EMBEDDED SYSTEM DESIGN (AEC016)

B.Tech -ECE-VII Sem

IARE-R16

Institute of Aeronautical Engineering

UNIT-I

EMBEDDED COMPUTING

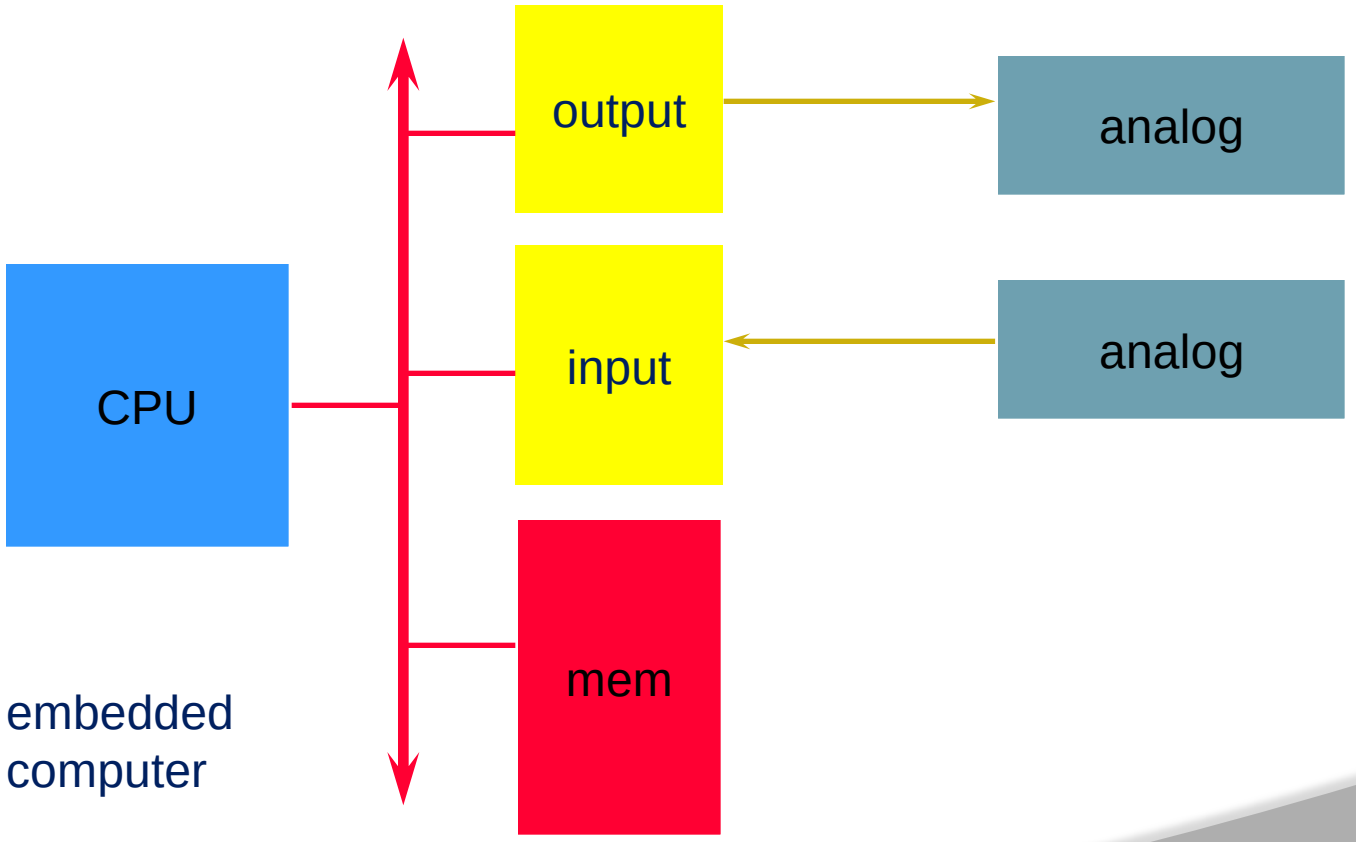
Definition

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

OR

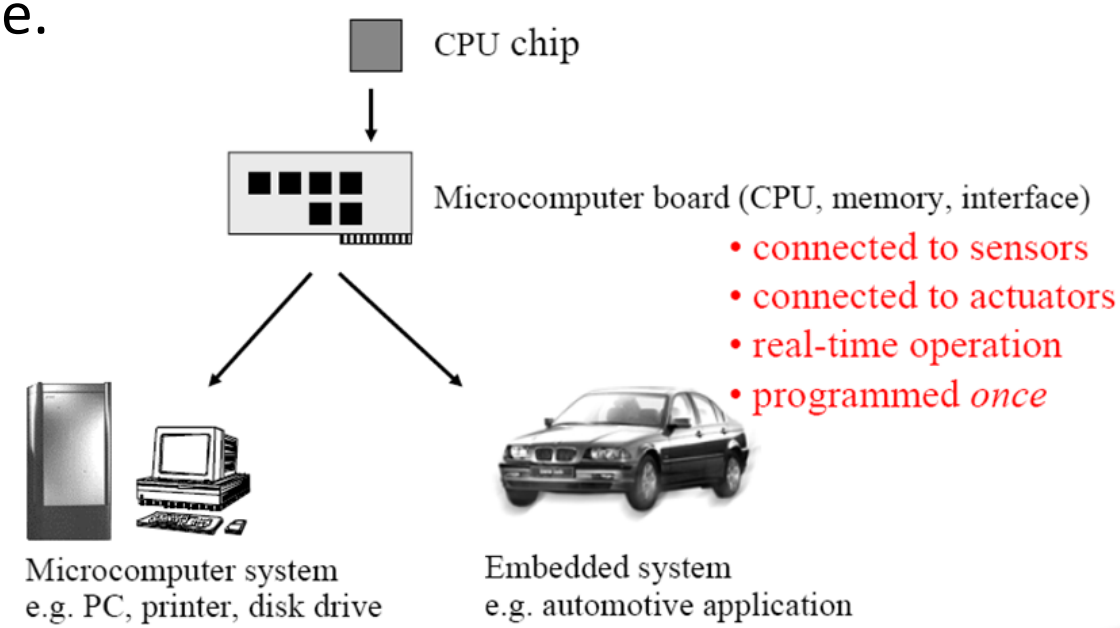
- A combination of hardware and software which together form a component of a larger machine.

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 (Cont)

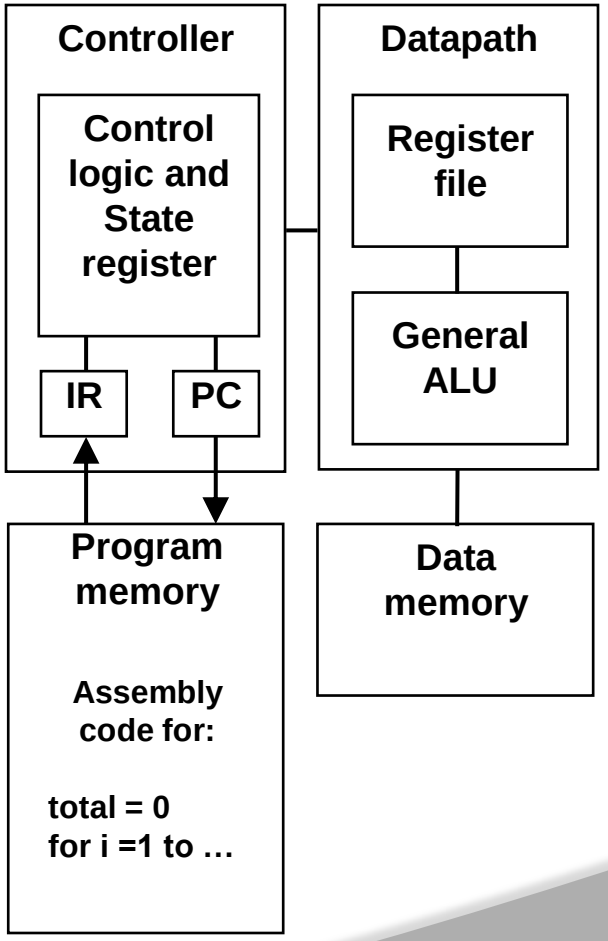
- Even though embedded systems cover a wide range of special-purpose systems, there are common characteristics
  - Low cost
    - Should be cheap to be competitive
      - Memory is typically very small compared to a general purpose computer system
      - Lightweight processors are used in embedded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

Embedded Systems (Cont)

- 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

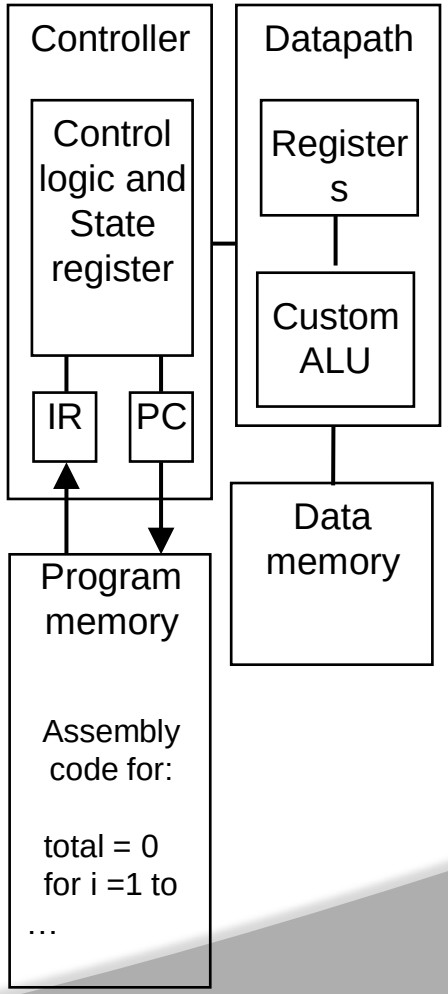
General-purpose processors

- 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

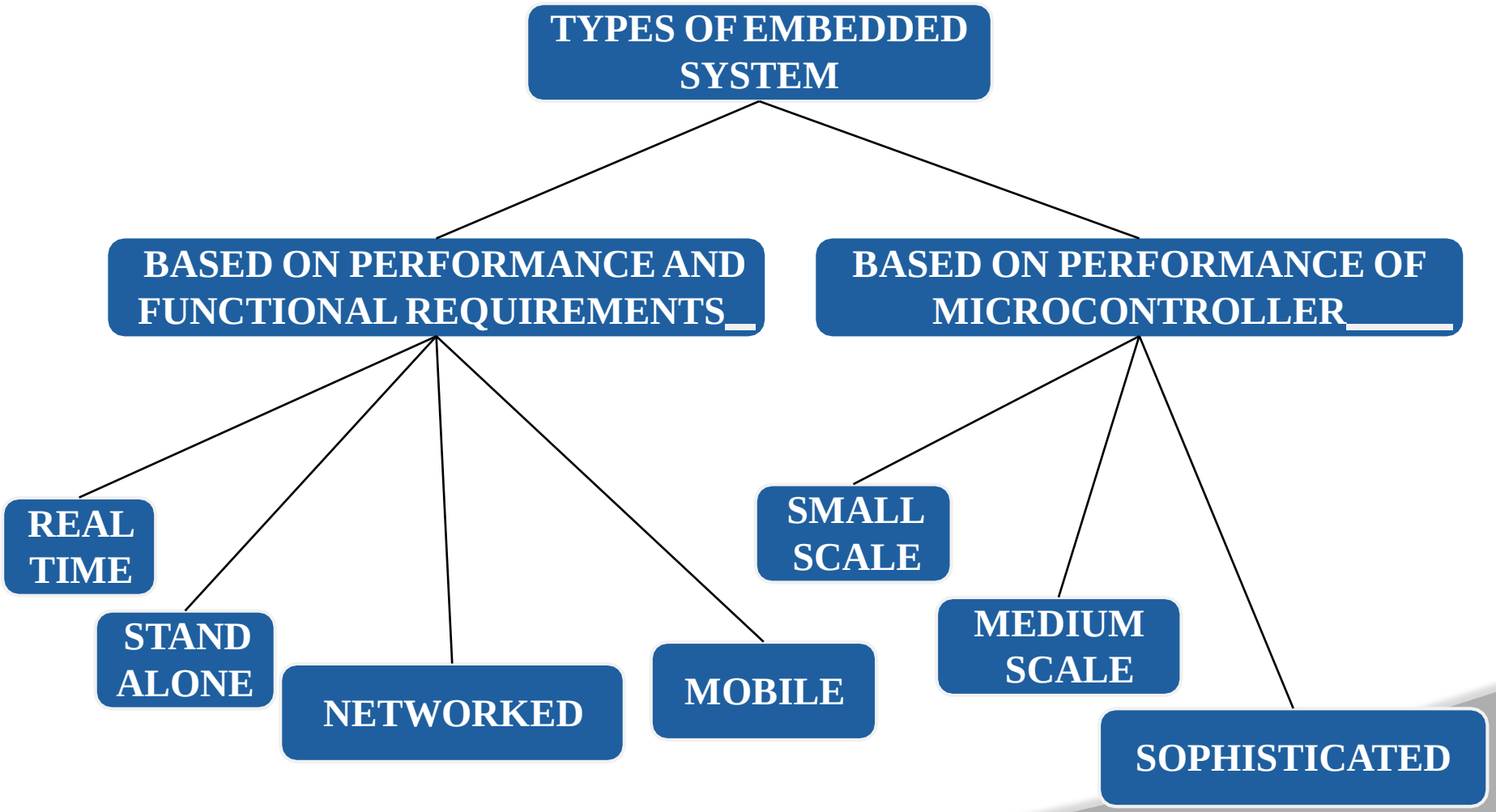
- Programmable processor optimized for a particular class of applications having common characteristics. Compromise between general-purpose and single-purpose processors
- Features
  - Program memory
  - Optimized datapath
  - Special functional units
- Benefits
  - Some flexibility, good performance,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

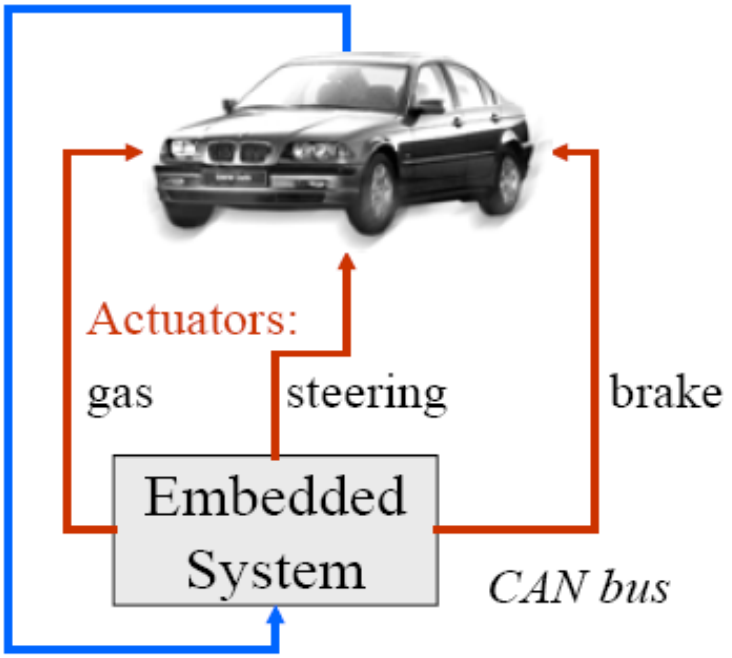
6. **Computer peripherals**
   - Printers, scanners, fax machines, etc…
7. **Computer Networking systems**
   - Network routers, switches, hubs, firewalls, etc…
8. **Health care**
   - CT scanner, ECG , EEG , EMG ,MRI, Glucose monitor, blood pressure monitor, medical diagnostic device, etc.
9. **Measurement & 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:

