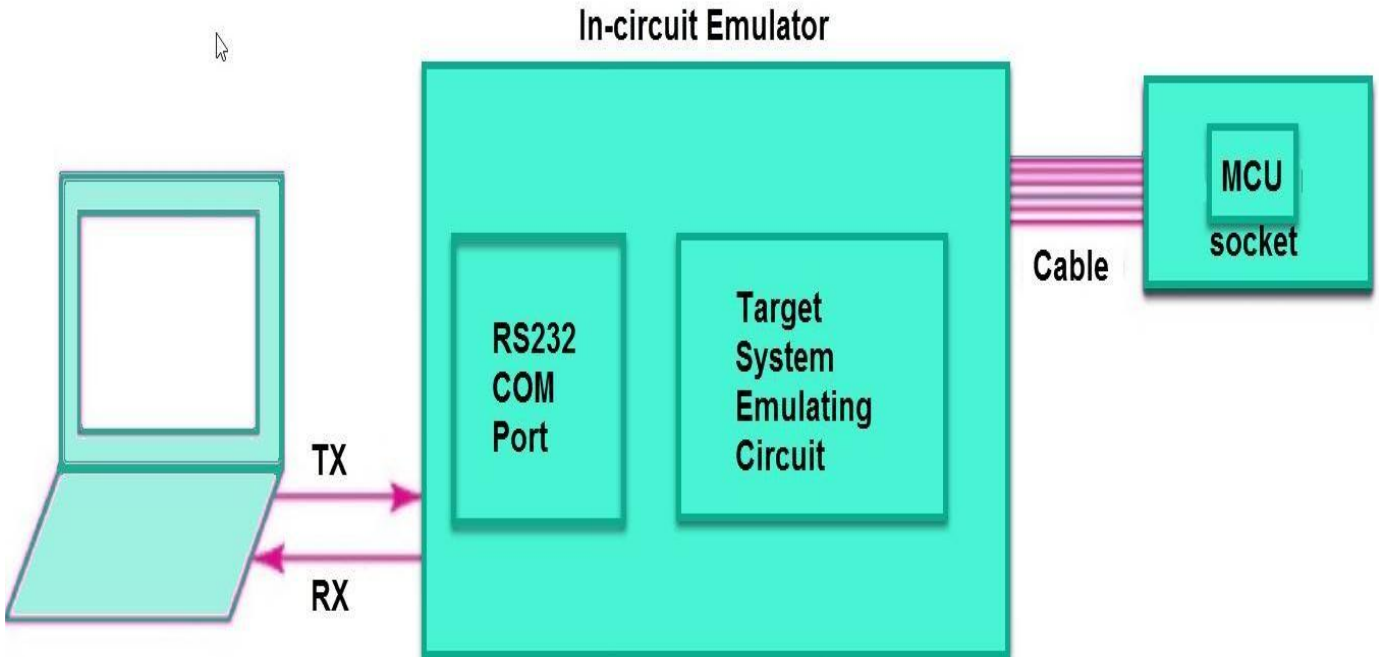
IN-CIRCUIT EMULATOR

- An In-circuit emulator (ICE) is a debugging tool that allows you to access a target MCU for in-depth debugging.

- In-circuit emulation (ICE) is the use of a hardware device or in-circuit emulator used to debug the software of an embedde system.

- It operates by using a processor with the additional ability to support debugging operations, as well as to carry out the main function of the system.

IN-CIRCUIT EMULATOR



IN-CIRCUIT EMULATOR

- ICE consists of a hardware board with accompanying software for the host computer. The ICE is physically connected between the host computer and the target MCU.
- The debugger on the host establishes a connection to the MCU via the ICE. ICE allows a developer to see data and signals that are internal to the MCU, and to step through the source code (e.g., C/C++ on the host) or set breakpoints; the immediate ramifications of executed software are observed during run time.
- Since the debugging is done via hardware, not software, the MCU's performance is left intact for the most part, and ICE does not compromise MCU resources.

MONITOR

- Monitor is a debugging tool for actual target microprocessor or microcontroller in ICE ROM emulator or in target development board.

- It also lets host system debugging interface just like as an ICE.

- Monitor means a ROM resident program at the target board or ROM emulator connected to ICE.It monitors the device applications ,the runs for different hardware architecture and is used for debugging.

UNIT-V
INTRODUCTION TO ADVANCED ARCHITECTURES

CLOs	Course Learning Outcome
CLO18	Remember the advanced processors such as ARM and SHARC.
CLO19	Understand the bus protocols such as I2C and CAN bus.
CLO20	Design an application based on advanced technological changes.

ARM instruction set

- ARM versions.
- ARM assembly language.
- ARM programming model.
- ARM memory organization.
- ARM data operations.
- ARM flow of control

ARM versions

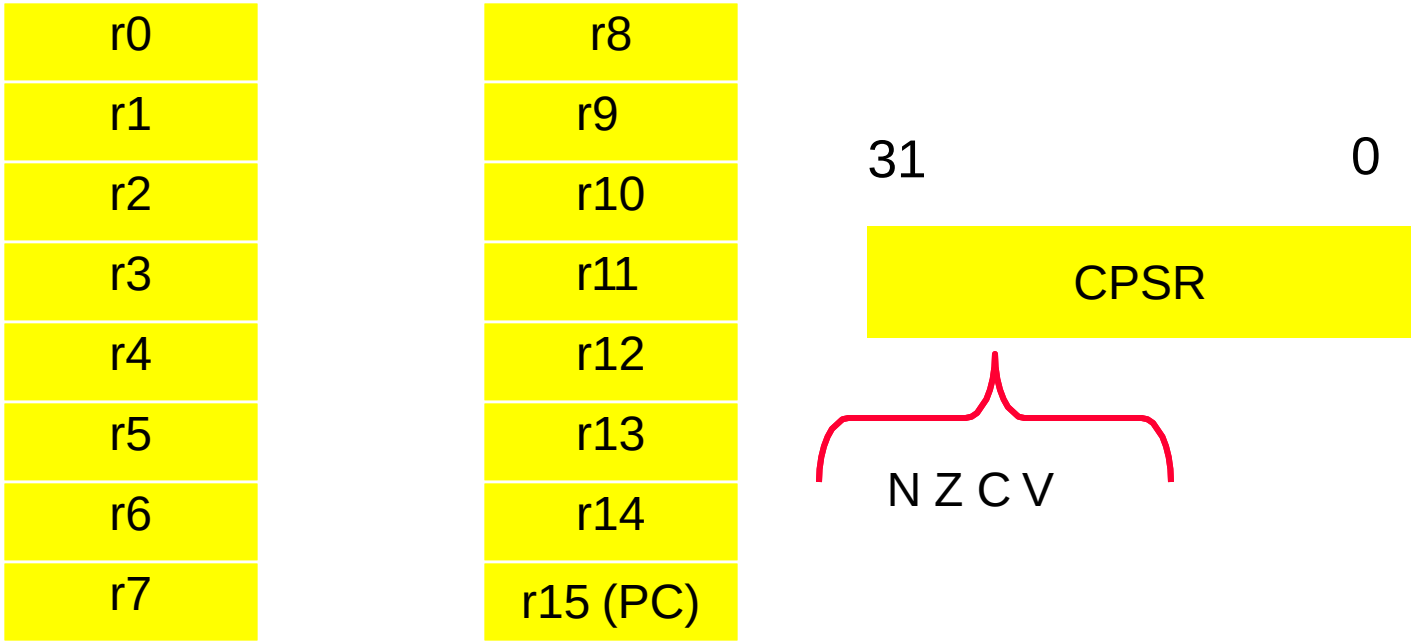
- ARM architecture has been extended over several versions.
- We will concentrate on ARM7.