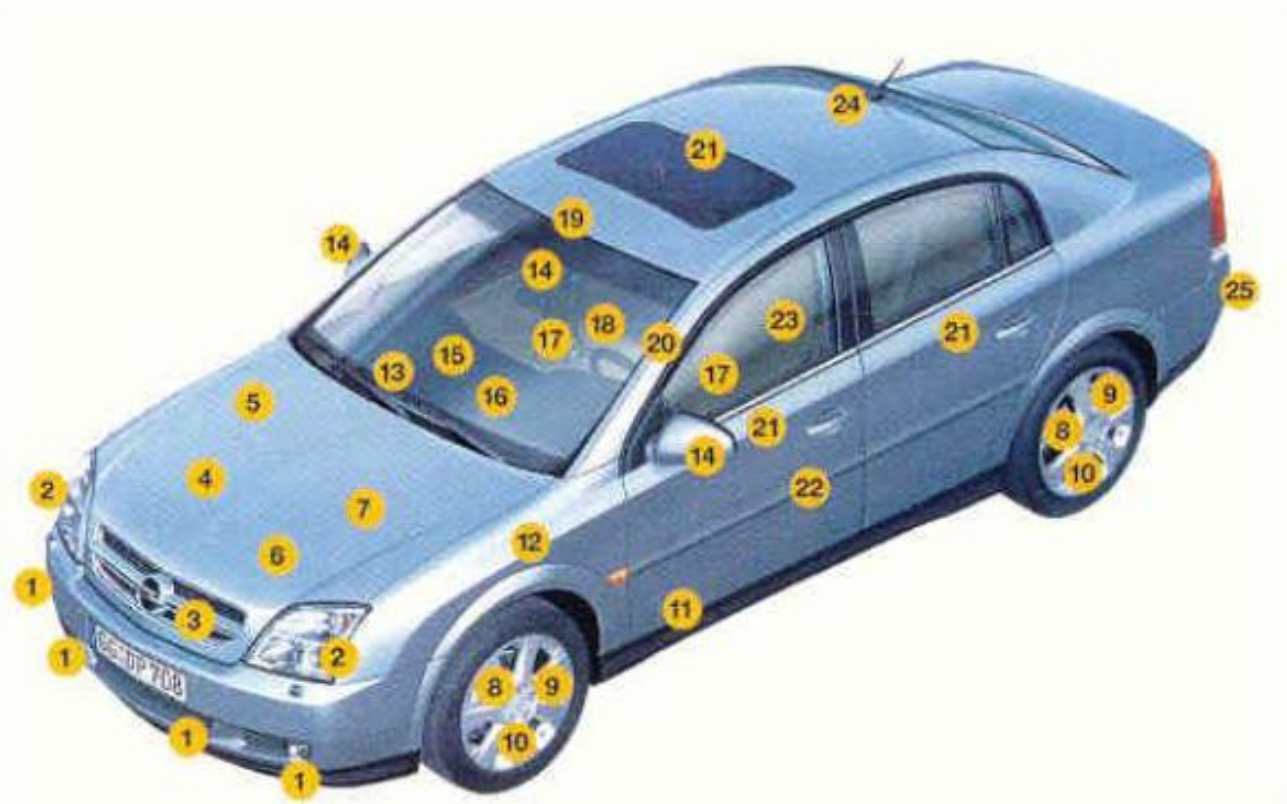
- Electronic gas
- Electronic brake
- Electronic steering

See: *The Daimler Story*



Application Areas

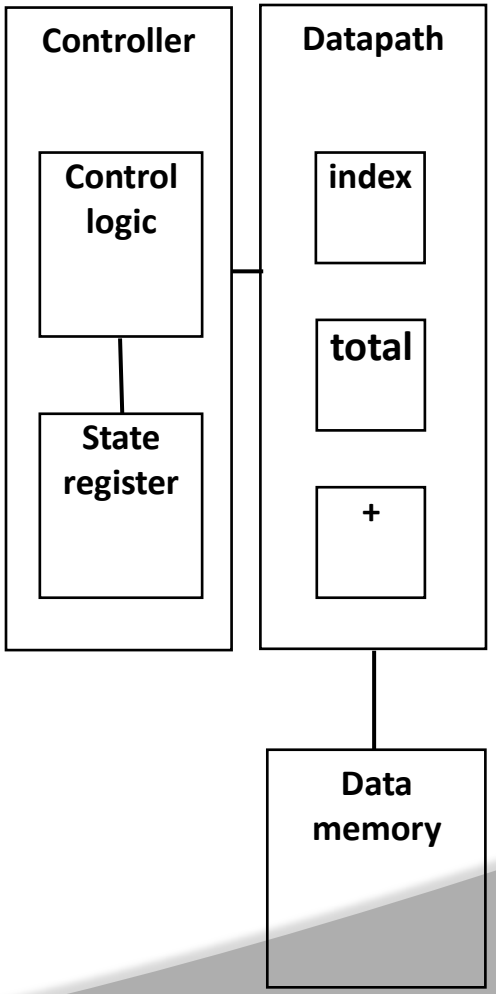
Automobiles



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

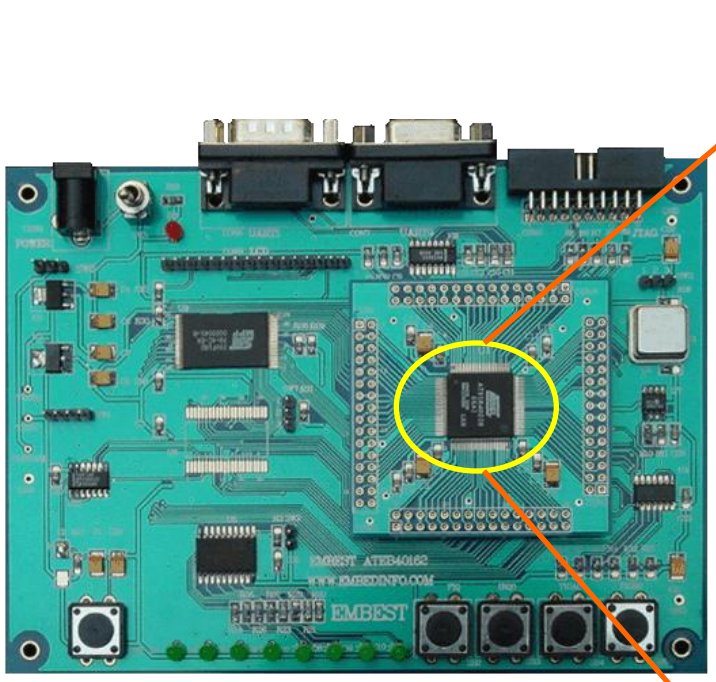
Microsoft Windows Embedded

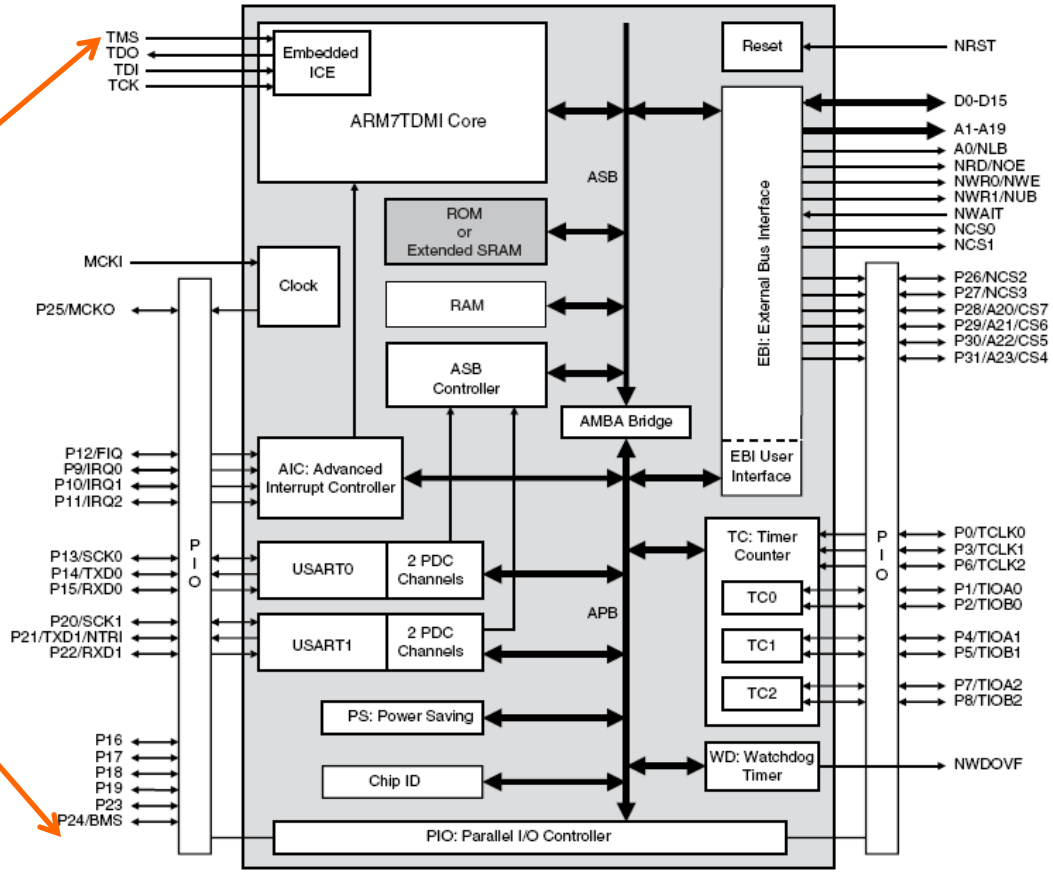
OS / Device Drivers

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)

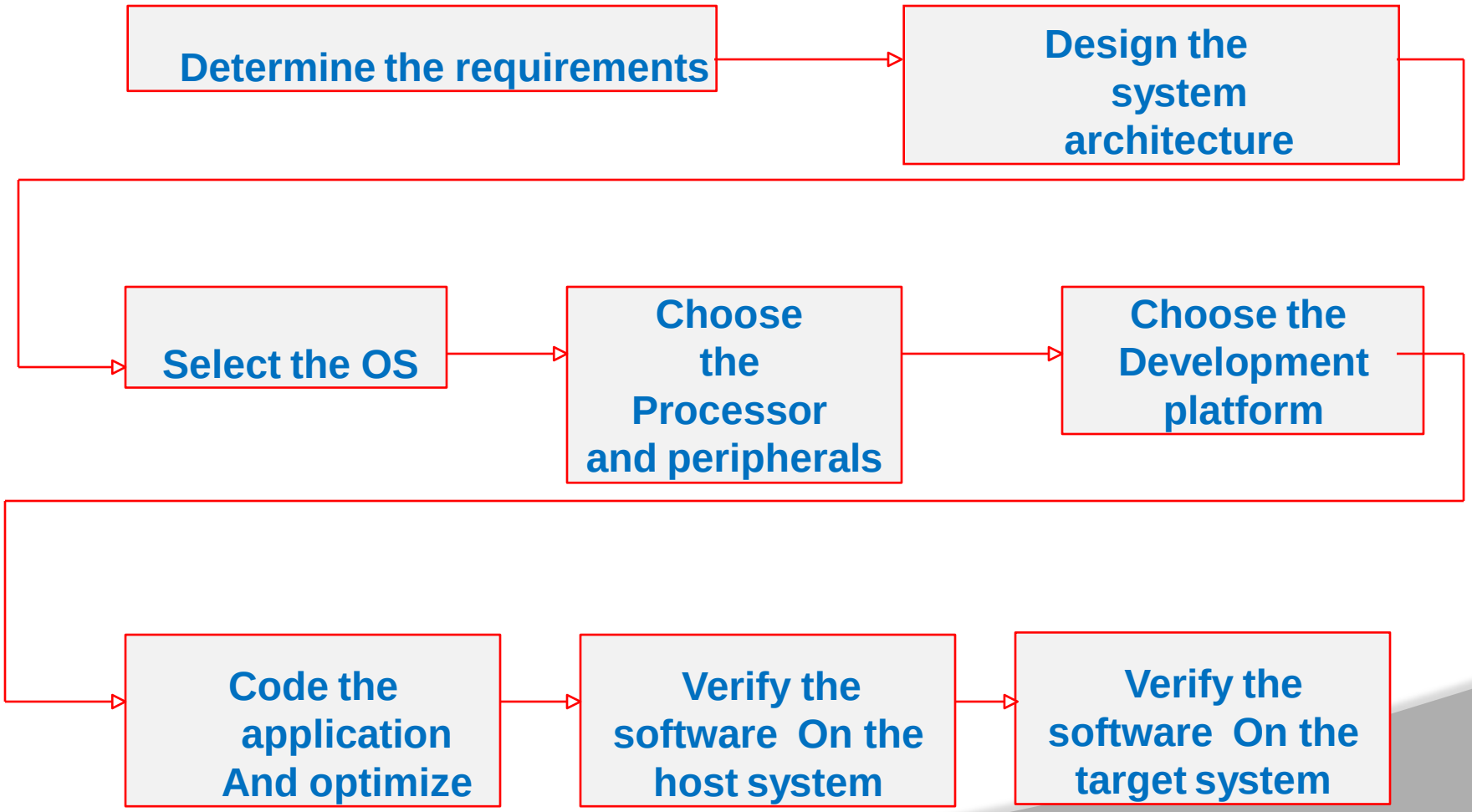
Components of Embedded Systems (Cont)