ARM assembly language

- **Fairly standard assembly language:**

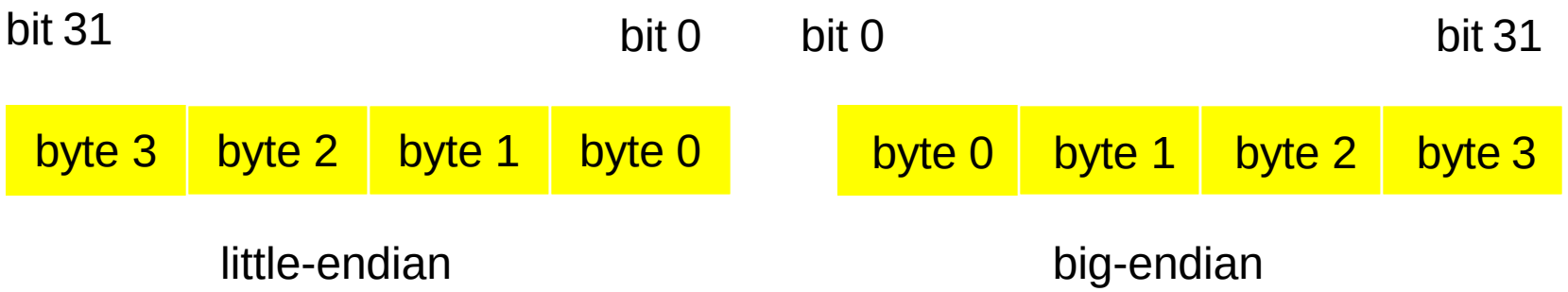
```
LDR r0,[r8] ; a comment
label   ADD r4,r0,r1
```

ARM programming model



Endianness

- **Relationship between bit and byte/word ordering defines endianness:**



ARM data types

- Word is 32 bits long.
- Word can be divided into four 8-bit bytes.
- ARM addresses can be 32 bits long.
- Address refers to byte.
- Address 4 starts at byte 4.
- Can be configured at power-up as either little- or bit-endian mode.

ARM status bits

- Every arithmetic, logical, or shifting operation sets CPSR bits:
  N (negative), Z (zero), C (carry), V (overflow).
- Examples:
  -1 + 1 = 0: NZCV = 0110.
  2^{31}-1+1 = -2^{31}: NZCV = 1001.

ARM data instructions

- Basic format:

  ADD r0,r1,r2

  -Computes r1+r2, stores in r0.

- Immediate operand:

  ADD r0,r1,#2

  -Computes r1+2, stores in r0.

ARM data instructions

- ADD, ADC : add (w. carry)
- SUB, SBC : subtract (w. carry)
- RSB, RSC : reverse subtract (w. carry)
- MUL, MLA : multiply (and accumulate)
- AND, ORR, EOR
- BIC : bit clear
- LSL, LSR : logical shift left/right
- ASL, ASR : arithmetic shift left/right
- ROR : rotate right
- RRX : rotate right extended with C

Data operation varieties

- Logical shift:

  -fills with zeroes.

- Arithmetic shift:

  -fills with ones.

- RRX performs 33-bit rotate, including C bit from CPSR above sign bit.

ARM comparison instructions

- CMP : compare
- CMN : negated compare
- TST : bit-wise AND
- TEQ : bit-wise XOR
- These instructions set only the NZCV bits of CPSR.

ARM move instructions

- MOV, MVN : move (negated)

  MOV r0, r1 ; sets r0 to r1

NUMBER BASE CONVERSION

- LDR, LDRH, LDRB : load (half-word, byte)
- STR, STRH, STRB : store (half-word, byte)
- Addressing modes:

-register indirect : LDR r0,[r1]

-with second register : LDR r0,[r1,-r2]

-with constant : LDR r0,[r1,#4]

ARM ADR pseudo-op

- Cannot refer to an address directly in an instruction.
- Generate value by performing arithmetic on PC.
- ADR pseudo-op generates instruction required to calculate address:

  ADR r1,FOO

Example: C assignments

- C:

```
x = (a + b) - c;
```

- Assembler:

```
ADR r4,a          ; get address for a
LDR r0,[r4]       ; get value of a
ADR r4,b          ; get address for b, reusing r4
LDR r1,[r4]       ; get value of b
ADD r3,r0,r1      ; compute a+b
ADR r4,c          ; get address for c
LDR r2,[r4]       ; get value of c
```

C assignment, cont'd.

```
SUB r3,r3,r2      ; complete computation of x
ADR r4,x          ; get address for x
STR r3,[r4]       ; store value of x
```

Example: C assignment

- C:

```
y = a*(b+c);
```

- Assembler:

```
ADR r4,b ; get address for b
LDR r0,[r4] ; get value of b
ADR r4,c ; get address for c
LDR r1,[r4] ; get value of c
ADD r2,r0,r1 ; compute partial result
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
```

C assignment, cont'd.

```
MUL r2,r2,r0 ; compute final value for y
ADR r4,y     ; get address for y
STR r2,[r4]  ; store y
```

Example: C assignment

- C:

```
z = (a << 2) | (b & 15);
```

- Assembler:

```
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
MOV r0,r0,LSL 2 ; perform shift
ADR r4,b ; get address for b
LDR r1,[r4] ; get value of b
AND r1,r1,#15 ; perform AND
ORR r1,r0,r1 ; perform OR
```

C assignment, cont'd.

```
ADR r4,z    ; get address for z
STR r1,[r4] ; store value for z
```

Additional addressing modes

- Base-plus-offset addressing:
  LDR r0,[r1,#16]
  Loads from location r1+16
- Auto-indexing increments base register:
  LDR r0,[r1,#16]!
- Post-indexing fetches, then does offset:
  LDR r0,[r1],#16
  Loads r0 from r1, then adds 16 to r1.