- 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,

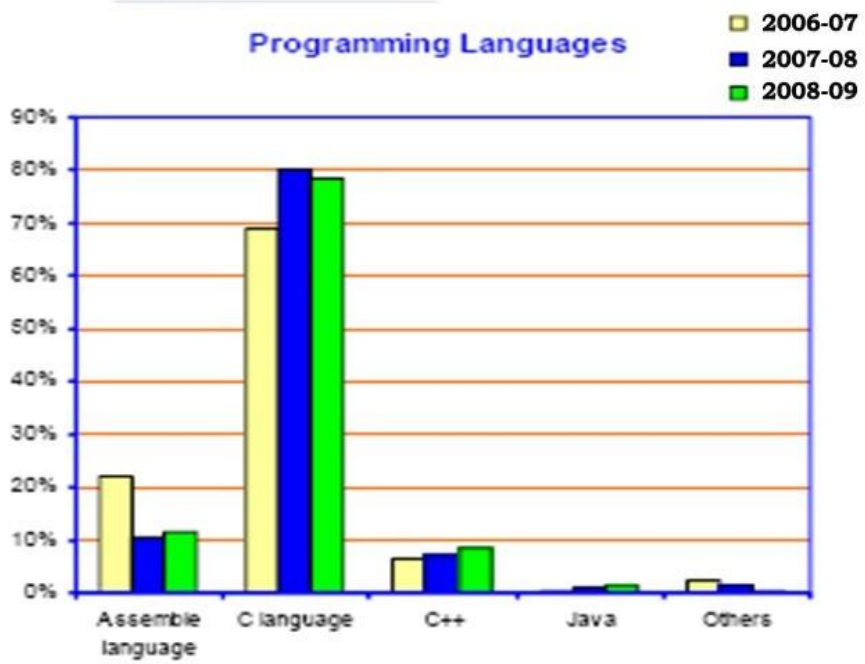
- Microprocessor8085,8086,Pentium
- Microcontroller
- MCS-51,PIC,AVR,MSP430
- Digital Signal Processor
- dsPIC,Blackfin,Sharc,TigerSharc

DEVELOPMENT PLATFORM

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

CODING THE APPLICATION

- **Choice of language.**



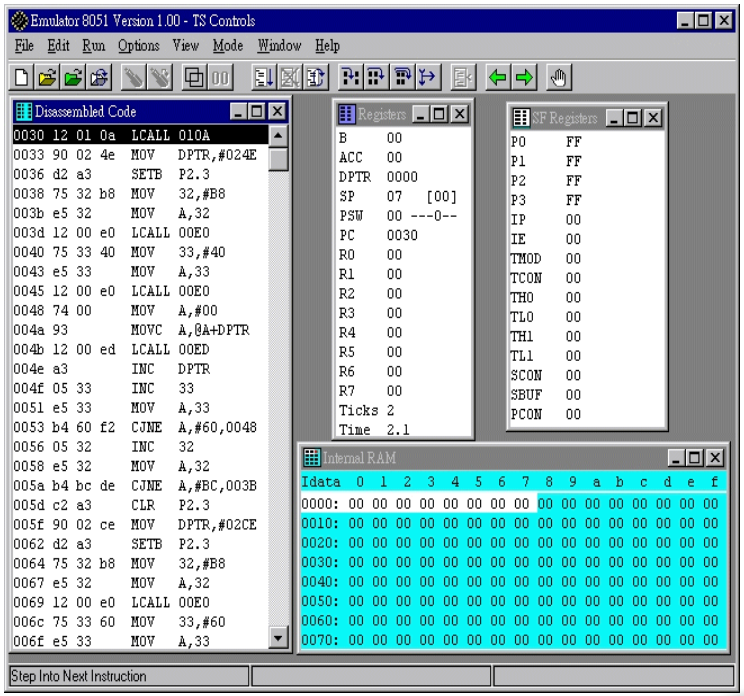
- **Assembly.**
- **C language.**
- **Object Oriented Language (C++, Java etc.).**

Optimizing the code

VERIFYING THE SOFTWARE ON THE HOST SYSTEM

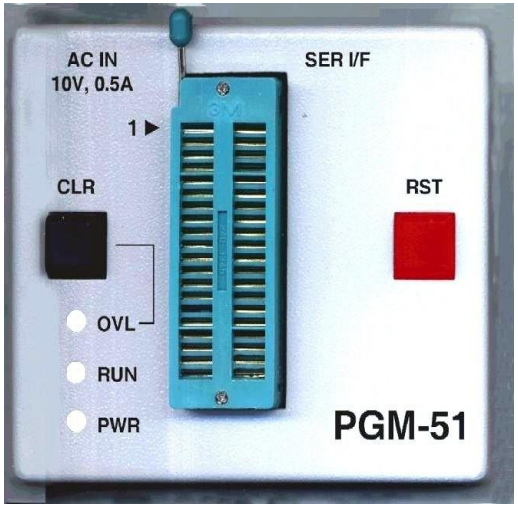
Compile and assemble the source code into object file.

Use a simulator to simulate the working o the system.



VERIFY 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

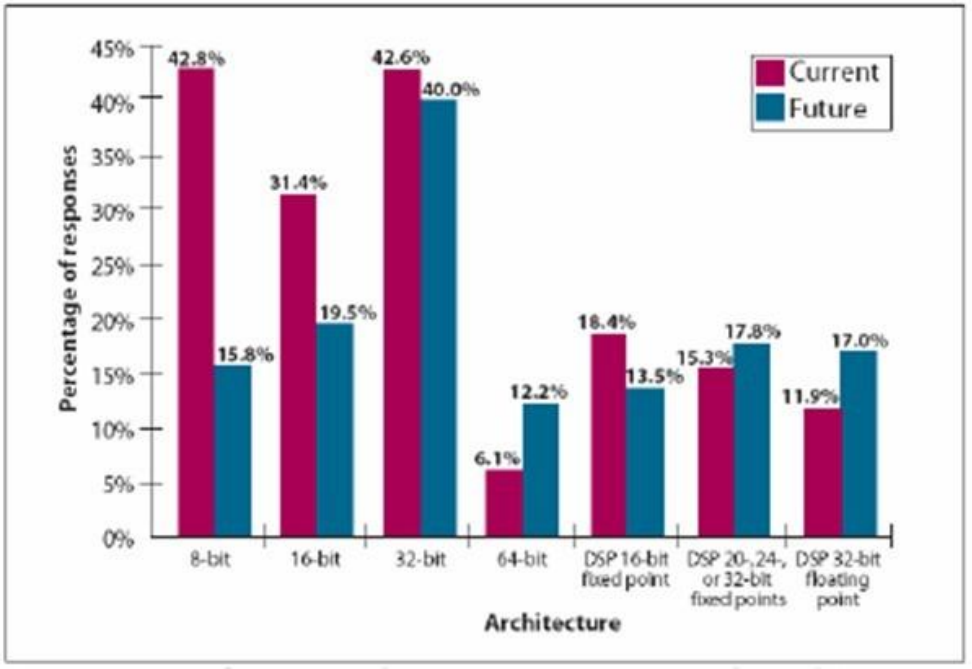
- **Due to the developments in Micro electronics availabilit of processors increased.**

- Reduces cost.

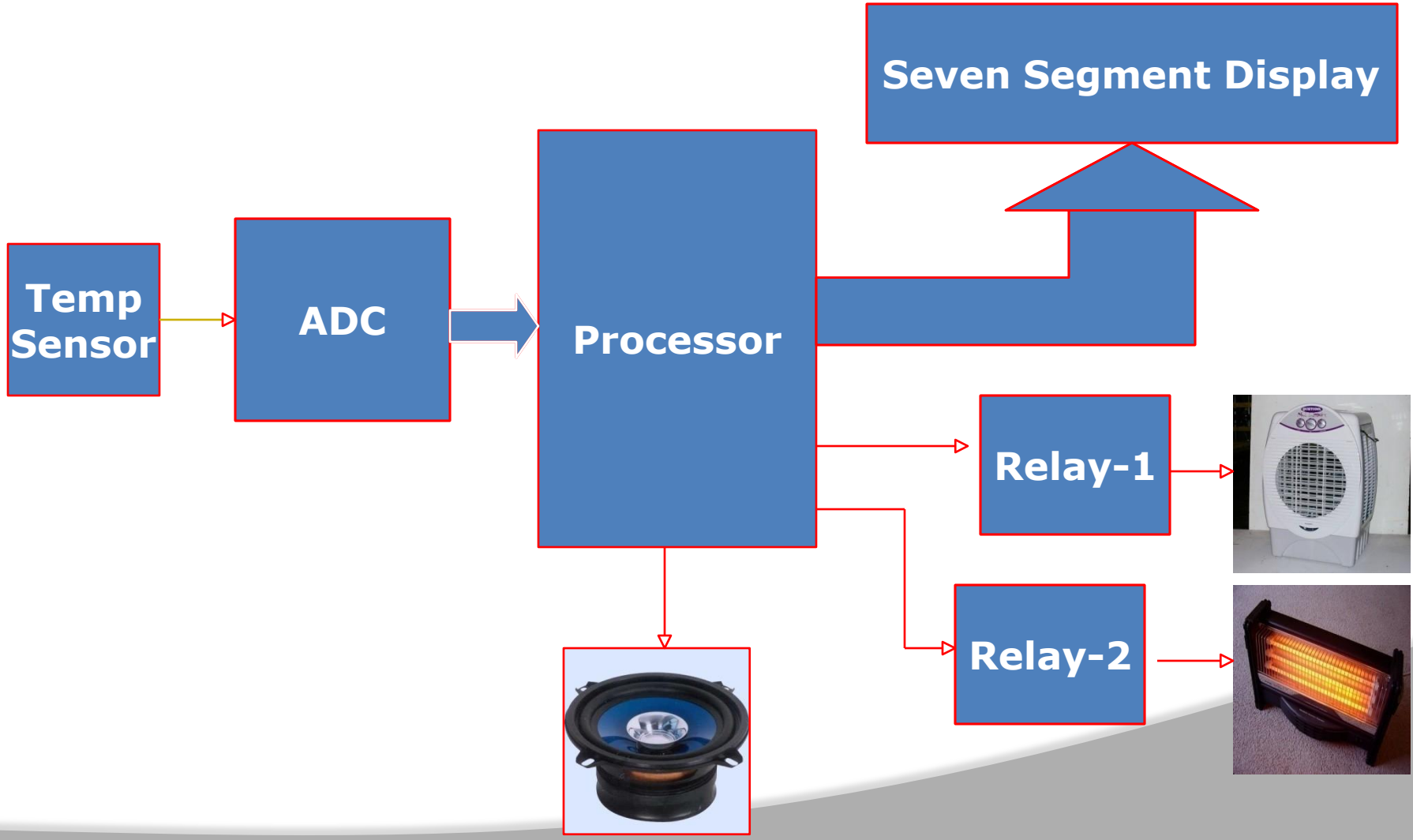
- Increased speed.

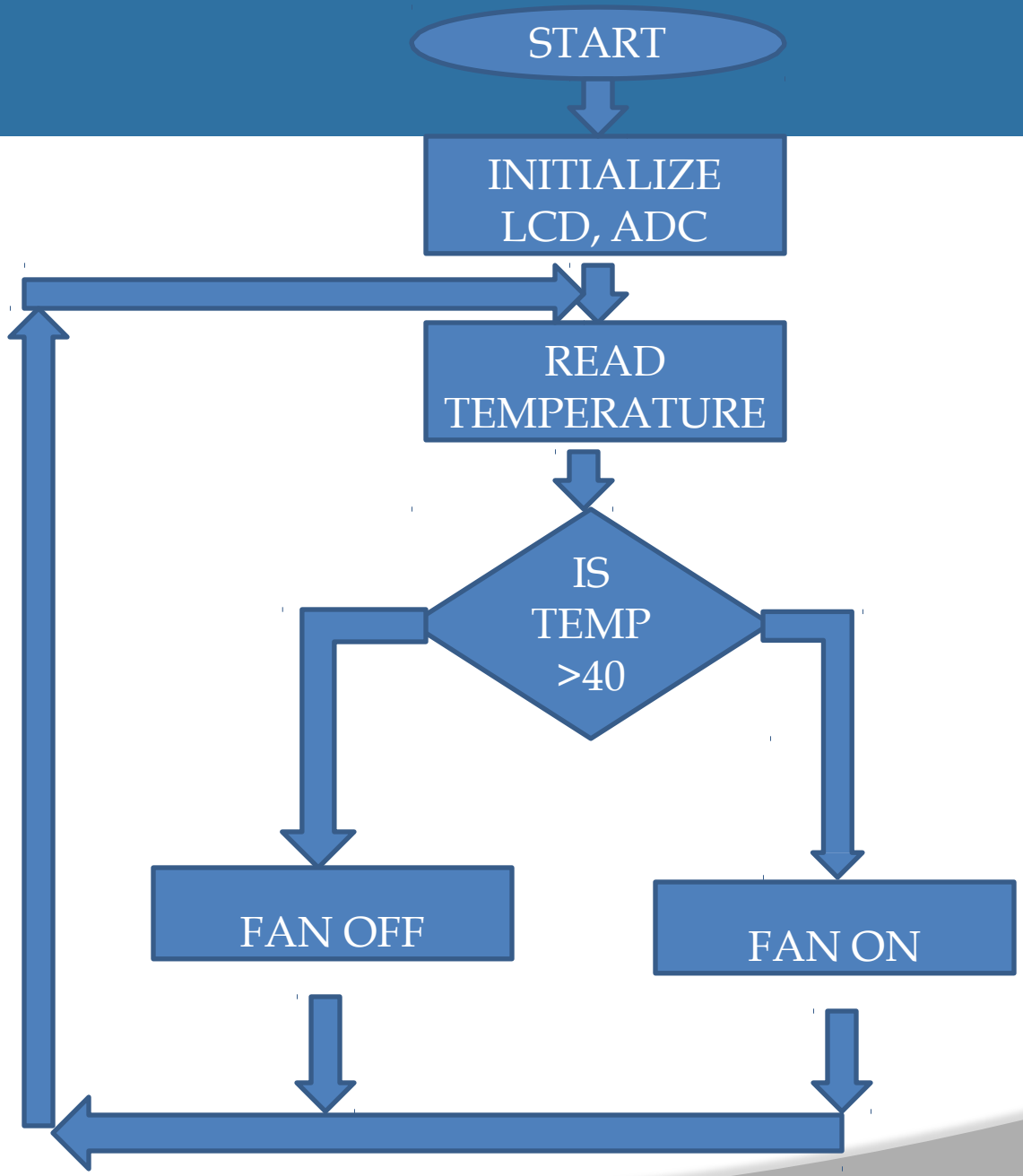
- Reduce Size

- Reduce Power.



Automatic Temperature Monitoring & 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

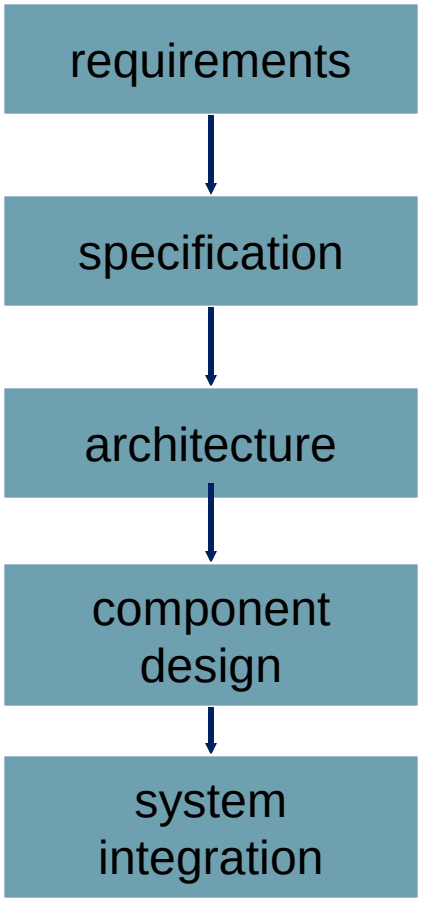
Glass braking sensor

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

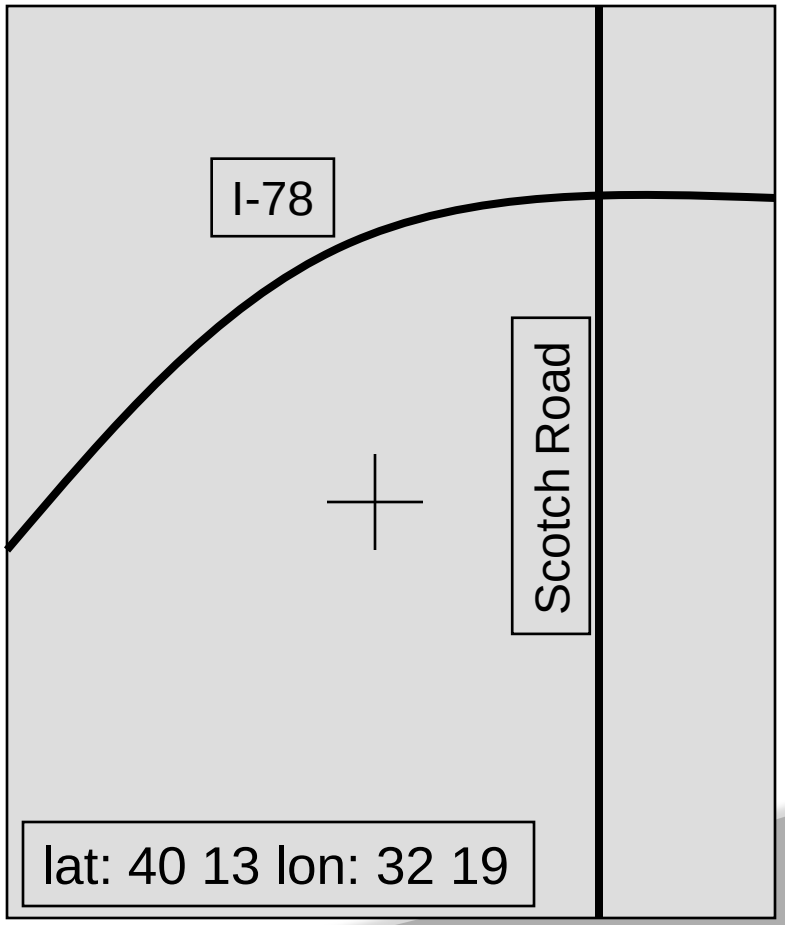
- Functional requirements:
  - output as a function of input.
- Non-functional requirements:
  - time required to compute output;
  - size, weight, etc.;
  - power consumption;
  - reliability;
  - etc.

Our requirements form

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.

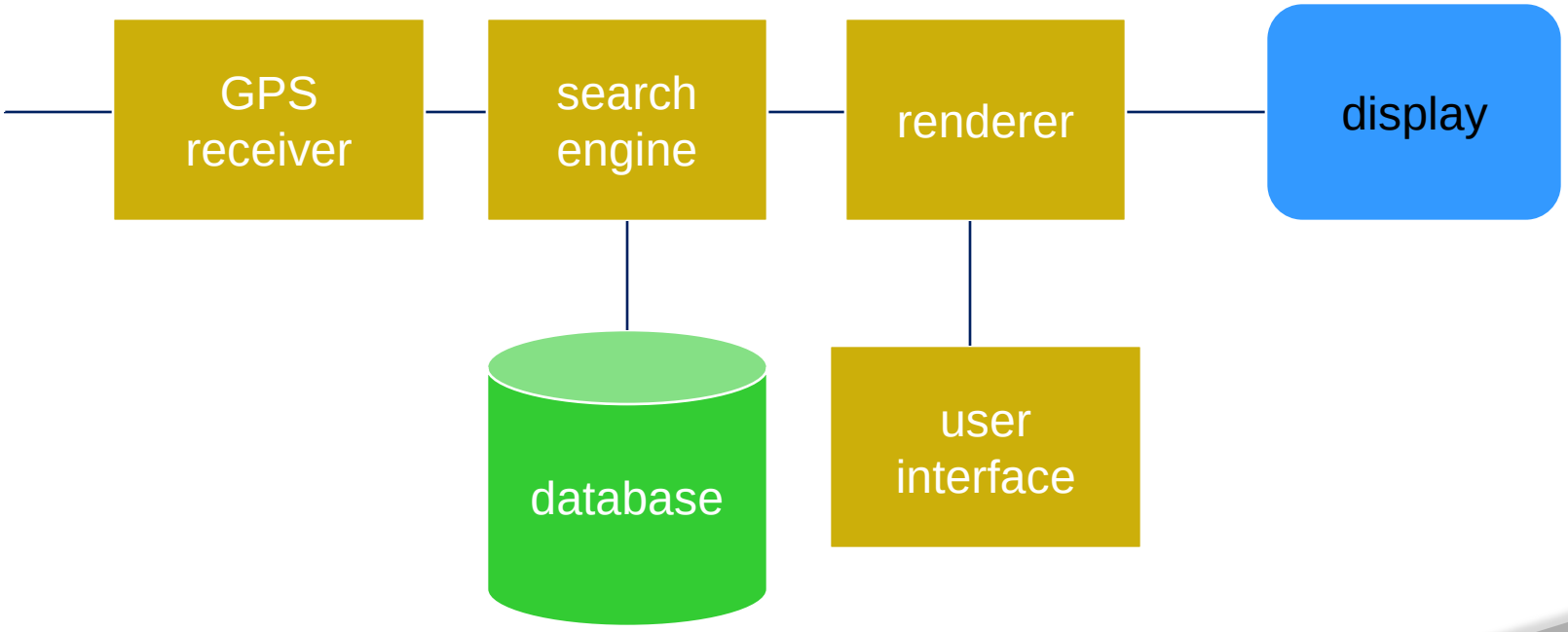
GPS specification

- 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.

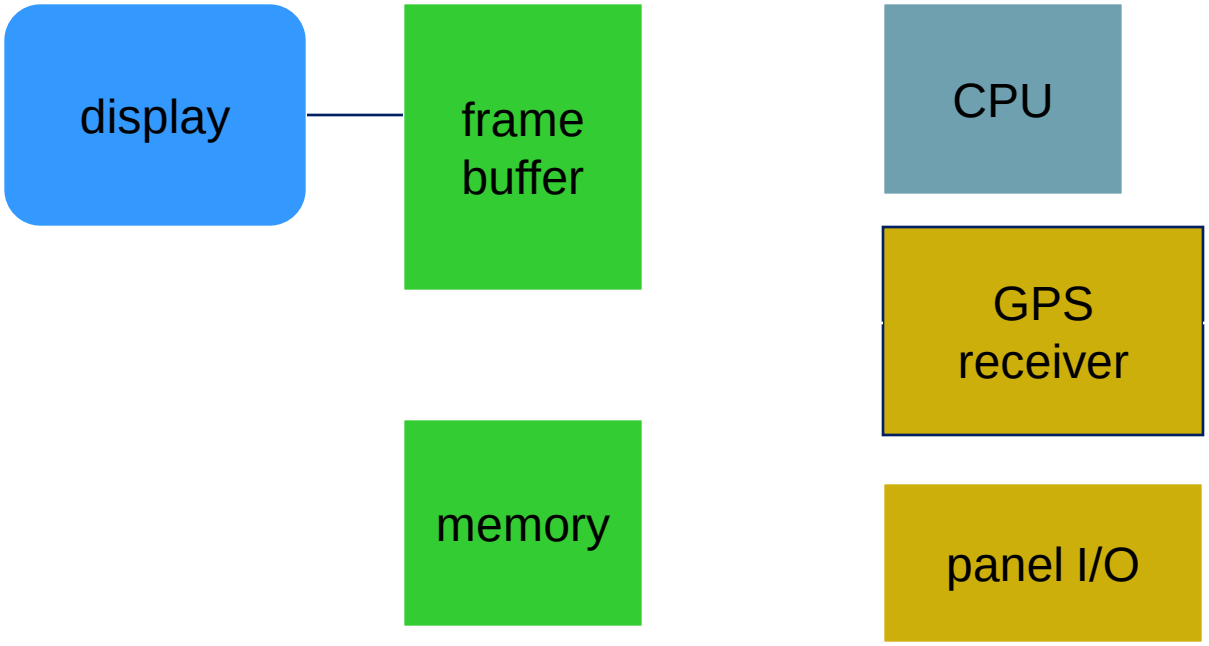
Architecture design

- 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.

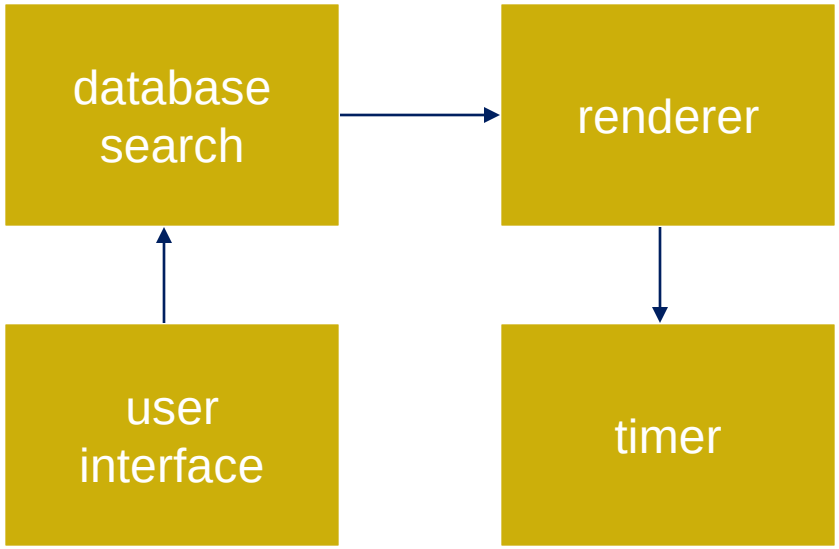
GPS moving map block diagram



GPS moving map hardware architecture



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.

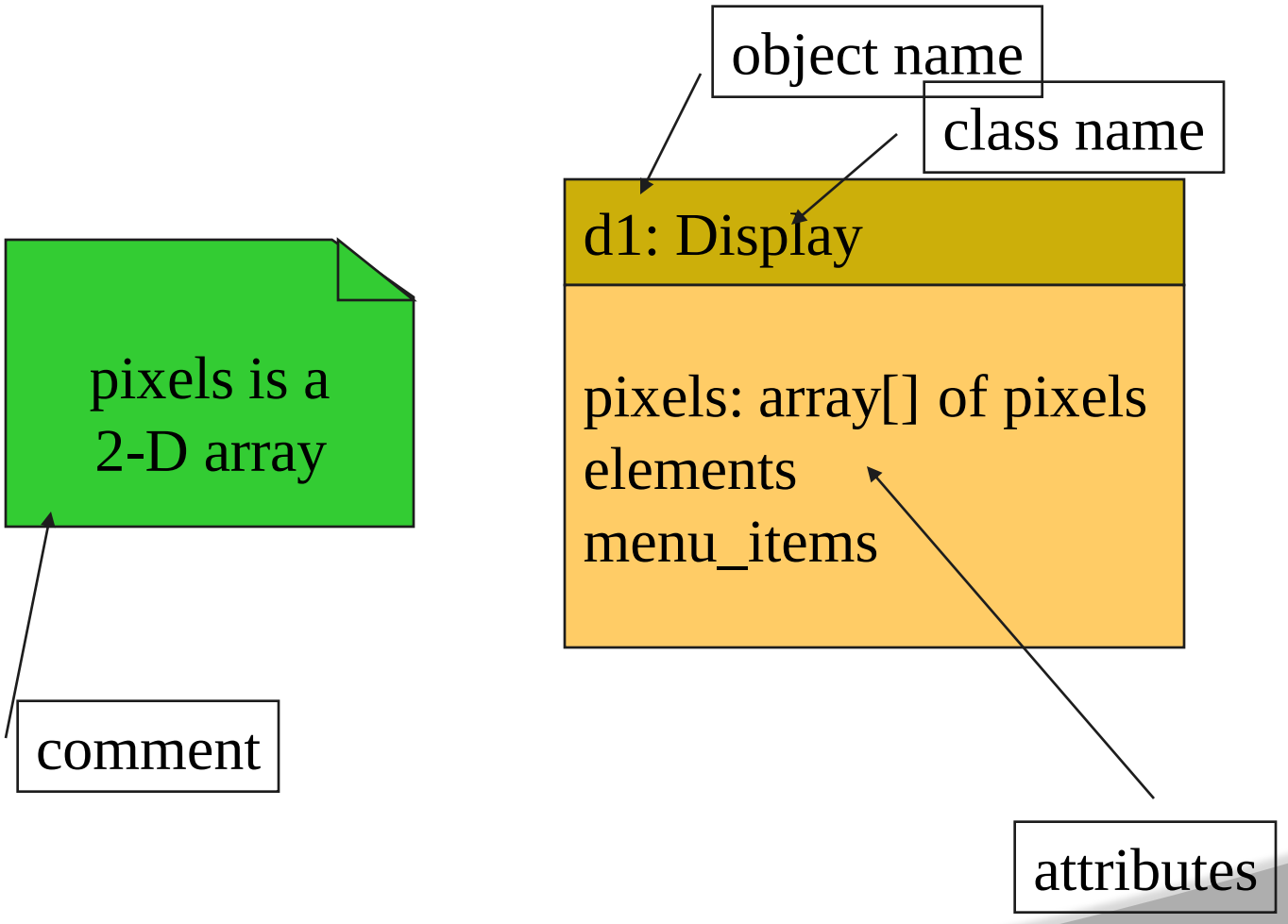
Objects and classes

- 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.