ARM flow of control

- All operations can be performed conditionally, testing CPSR:
  EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE
- Branch operation:
  B #100
  Can be performed conditionally

Example: if statement

- C:

```
if (a > b) { x = 5; y = c + d; } else x = c - d;
```

- Assembler:

```
; compute and test condition
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
ADR r4,b ; get address for b
LDR r1,[r4] ; get value for b
CMP r0,r1 ; compare a < b
BLE fblock ; if a >< = b, branch to false block
```

If statement, cont'd.

```
; true block
  MOV r0,#5 ; generate value for x
  ADR r4,x ; get address for x
  STR r0,[r4] ; store x
  ADR r4,c ; get address for c
  LDR r0,[r4] ; get value of c
  ADR r4,d ; get address for d
  LDR r1,[r4] ; get value of d
  ADD r0,r0,r1 ; compute y
  ADR r4,y ; get address for y
  STR r0,[r4] ; store y
  B after ; branch around false block
```

If statement, cont'd.

```
; false block
 fblock ADR r4,c ; get address for c
    LDR r0,[r4] ; get value of c
    ADR r4,d ; get addressfor d
    LDR r1,[r4] ; get value ford
    SUB r0,r0,r1 ; computea-b
    ADR r4,x ; get address for x
  STR r0,[r4] ; store value of x
  after ...
```

Example: switch statement

- **C:**

  switch (test) { case 0: … break; case 1: … }

- **Assembler:**

```
 ADR r2,test ; get address for test
 LDR r0,[r2] ; load value for test
 ADR r1,switchtab ; load address for switch table
LDR r1,[r1,r0,LSL #2] ; index switch table
switchtab DCD case0
 DCD case1
…
```

Example: FIR filter

- **C**:

```
for (i=0, f=0; i<N; i++)
f = f + c[i]*x[i];
```

- **Assembler**

```
; loop initiation code
MOV r0,#0 ; use r0 for I
MOV r8,#0 ; use separate index for arrays
ADR r2,N ; get address for N
LDR r1,[r2] ; get value of N
MOV r2,#0 ; use r2 for f
```

FIR filter, cont'.d

```
        ADR r3,c ; load r3 with base of c
        ADR r5,x ; load r5 with base of x
; loop body
loop LDR r4,[r3,r8] ; get c[i]
        LDR r6,[r5,r8] ; get x[i]
        MUL r4,r4,r6 ; compute c[i]*x[i]
        ADD r2,r2,r4 ; add into running sum
        ADD r8,r8,#4 ; add one word offset to array index
        ADD r0,r0,#1 ; add 1 to i
        CMP r0,r1 ; exit?
        BLT loop ; if i < N, continue
```

ARM subroutine linkage

- Branch and link instruction:

BL foo

Copies current PC to r14.

To return from subroutine:

MOV r15,r14

Nested subroutine calls

- **Nesting/recursion requires coding convention:**

```
f1 LDR r0,[r13] ; load arg into r0 from stack
; call f2()
STR r14,*r13+! ; store f1's return adrs
STR r0,[r13]! ; store arg to f2 on stack
BL f2 ; branch and link to f2
; return from f1()
SUB r13,#4 ; pop f2's arg off stack
LDR r13!,r15 ; restore register and return
```

SHARC instruction set

- SHARC programming model.
- SHARC assembly language.
- SHARC memory organization.
- SHARC data operations.
- SHARC flow of control

SHARC programming model

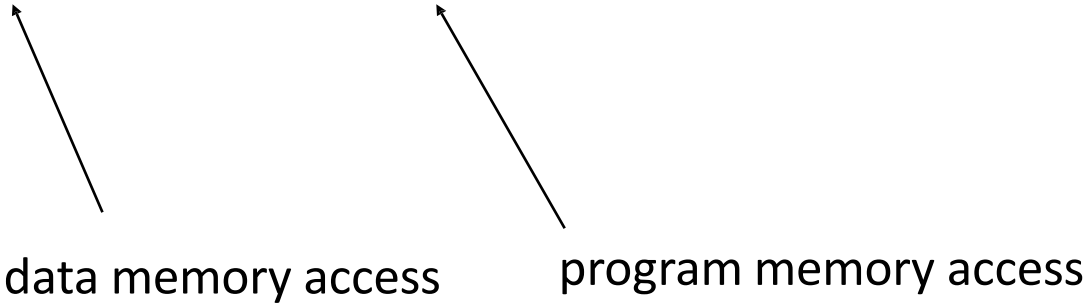
- **Register files:**

  R0-R15 (aliased as F0-F15 for floating point)
- **Status registers.**
- **Loop registers.**
- **Data address generator registers.**
- **Interrupt registers.**

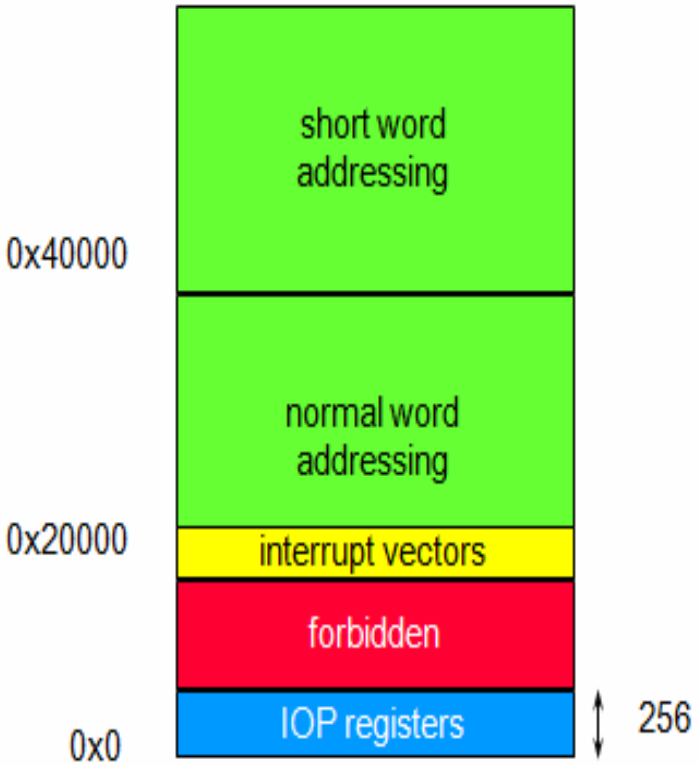
SHARC assembly language

Algebraic notation terminated by semicolon:

```
R1=DM(M0,I0), R2=PM(M8,I8); ! comment
label: R3=R1+R2;
```



SHARC MEMORY SPACE



SHARC DATA TYPES

- 32-bit IEEE single-precision floating-point.
- 40-bit IEEE extended-precision floating-point.
- 32-bit integers.
- Memory organized internally as 32-bit words.

SHARC MICRO ARCHITECTURE

- Modified Harvard architecture.
- Program memory can be used to store some data.
- Register file connects to:
- multiplier
- shifter;
- ALU.

SHARC MODE REGISTERS

- Most important:
- ASTAT: arithmetic status.
- STKY: sticky.
- MODE 1: mode 1.

ROUNDING AND SATURATION

- Floating-point can be:
  - rounded toward zero;
  - rounded toward nearest.
- ALU supports saturation arithmetic (ALUSAT bit in MODE1).
  - Overflow results in max value, not rollover.

MULTIPLIER

- Fixed-point operations can accumulate into local MR registers or be written to register file. Fixed-point result is 80 bits.
- Floating-point results always go to register file.
- Status bits: negative, under/overflow, invalid, fixed-point underflow, floating-point underflow, floating-point invalid.

ALU/SHIFTER STATUS FLAGS

ALU:

- zero, overflow, negative, fixed-point carry, input sign, floating-point invalid, last op was floating-point, compare accumulation registers, floating-point under/oveflow, fixed-point overflow, floating-point invalid

Shifter:

- zero, overflow, sign

FLAG OPERATIONS

- All ALU operations set AZ (zero), AN (negative), AV (overflow), AC (fixed-point carry), AI (floating-point invalid) bits in ASTAT.
- STKY is sticky version of some ASTAT bits.

SHARC load/store

- Load/store architecture: no memory-direct operations.
- Two data address generators (DAGs):
  - program memory;
  - data memory.
- Must set up DAG registers to control loads/stores.

SHARC program sequencer

Features:

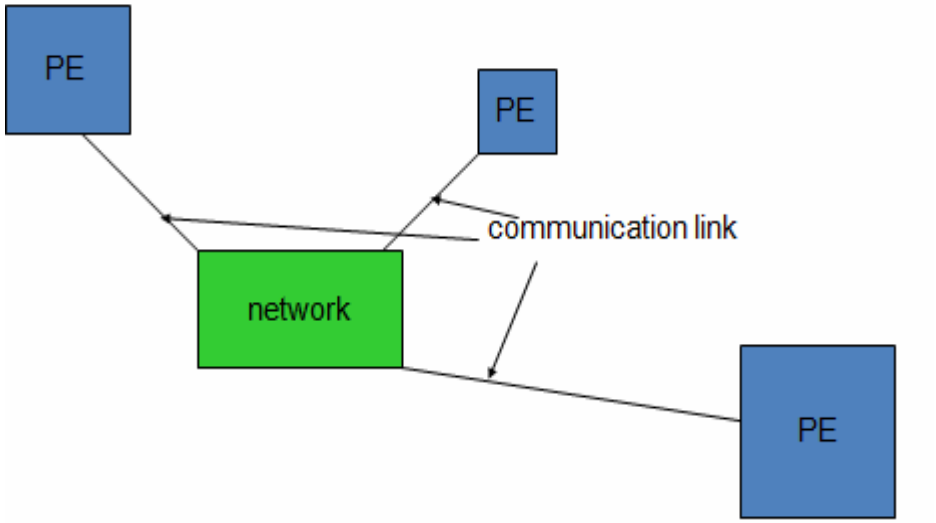
- instruction cache;
- PC stack;
- status registers;
- loop logic;
- data address generator;

Networking for Embedded Systems

- Why we use networks.
- Network abstractions.
- Example networks.