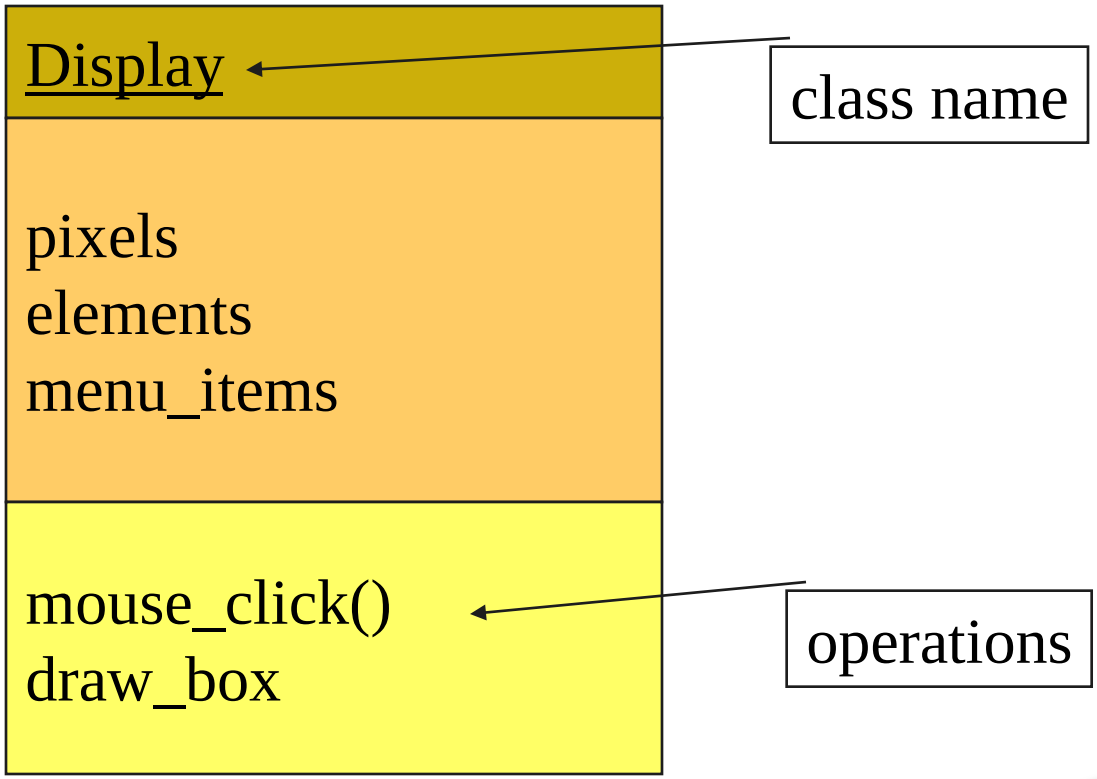
Relationships between objects and classes

- 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

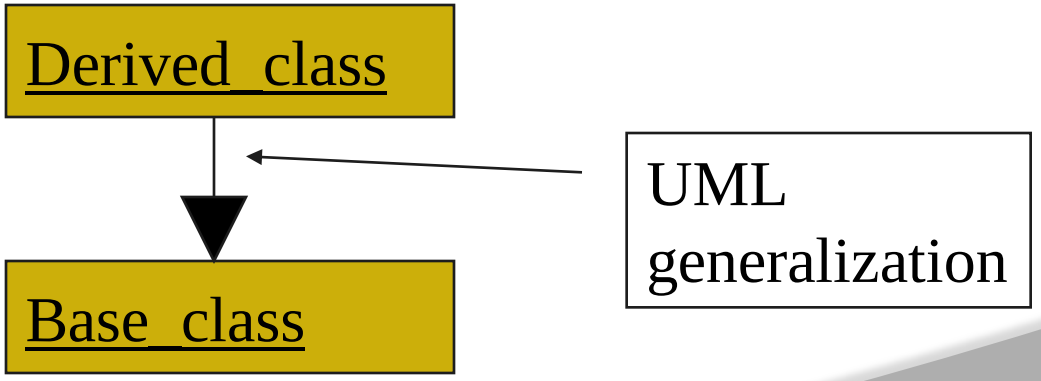
- The operations provide the abstract interface between the class's implementation and other classes.
- Operations may have arguments, return values.
- An operation can examine and/or modify the object's state.

Choose your interface properly

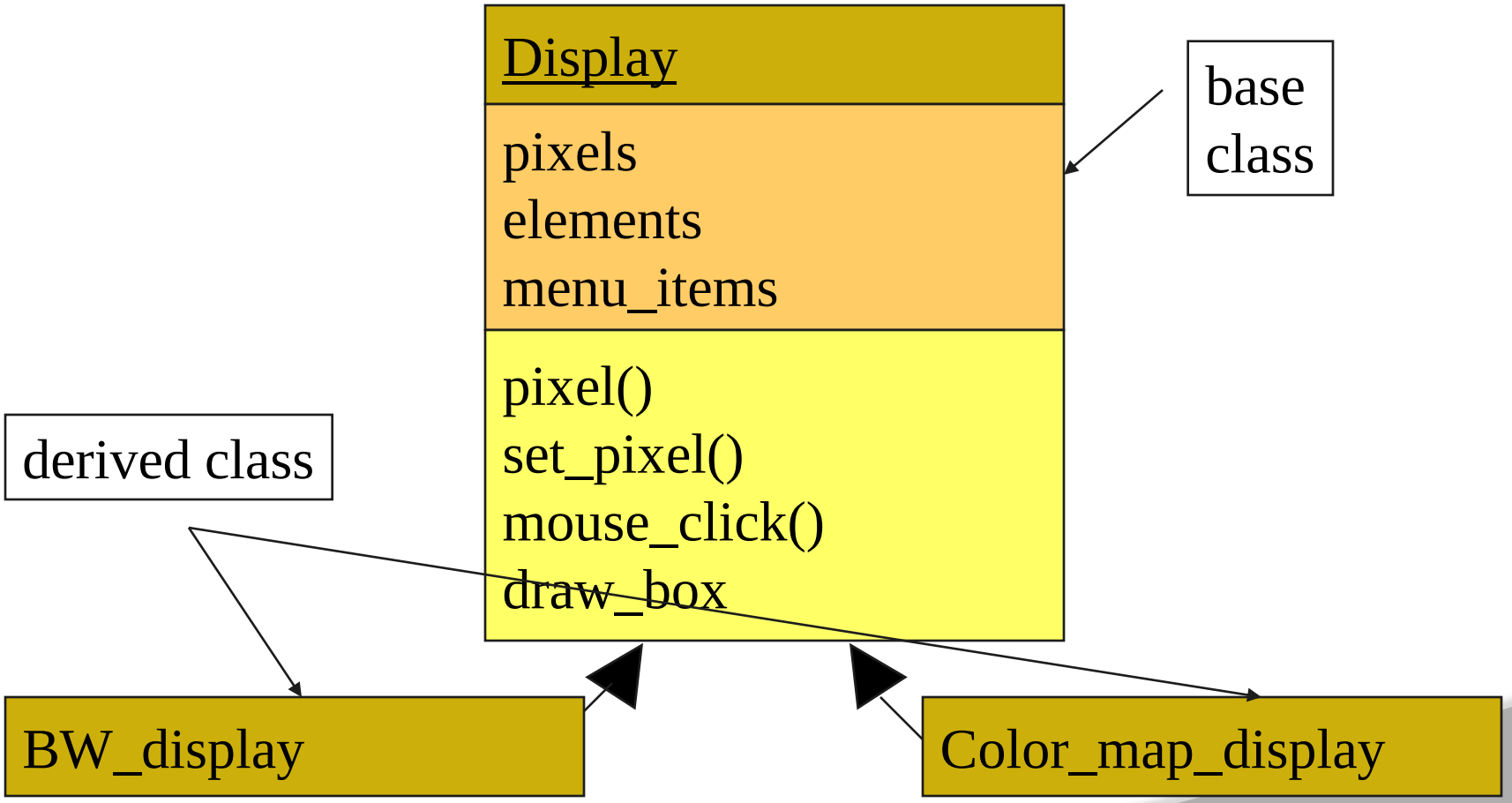
- If the interface is too small/specialized:
  - object is hard to use for even one application;
  - even harder to reuse.
- If the interface is too large:
  - class becomes too cumbersome for designers to understand;
  - implementation may be too slow;
  - spec and implementation are probably buggy.

Class derivation

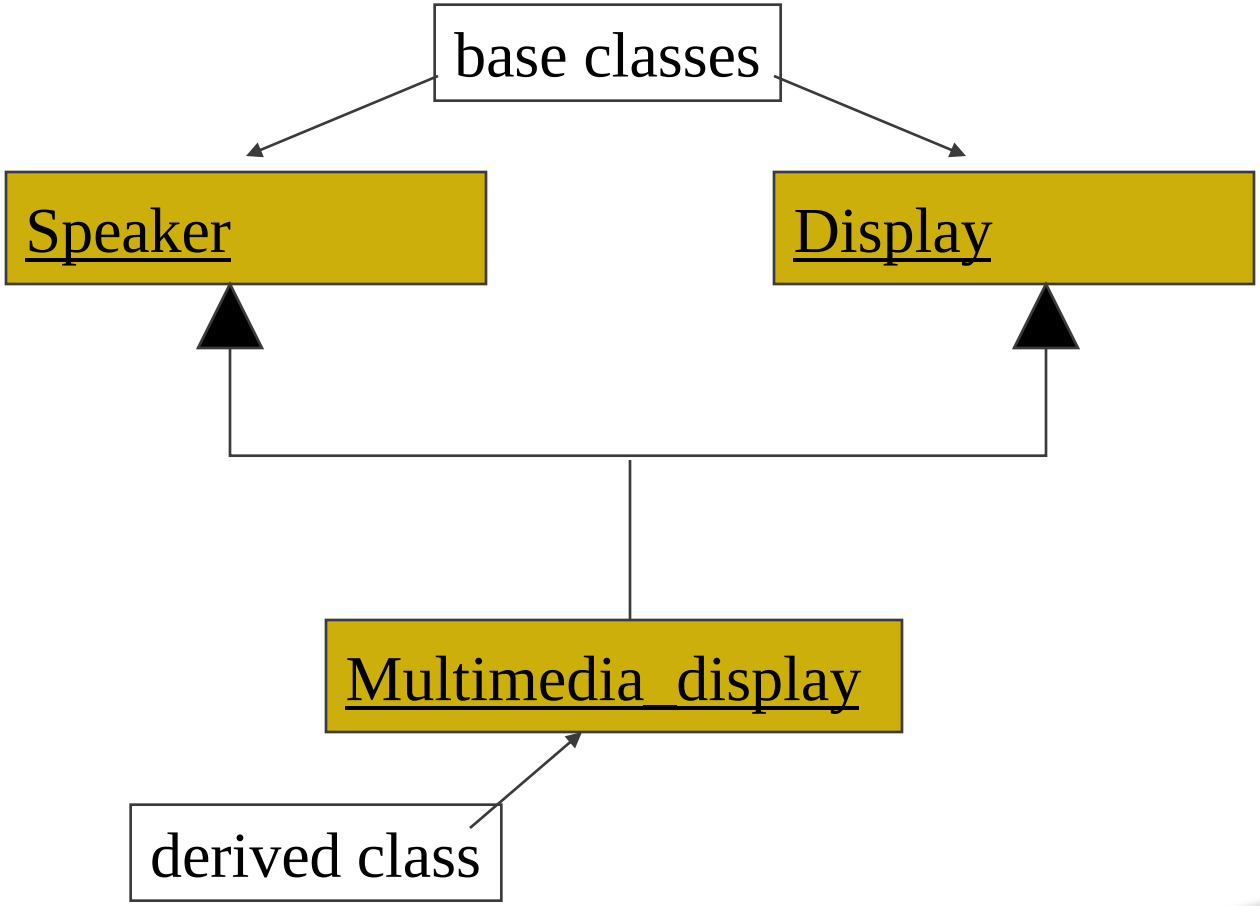
- May want to define one class in terms of another.
  - Derived class inherits attributes, operations of base class.