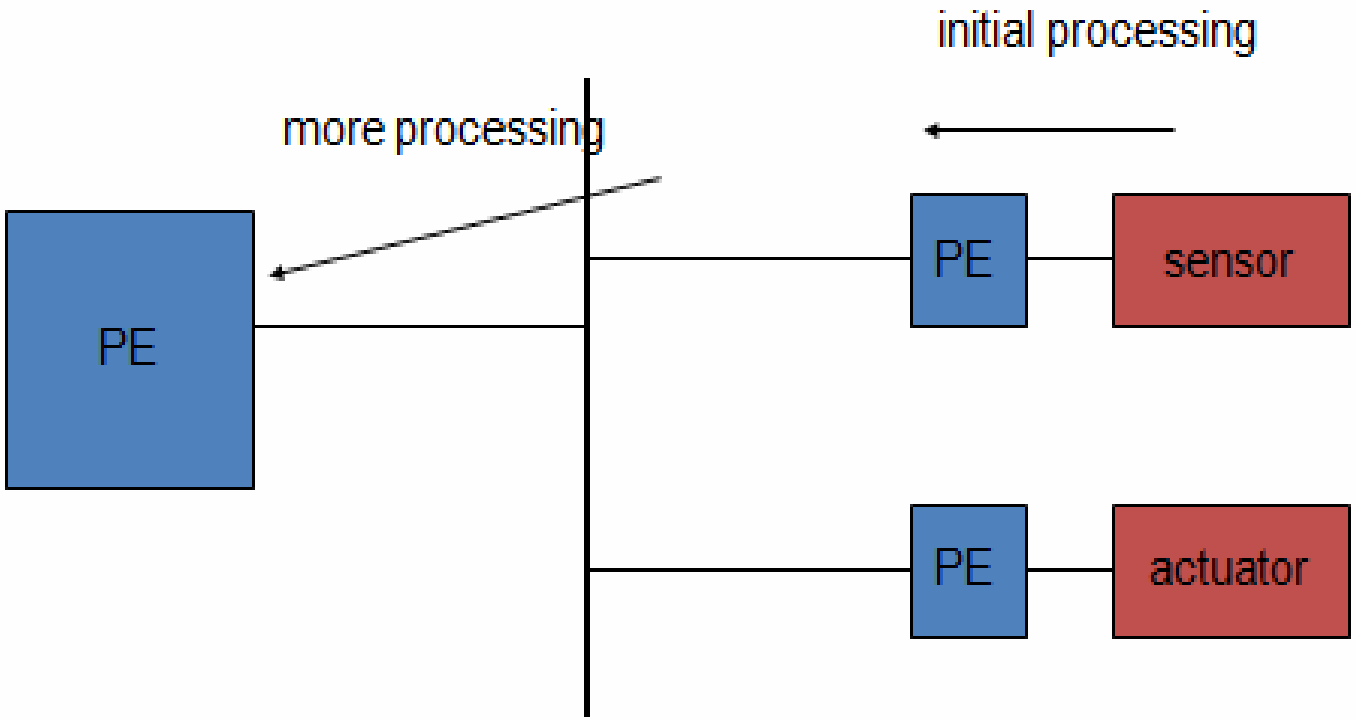
Network elements

Distributed computing platform:



PEs may be CPUs or ASICs.

Networks in embedded systems



Why distributed?

- Higher performance at lower cost.
- Physically distributed activities---time constants may not allow transmission to central site.
- Improved debugging---use one CPU in network to debug others.
- May buy subsystems that have embedded processors.

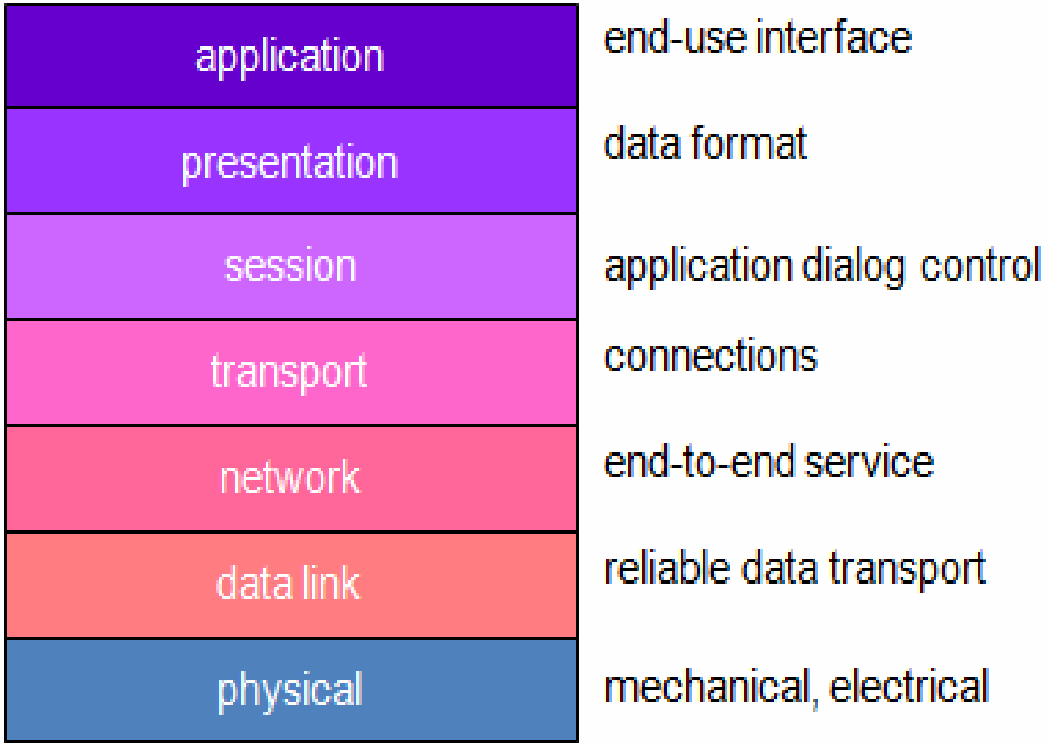
Network abstractions

- International Standards Organization (ISO) developed the Open Systems Interconnection (OSI) model to describe networks:

  7-layer model.

- Provides a standard way to classify network components and operations.

OSI model

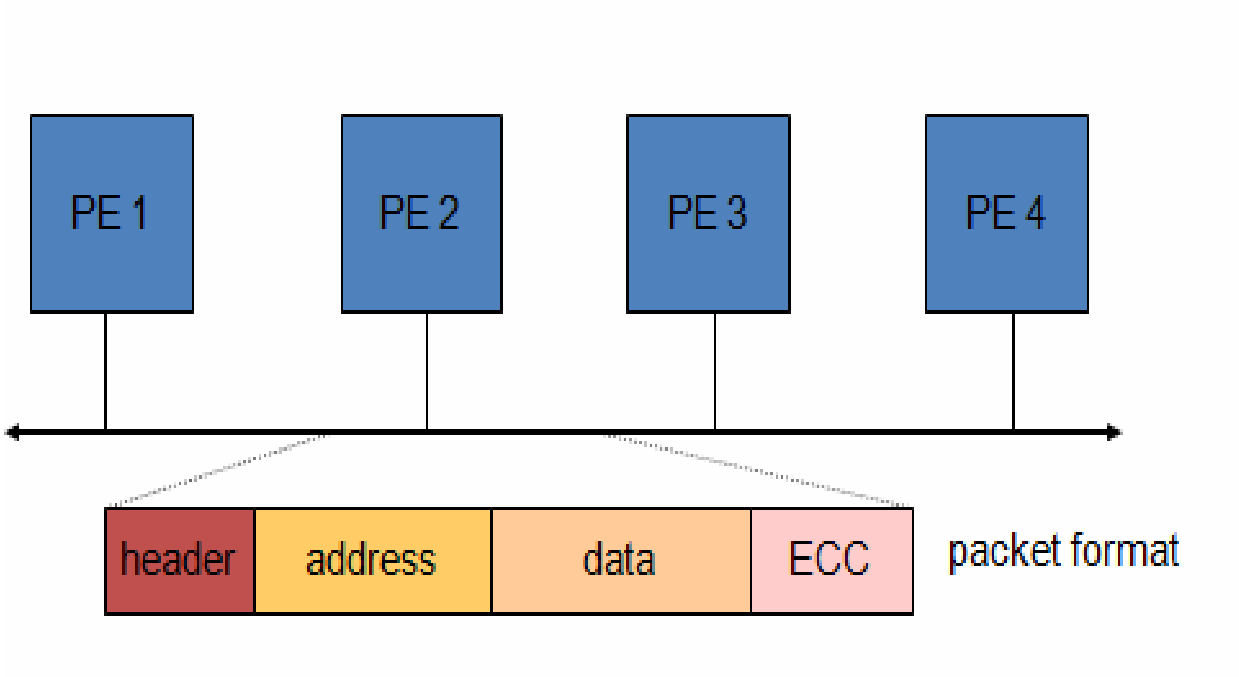

application
presentation
session
transport
network
data link
physical
end-use interface
data format
application dialog control
connections
end-to-end service
reliable data transport
mechanical, electrical

OSI layers

- Physical: connectors, bit formats, etc.
- Data link: error detection and control across a single link (single hop).
- Network: end-to-end multi-hop data communication.
- Transport: provides connections; may optimize network resources.
- Session: services for end-user applications: data grouping, check pointing, etc.
- Presentation: data formats, transformation services.
- Application: interface between network and end-user programs.

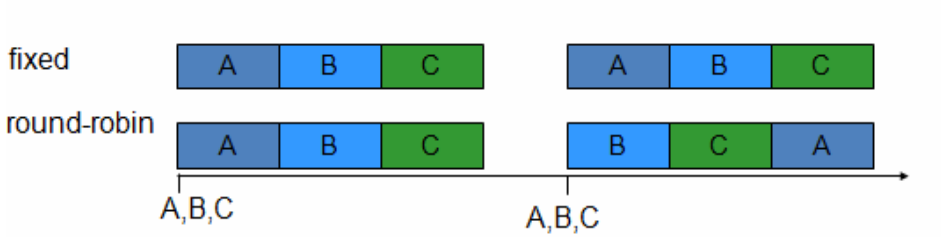
Bus networks

- Common physical connection:

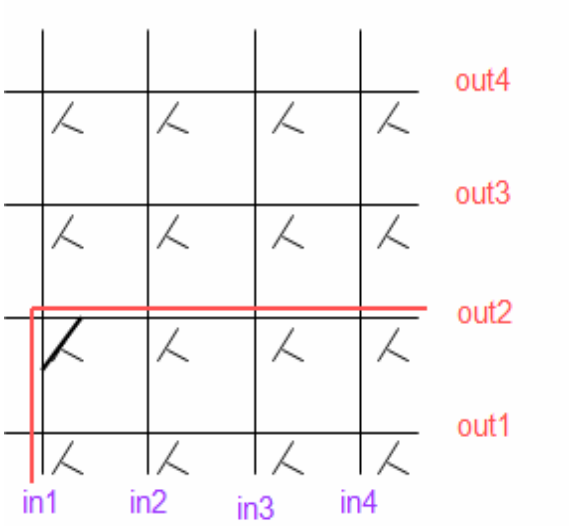


Bus arbitration

- Fixed: Same order of resolution every time.
- Fair: every PE has same access over long periods.
- Round-robin: rotate top priority among Pes.



Crossbar



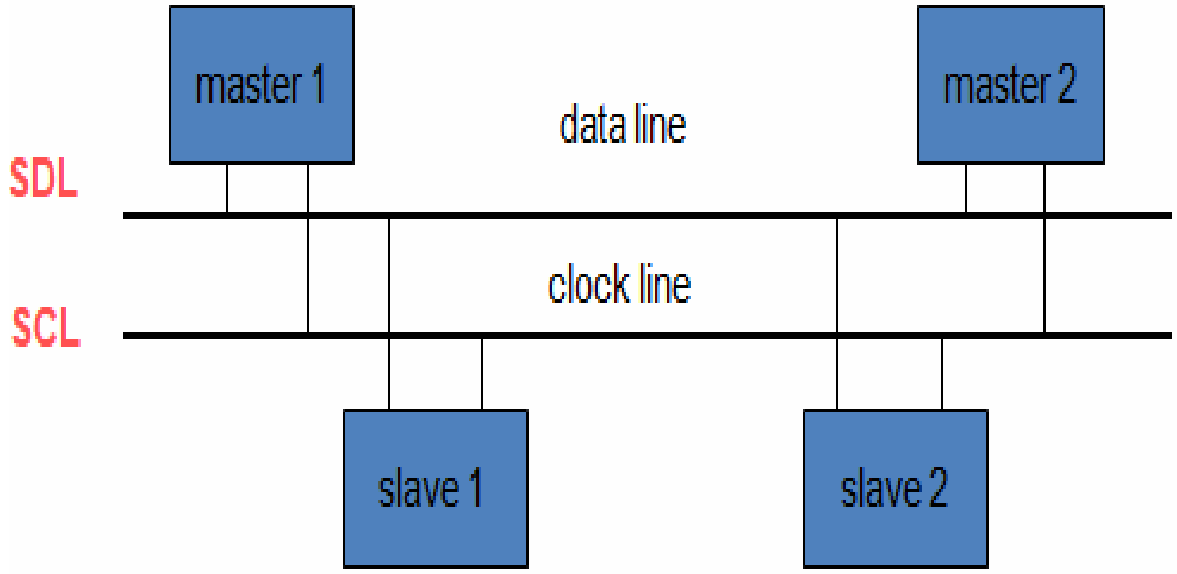
Crossbar characteristics:

- Non-blocking.
- Can handle arbitrary multi-cast combinations.
- Size proportional to n^2.