Class derivation example



Multiple inheritance

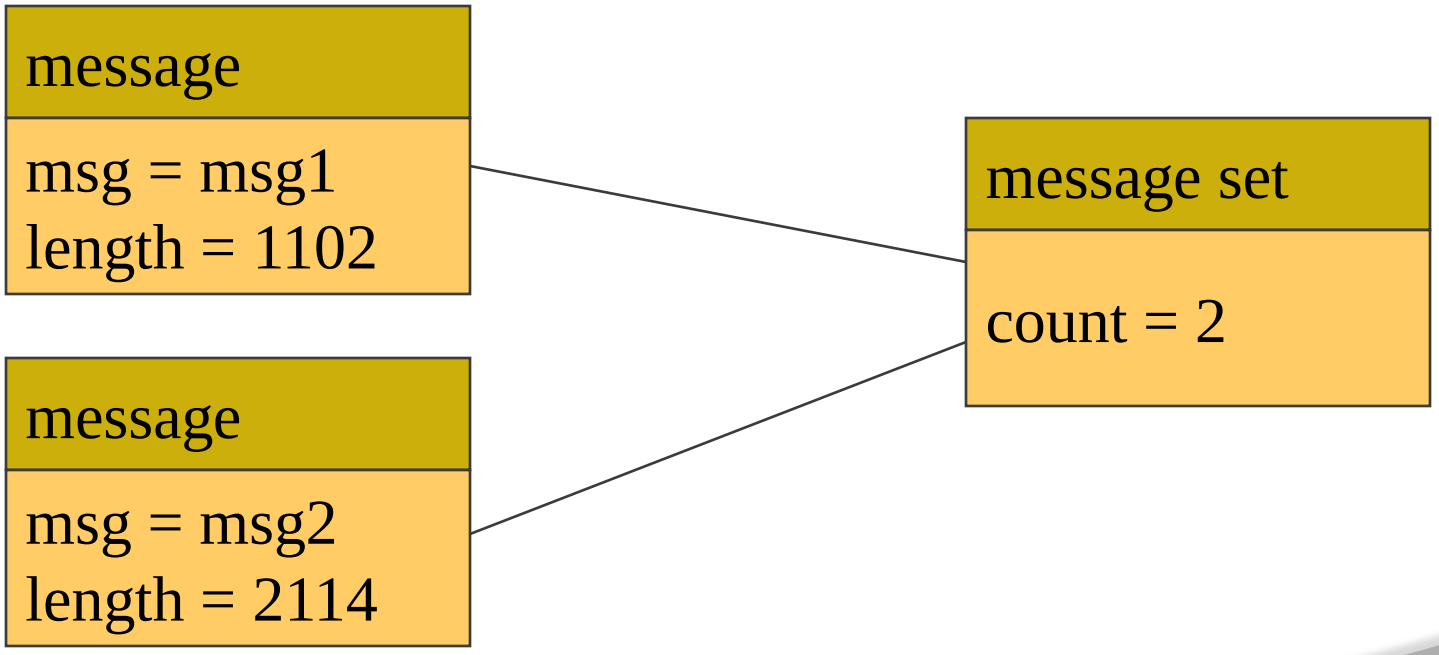


Links and associations

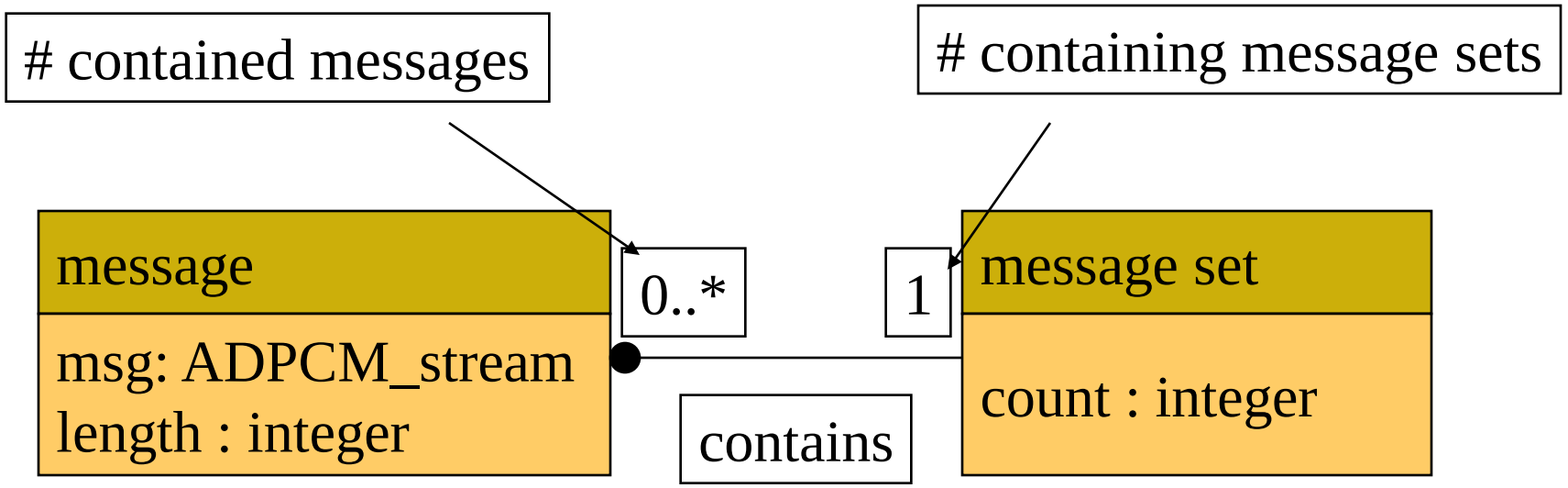
- Link: describes relationships between objects.
- Association: describes relationship between classes.

Link example

- Link defines the contains relationship:



Association example



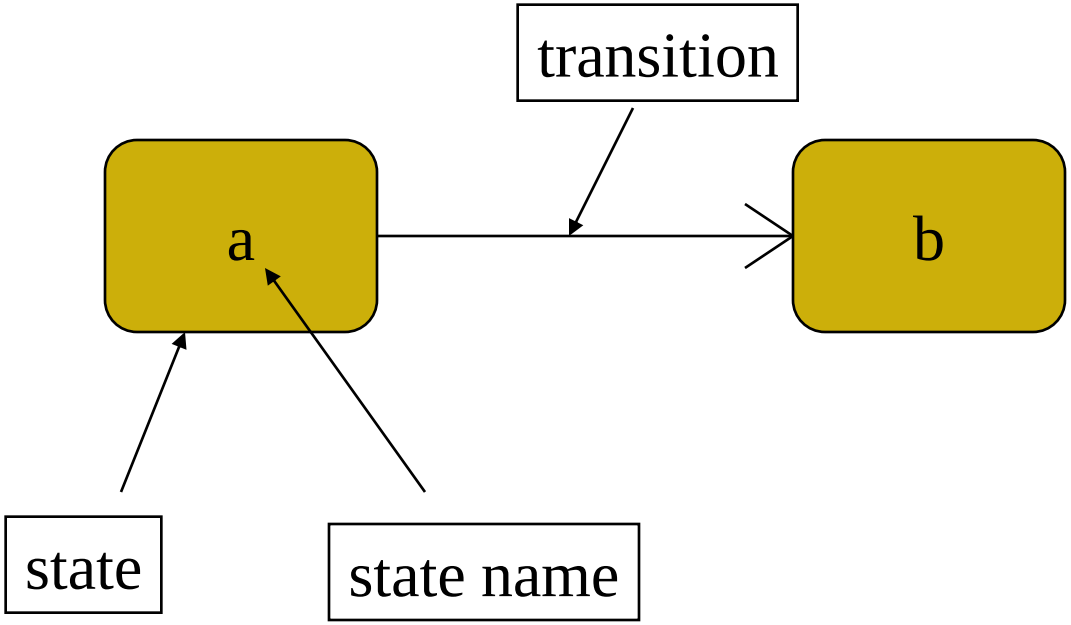
Stereotypes

- 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



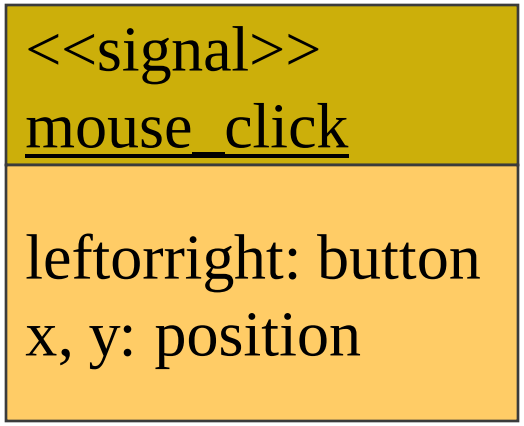
Event-driven state machines

- 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.

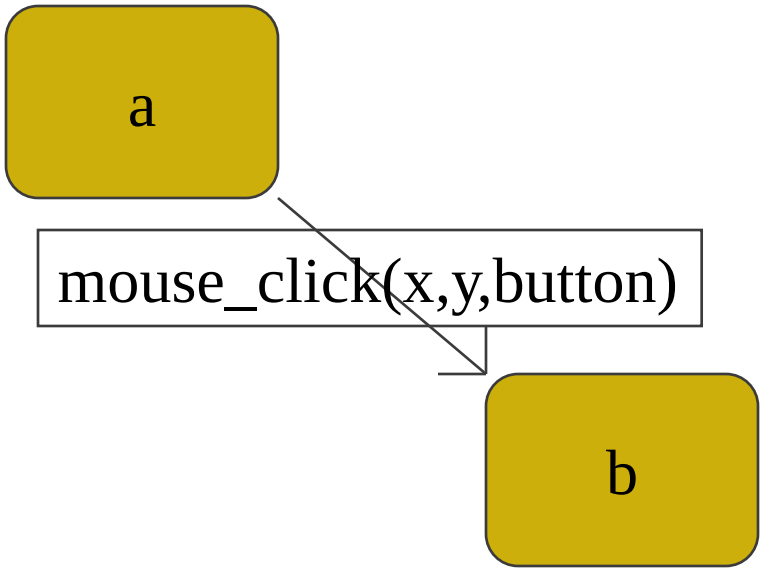
Types of events

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

Signal event

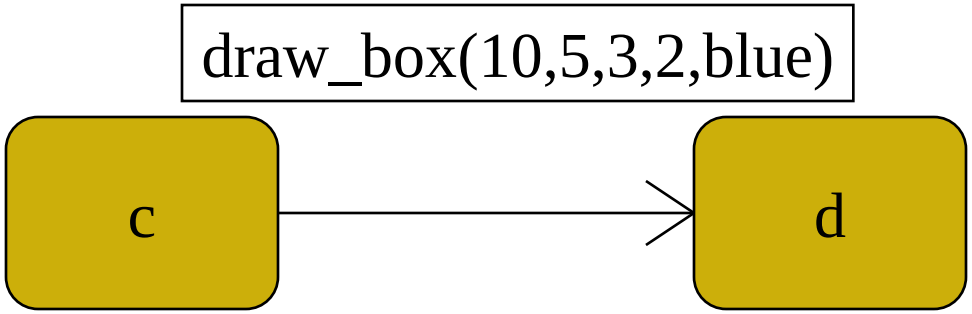


declaration

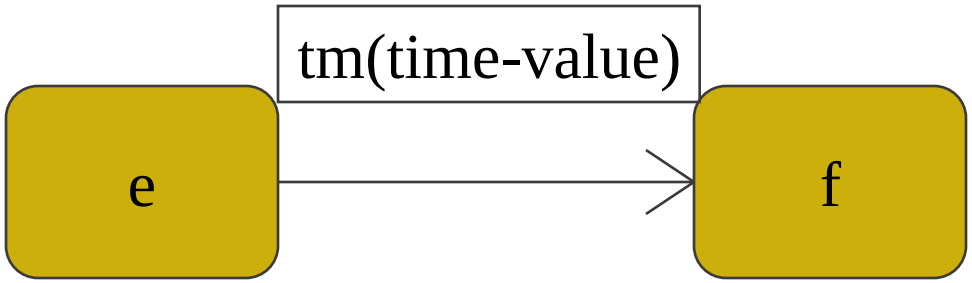


event description

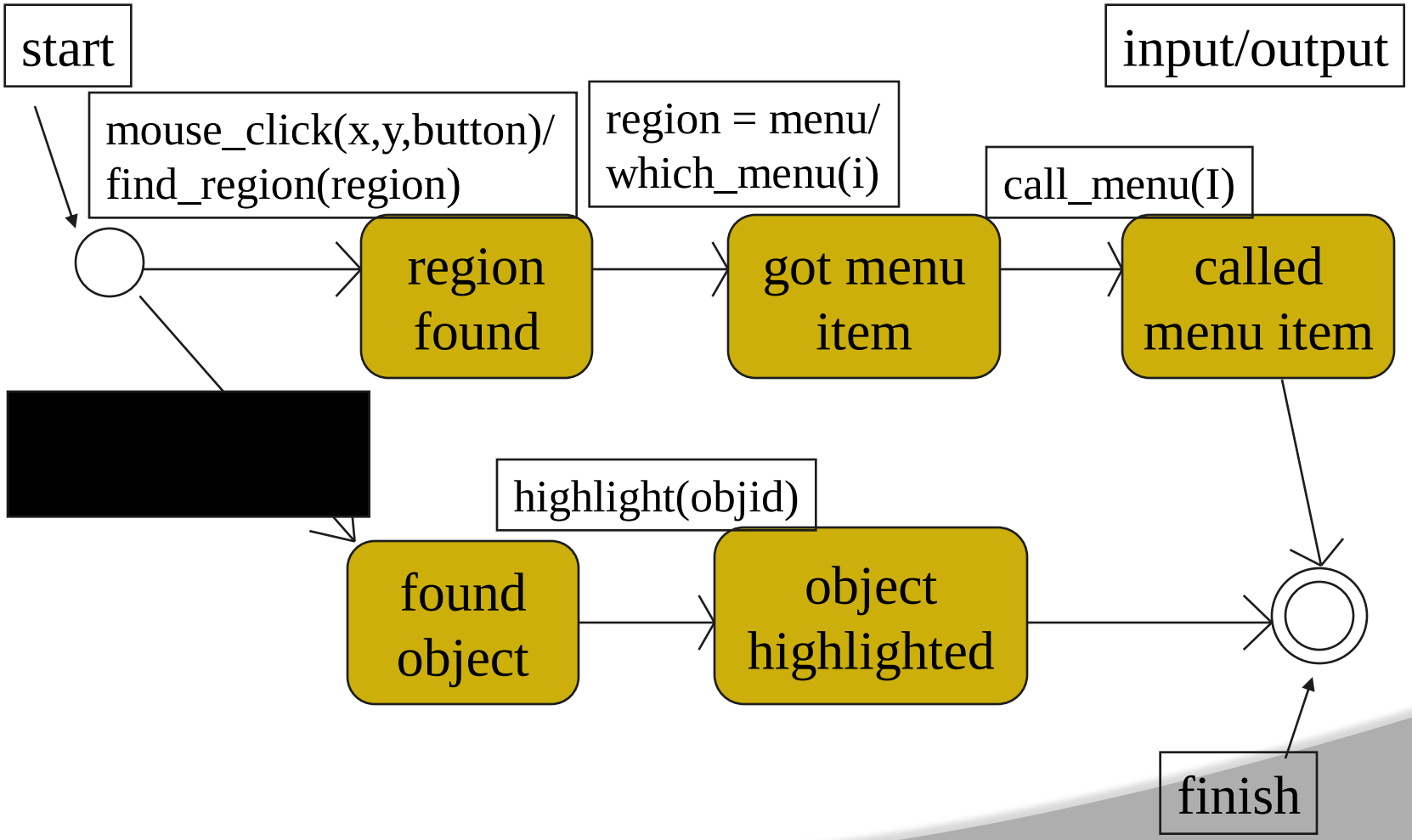
Call event



Timer event



Example state machine



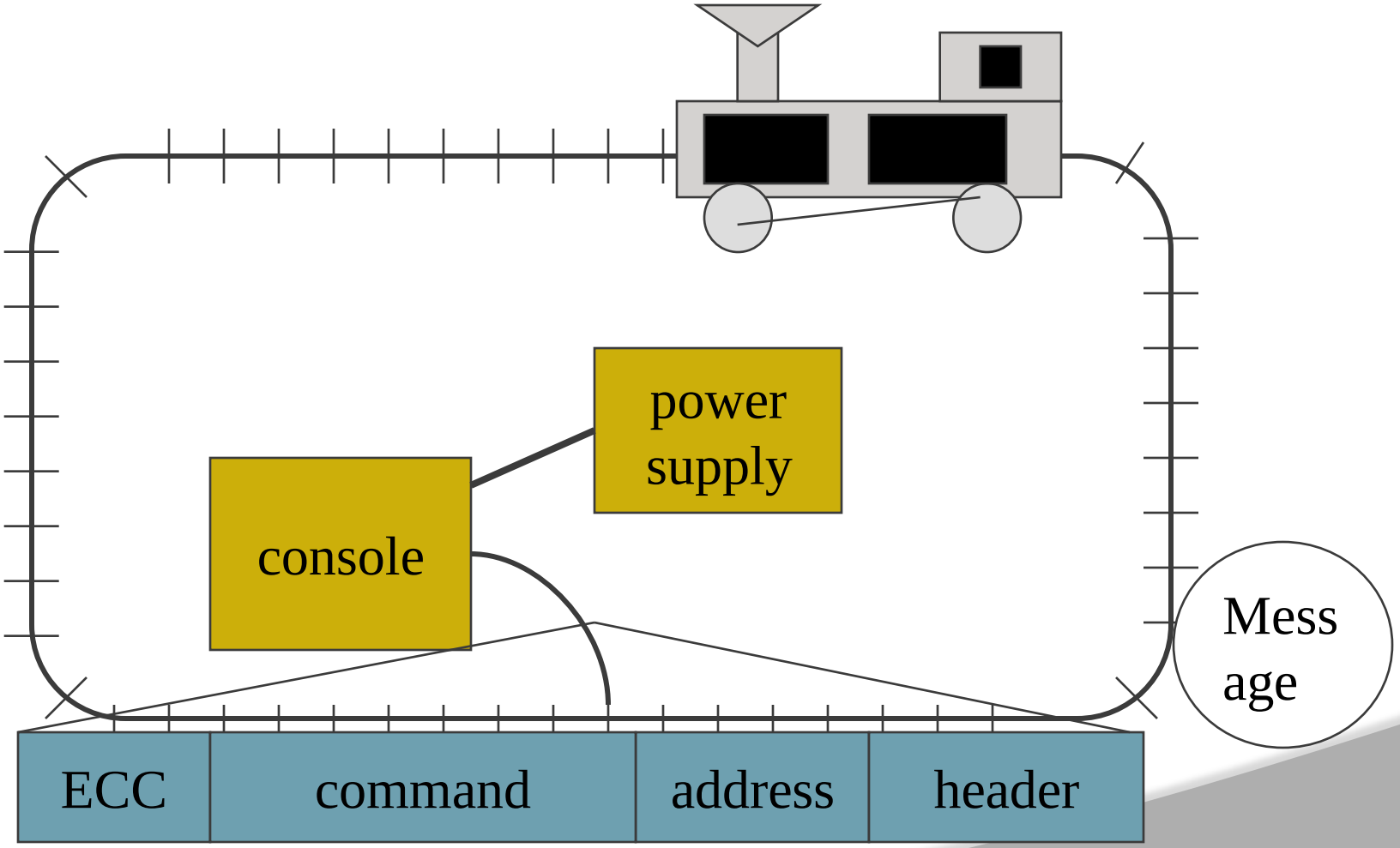
Introduction

- 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

C Looping Structures

This section looks at the most efficient ways to code for and while loops on the ARM. We start by looking at loops with a fixed number of iterations and then move on to loops with a variable number of iterations. Finally we look at loop unrolling.