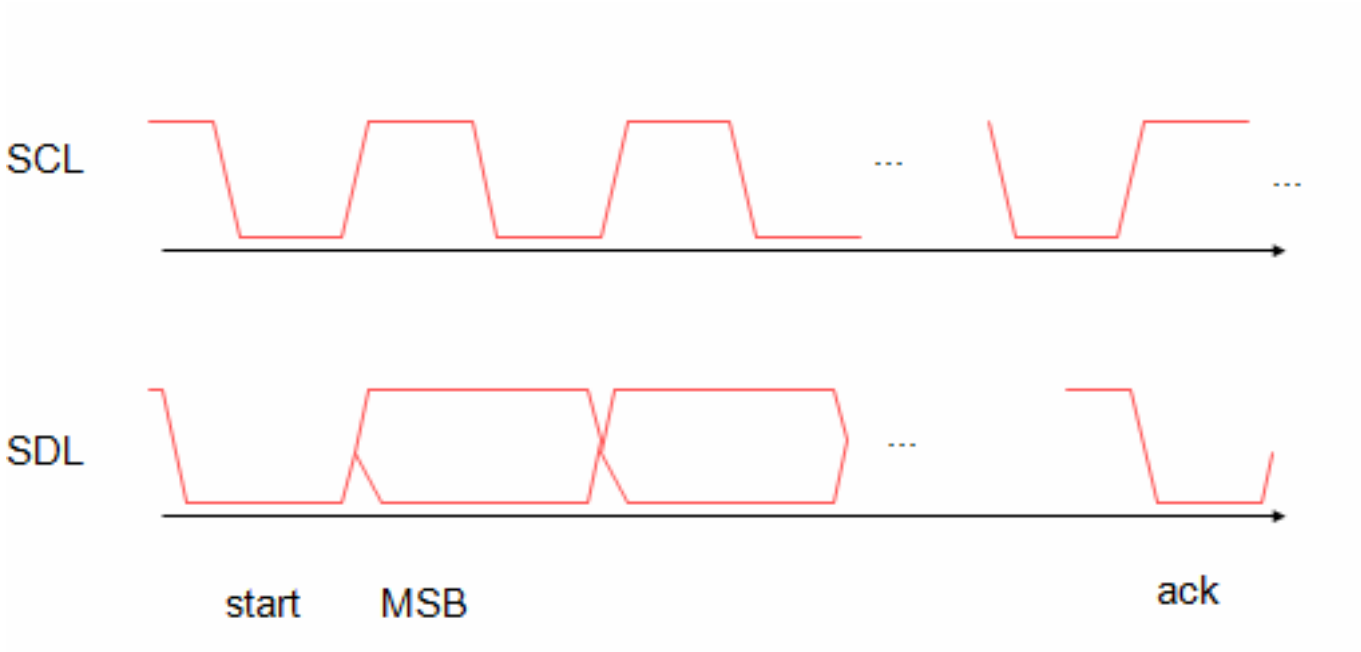
I2C bus

- Designed for low-cost, medium data rate applications.
- Characteristics:
  - serial;
  - multiple-master;
  - fixed-priority arbitration.
- Several microcontrollers come with built-in I^2C controllers.

I2C physical layer



I2C data format



I2C signaling

- Sender pulls down bus for 0.
- Sender listens to bus---if it tried to send a 1 and heard a 0, someone else is simultaneously transmitting.
- Transmissions occur in 8-bit bytes.

I^2C data link layer

- Every device has an address (7 bits in standard, 10 bits in extension).Bit 8 of address signals read or write.
- General call address allows broadcast.

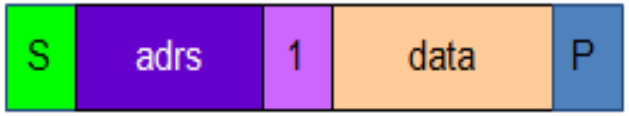
I2C bus arbitration

- Sender listens while sending address.
- When sender hears a conflict, if its address is higher, it stops signaling.
- Low-priority senders relinquish control early enough in clock cycle to allow bit to be transmitted reliably.

I2C transmissions

multi-byte write

read from slave



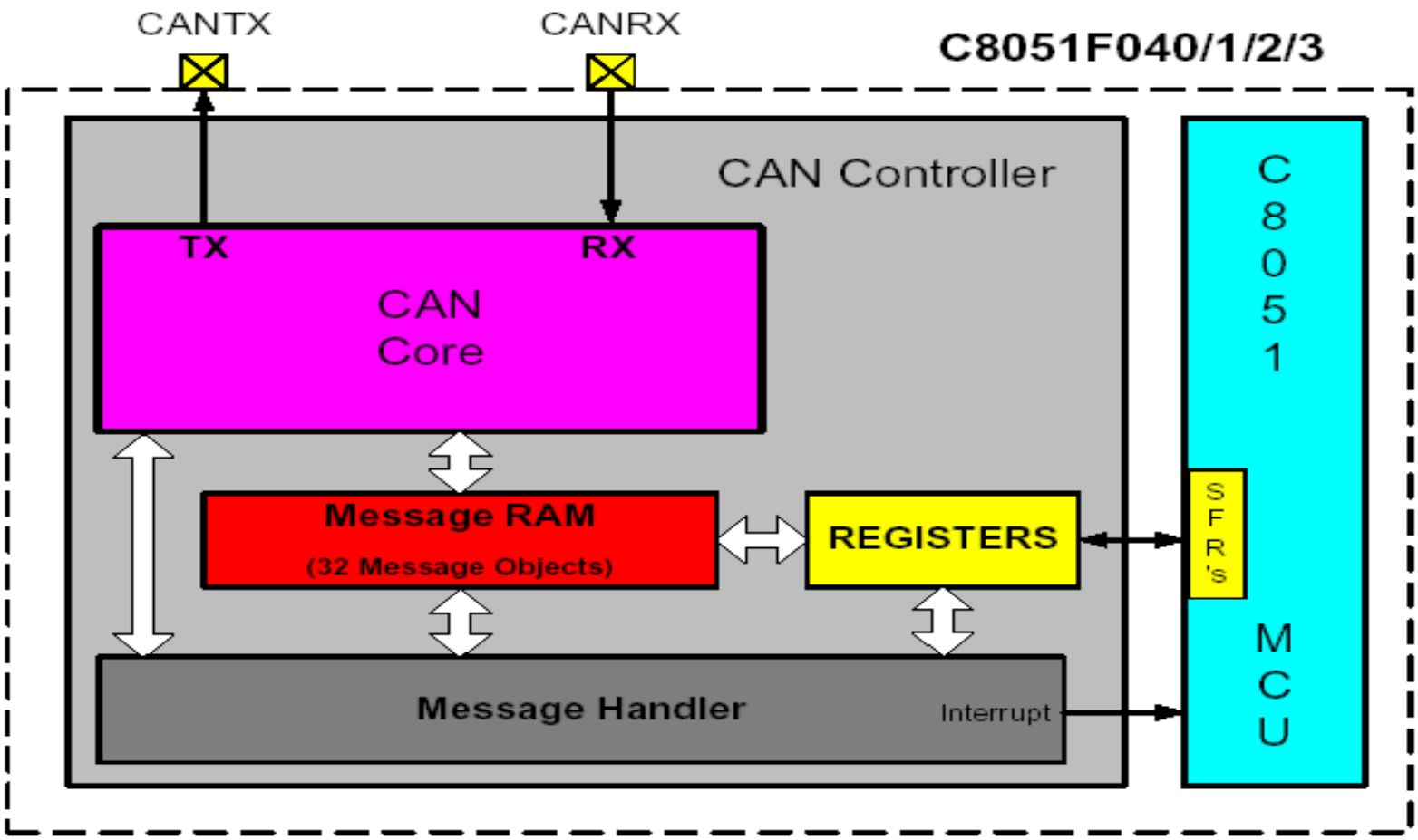
write, then read

CAN BUS

- CAN (Controller Area Network) is a serial bus system used to communicate between several embedded 8-bit and 16-bit microcontrollers.

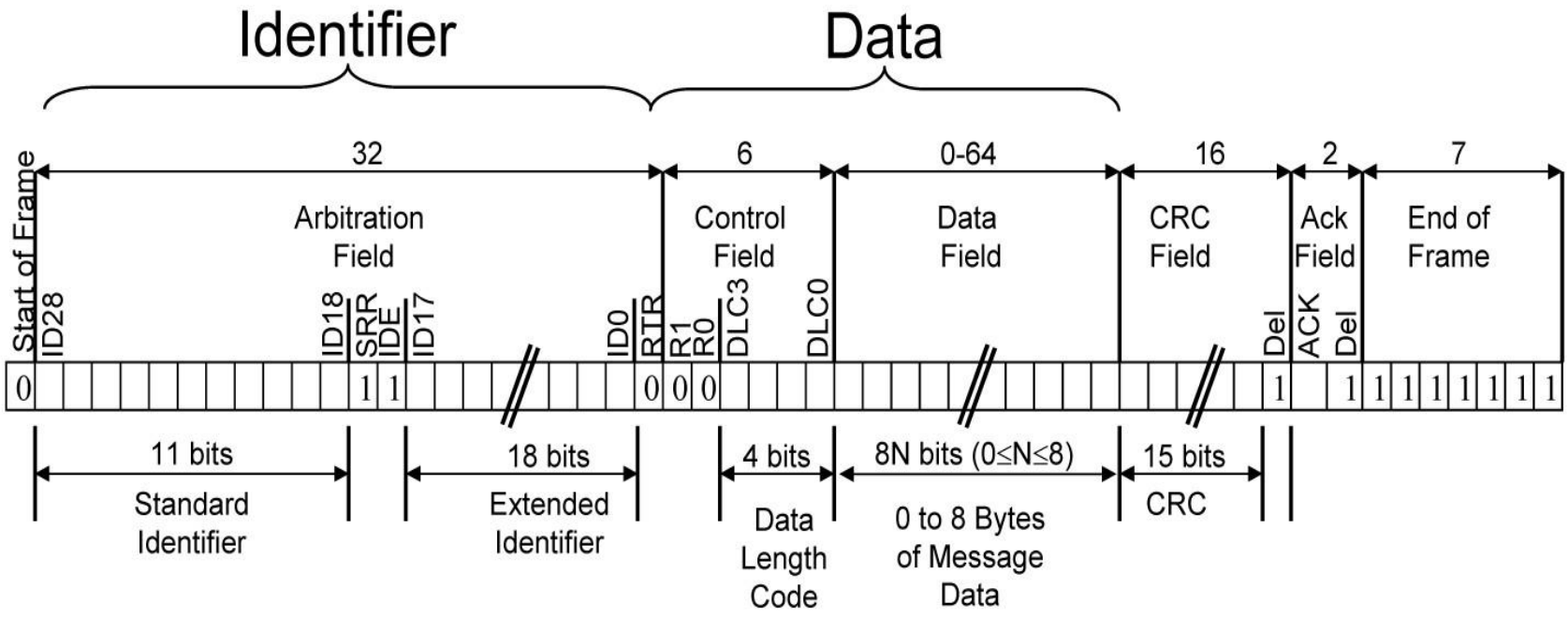
- It was originally designed for use in the automotive industry but is used today in many other systems (e.g. home appliances and industrial machines).

CAN Controller Diagram



Data Format

- Each message has an ID, Data and overhead.
- Data –8 bytes max
- Overhead – start, end, CRC, ACK



Internet –EnAnalyzed systems

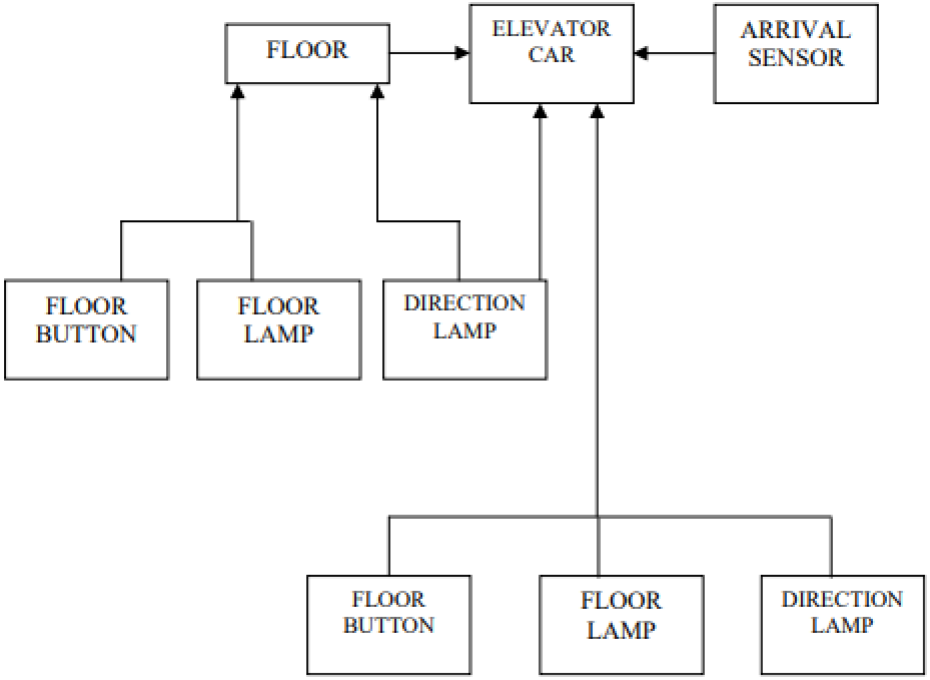
- Embedded systems are internet enabled by using TCP/IP protocols for networking to internet and assigning IP addresses to each systems.
- Internet provides a standard way for embedded systems to act in concert with other devices and with users.eg.

1. High end laser printers use internet protocols to receive print jobs from host machines.
2. PDA can display web pages ,read email and synchronous calendar information with remote computer.

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.

ELEVATOR CONTROLLER



ELEVATOR SYSTEM OVERVIEW