Loops with a Fixed Number of Iterations

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

Here is the last version of the 64-word packet checksum routine we studied This shows how the compiler treats a loop with incrementing count i++.

C Looping Structures

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

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

This compiles to

```
checksum_v5

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

C Looping Structures

It takes three instructions to implement the for loop structure:

An ADD to increment i

A compare to check if i is less than 64

A conditional branch to continue the loop if i < 64

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

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

A conditional branch instruction

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

REGISTER ALLOCATION

The compiler attempts to allocate a processor register to each local variable you use in a C function. It will try to use the same register for different local variables if the use of the variables do not overlap.

When there are more local variables than available registers, the compiler stores the excess variables on the processor stack. These variables are called *spilled* or *swapped out* variables since they are written out to memory (in a similar way virtual memory is swapped out to disk).

Spilled variables are slow to access compared to variables allocated to registers.

To implement a function efficiently, you need to minimize the number of spilled variables,ensure that the most important and frequently accessed variables are stored in registers

FUNCTION CALLS

The ARM Procedure Call Standard (APCS) defines how to pass function arguments and return values in ARM registers. The more recent ARM-Thumb Procedure Call Standard (ATPCS) covers ARM and Thumb interworking as well.

The first four integer arguments are passed in the first four ARM registers: *r0, r1, r2*, and *r3*. Subsequent integer arguments are placed on the full descending stack, ascending in memory as in Figure 5.1. Function return integer values are passed in *r0*.

Calling Functions Efficiently

Try to restrict functions to four arguments. This will make them more efficient to call. Use structures to group related arguments and pass structure pointers instead of multiple arguments.

Define small functions in the same source file and before the functions that call them. The compiler can then optimize the function call or inline the small function.Critical functions can be inlined using the inline keyword.

POINTER ALIASING

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

Let's start with a very simple example. The following function increments two timer values by a step amount:

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

POINTER ALIASING

This compiles to

```
timers_v1
LDR   r3,[r0,#0]    ; r3 = *timer1
LDR   r12,[r2,#0]  ; r12 = *step
ADD  r3,r3,r12     ; r3 += r12
STR   r3,[r0,#0]     ; *timer1 = r3
LDR   r0,[r1,#0]     ; r0 = *timer2
LDR   r2,[r2,#0]     ; r2 = *step
ADD  r0,r0,r2        ; r0 += r2
STR    r0,[r1,#0]    ; *timer2 = t0
MOV  pc,r14          ; return
```

Note that the compiler loads from step twice. Usually a compiler optimization called *common sub expression elimination* would kick in so that *step was only evaluated once, and the value reused for the second occurrence. However, the compiler can't use this optimization here. The pointers timer1 and step might alias one another. In other words, the compiler cannot be sure that the write to timer1 doesn't affect the read from step.

STRUCTURE ARRANGEMENT

The way you lay out a frequently used structure can have a significant impact on its perfor- mance and code density. There are two issues concerning structures on the ARM: alignment of the structure entries and the overall size of the structure.

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

For example, consider the structure

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

BIT-fiELDS

- Bit-fields are probably the least standardized part of the ANSI C specification. The compiler can choose how bits are allocated within the bit-field container. For this reason alone, avoid using bit-fields inside a union or in an API structure definition. Different compilers can assign the same bit-field different bit positions in the container.

- It is also a good idea to avoid bit-fields for efficiency. Bit-fields are structure elements and usually accessed using structure pointers; consequently, they suffer from the pointer aliasing problems described in Section 5.6. Every bit-field access is really a memory access. Possible pointer aliasing often forces the compiler to reload the bit-field several times.

- The following example, dostages_v1, illustrates this problem. It also shows that compilers do not tend to optimize bit-field testing very well.

UNALIGNED DATA AND ENDIANNESS

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

- The ARM load and store instructions assume that the address is a multiple of the type you are loading or storing.

- If you load or store to an address that is not aligned to its type, then the behavior depends on the particular implementation. The core may generate a data abort or load a rotated value. For well-written, portable code you should avoid unaligned accesses.

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

INLINE FUNCTIONS AND INLINE ASSEMBLY

how to call functions efficiently.

- You can remove the function call overhead completely by inlining functions. Additionally many compilers allow you to include inline assembly in your C source code.

- Using inline functions that contain assembly you can get the compiler to support ARM instructions and optimizations that aren't usually available. For the examples of this section we will use the inline assembler in *armcc.*

- The main benefit of inline functions and inline assembly is to make accessible in C operations that are not usually available as part of the C language. It is better to use inline functions rather than #define macros because the latter doesn't check the types of the function arguments and return value.

PORTABILITY ISSUES

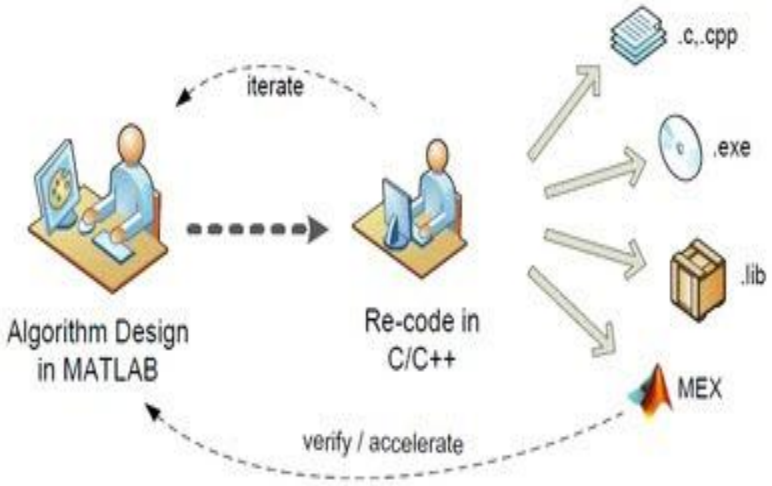
Here you may encounter when porting C code to the ARM.

The char *type*. On the ARM, char is unsigned rather than signed as for many other processors. A common problem concerns loops that use a char loop counter i and the continuation condition i ≥ 0, they become infinite loops. In this situation, *armcc*

Embedded C

Embedded C Programming is the soul of the processor functioning inside each and every embedded system we come across in our daily life, such as mobile phone, washing machine, and digital camera.

Each processor is associated with an embedded software. The first and foremost thing is the embedded software that decides functioning of the embedded system. Embedded C language is most frequently used to program the microcontroller.



The Structure of an Embedded C Program

comments
Pre processor directives
global variables
main() function
{
local variables
statements
……..
……..
}
fun(1)
{
local variables
statements
……..
……..
}

The Structure of an Embedded C Program

Comments: In embedded C programming language, we can place comments in our code which helps the reader to understand the code easily.

C=a+b; /* add two variables whose value is stored in another variable C*/

Preprocessor directives: All the functions of the embedded C software are included in the preprocessor library like "#includes<reg51.h>, #defines". These functions are executed at the time of running the program.

Global variable

A global variable is a variable that is declared before the main function, and can be accessed on any function in the program.

```
#include<reg51.h>
sbit a=p1^5;          /*global declaration*/

void main()
{
```

The Structure of an Embedded C Program

Local variable

A local variable is a variable declared within a function, and it is valid only to be used within that function.

```
void main()
{

    unsigned int k;      /*local declaration*/
    a=0x00;
    while(1)
    {
```

Main () function

The execution of a program starts with the main function. Every program uses only one main () function.

Advantages of embedded C program

- Its takes less time to develop application program.
- It reduces complexity of the program.
- It is easy to verify and understand.
- It is portable in nature from one controller to another.

Example Program

1. Write a Program to toggle microcontroller port0 continuously.

```
#include<reg51.h>          /*prepocessor directive */

void main()
{

   unsigned int i;          /*local variable*/

   P0=0x00;
   while(1)
    {
    P0=oxff;               /*statements*/
    for(i=0;i<255;i++);
    P0=0x00;
    for(i=0;i<255;i++);
    }
```

2. write a program to toggle P1.5 of 8051 microcontroller

```
#include<reg51.h>

sbit a=p1^5;

void main()
{

     unsigned int k;
     a=0x00;
     while(1)
     {
         a=0xff;
        for(i=0;i<255;i++);
         a=0x00;
        for(i=0;i<255;i++);
      }
}
```

Example Program

3. WAP to monitor P2.5 and mov the values of P2.5 to P3.6

```
#include<reg51.h>
 sbit a=P2^5;
 sbit b=P3.6;
 bit c;
 void main()
  {

      if(a==0)
      {
          b=a;
      }
   }
```

4. WAP to Increament the values of port1 from 0 to FF

```
#include<reg51.h>
  SFR a=0x00;
  void main()
  {
        unsigned char i;
        P1=0x00;
        a=0;
        while(1)
         {
         for(i=0;i<=255;i++)
          {
            P1++;
             a++;
          }
         }
    }
```

Example Program

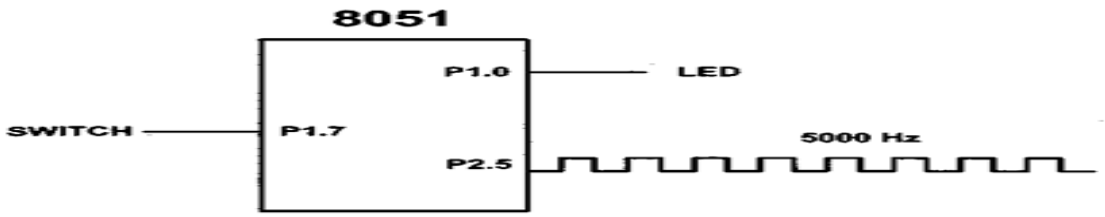
Write a C program that continuously gets a single bit of data from PI. 7 and sends it to Pl.O, while simultanepusly creating a square wave of 200 (as period on pin P2.5. Use timer 0 to create the square wave. Assume that XTAL = 11.0592 MHz.

We will use timer 0 in mode 2 (auto-reload). One half of the period is 100 µs.
100 /1. 085 µs = 92, and TH0 = 256 – 92 = 164 or A4H

```
#include <reg51.h>

sbit SW      = P1^7;
sbit IND     = P1^0;
sbit WAVE    = P2^5;

void timer0(void) interrupt 1
   {
      WAVE = ~WAVE;        //toggle pin
   }

void main()
   {
      SW = 1;              //make switch input
      TMOD = 0x02;
      TH0 = 0xA4;          //TH0 = -92
      IE = 0x82;           //enable interrupts for timer 0
      while(1)
         {
            IND = SW;      //send switch to LED
         }
   }
```

200 µs / 2 = 100 µs

100 µs / 1.085 µs = 92



Example Program

Write a C program that continuously gets a single bit of data from PI. 7 and sends it to Pl.O in the main, while simultaneously (a) creating a square wave of 200 us period on pin P2.5, and (b) sending letter ‘A’ to the serial port. Use Timer 0 to create the square wave. Assume that XTAL = 11.0592 MHz. Use the 9600 baud rate.

Example Program

We will use Timer 0 in mode 2 (auto-reload). TH0 = 100/1.085 μs = –92, which is A4H

```
#include <reg51.h>

sbit SW      = P1^7;
sbit IND     = P1^0;
sbit WAVE    = P2^5;

void timer0(void) interrupt 1
   {
      WAVE = ~WAVE;      //toggle pin
   }

void serial0() interrupt 4
   {
      if(TI == 1)
         {
            SBUF = 'A';   //send A to serial port
            TI = 0;       //clear interrupt
         }
      else
         {
            RI = 0;       //clear interrupt
         }
   }

void main()
   {
      SW = 1;             //make switch input
      TH1 = -3;           //9600 baud
      TMOD = 0x22;        //mode 2 for both timers
      TH0 = 0xA4;         //-92=A4H for timer 0
      SCON = 0x50;
      TR0 = 1;
      TR1 = 1;                  //start timer
      IE = 0x92;          //enable interrupt for T0
      while(1)            //stay here
         {
            IND = SW;     //send switch to LED
         }
   }
```

Example Program

Write a C program using interrupts to do the following:

Generate a 10000 Hz frequency on P2.1 using TO 8-bit auto-reload, Use timer 1 as an event counter to count up a 1-Hz pulse and display it on PO. The pulse is connected to EX1.Assume that XTAL = 11.0592 MHz's Set the baud rate at 9600.

Example Program

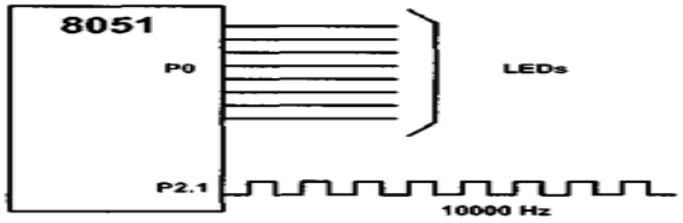
Solution:

```
#include <reg51.h>

sbit WAVE = P2^1;
unsigned char cnt;

void timer0() interrupt 1
   {
      WAVE = ~WAVE;                 //toggle pin
   }
void timer1() interrupt 3
   {
      cnt++;                        //increment counter
      P0 = cnt;                     //display value on pins
   }

void main()
   {
      cnt = 0;                      //set counter to zero
      TMOD = 0x42;
      TH0 = 0x-46;                  //10000 Hz
      IE = 0x86;                    //enable interrupts
      TR0 = 1;                      //start timer 0
      TR1 = 1;                      //start timer 1
      while(1);                     //wait until interrupted
   }
```

1 / 10000 Hz = 100 µs

100 µs / 2 = 50 µs

50 µs / 1.085 µs = 46



Example Program

The interrupt service routine should only execute the critical code; the rest of the task can be relegated to the main process by setting a flag variable. Note that since flags generally take binary values (0 or 1), these should be declared in bitwise memory wherever possible (like in 8051). This reduces the push/pop overhead and the execution time. Example:

Example Program

```
bit flag;
#pragma interrupt_handler ISR
void ISR(void)
{
flag=1;
}
void main()
{
   --
   --
   while(1)
   {
      --
      --
      if (flag)            /* Wait for the ISR to set the * flag; reset *before taking any action. */
         {
         flag = 0;
         /* Perform the required action here */
      }
   }
}
```

UNIT-III
RTOS FUNDAMENTALS AND PROGRAMMING

Operating System Basics

An Operating system (OS) is a piece of software that controls the overall operation of the Computer. It acts as an interface between hardware and application programs .It facilitates the user to format disks, create, print, copy, delete and display files, read data from files ,write data to files , control the I/O operations, allocate memory locations and process the interrupts etc.

Operating System Basics..

It provides the users an interface to the hardware resources. In a multiuser system it allows several users to share the CPU time, share the other system resources and provide inter task communication, Timers, clocks, memory management and also avoids the interference of different users in sharing the resources etc. Hence the OS is also known as a resource manager.

Types of operating systems

An Operating system (OS) is nothing but a piece of software that controls the overall operation of the Computer. It acts as an interface between hardware and application programs .It facilitates the user to format disks, create ,print ,copy , delete and display files , read data from files ,write data to files ,control the I/O operations , allocate memory locations and process the interrupts etc. It provides the users an interface to the hardware resources.

Types of operating systems..

In a multiuser system it allows several users to share the CPU time, share the other system resources and provide inter task communication, Timers, clocks, memory management and also avoids the interference of different users in sharing the resources etc. Hence the OS is also known as a resource manager.

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

REAL TIME SYSTEMS:

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

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

REAL TIME OPERATING SYSTEM (RTOS)

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

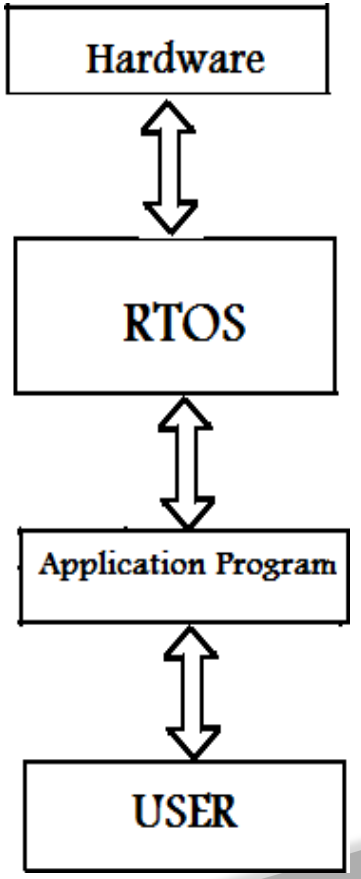
Though the real-time operating systems may or may not increase the speed of execution, but they provide more precise and predictable timing characteristics than general-purpose OS.

The figure below shows the embedded system with RTOS.

REAL TIME OPERATING SYSTEM (RTOS)

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

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



Task

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

There are two types of tasks.

(i)Simple Task(S-Task) and

(ii) Complex Task(C-Task).

Task States

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

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

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

Task States

An Idle Task does nothing .The idle task has the lowest priority.

```
void Idle task(void)
{
While(1);
}
```

Creation of a Task

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

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

Task Scheduler

Task scheduler is one of the important component of the Kernel .Basically it is a set of algorithms that manage the multiple tasks in an embedded system. The various tasks are handled by the scheduler in an orderly manner. This produces the effect of simple multitasking with a single processor. The advantage of using a scheduler is the ease of implementing the sleep mode in microcontrollers which will reduce the power consumption considerably (from mA to μA). This is important in battery operated embedded systems.

Task Scheduler….

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

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

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

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

Process or Task:

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

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

Threads

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

Multitasking

A multitasking environment allows applications to be constructed as a set of independent tasks, each with a separate thread of execution and its own set of system resources. The inter-task communication facilities allow these tasks to synchronize and coordinate their activity. Multitasking provides the fundamental mechanism for an application to control and react to multiple, discrete real-world events and is therefore essential for many real-time applications.

Multitasking….

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….

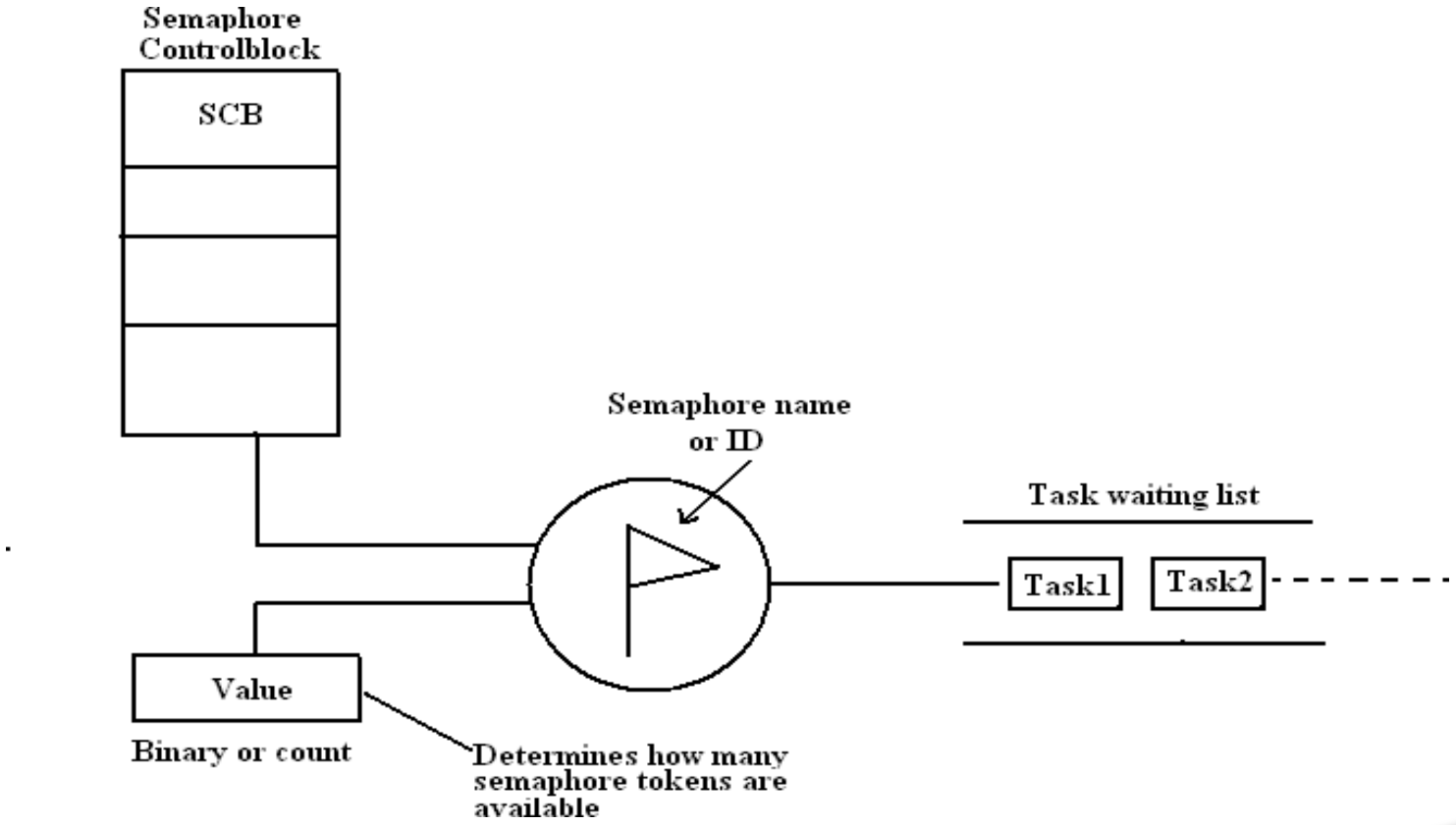
Co-operation will require that the tasks communicate with each other and share common data in an orderly an disciplined manner, without creating undue contention and deadlocks. The way in which tasks communicate and share data is to be regulated such that communication or shared data access error is 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.

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…

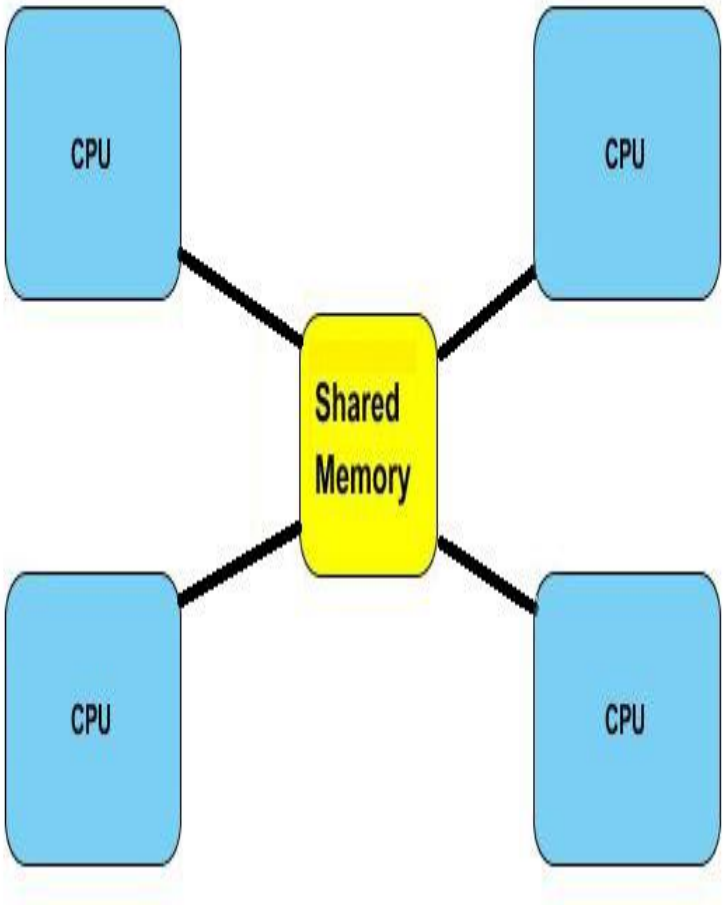
Shared memory:

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.

UNIT-IV

EMBEDDED SOFTWARE DEVELOPMENT TOOLS

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

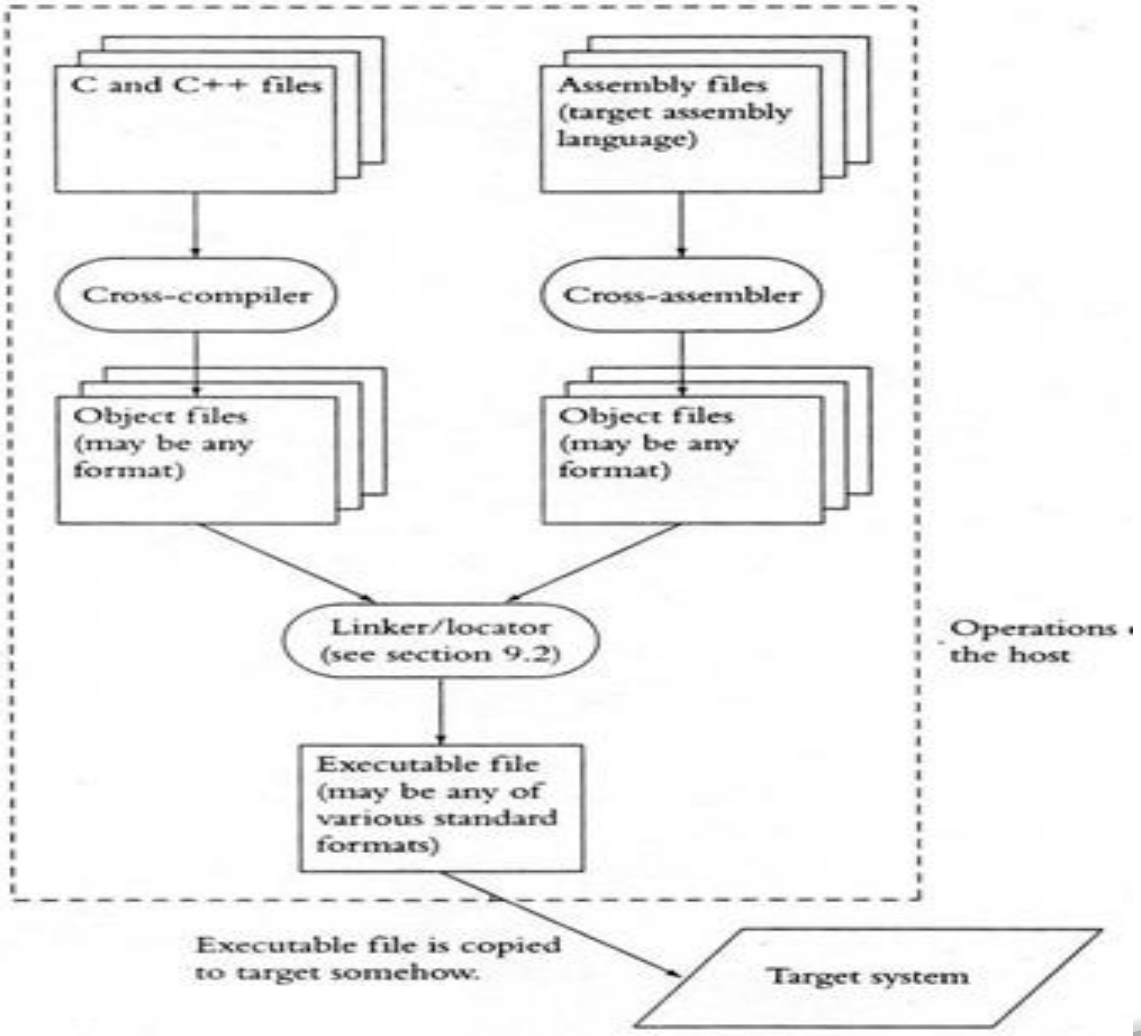
- **Cross-Compilers :**

  - 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:

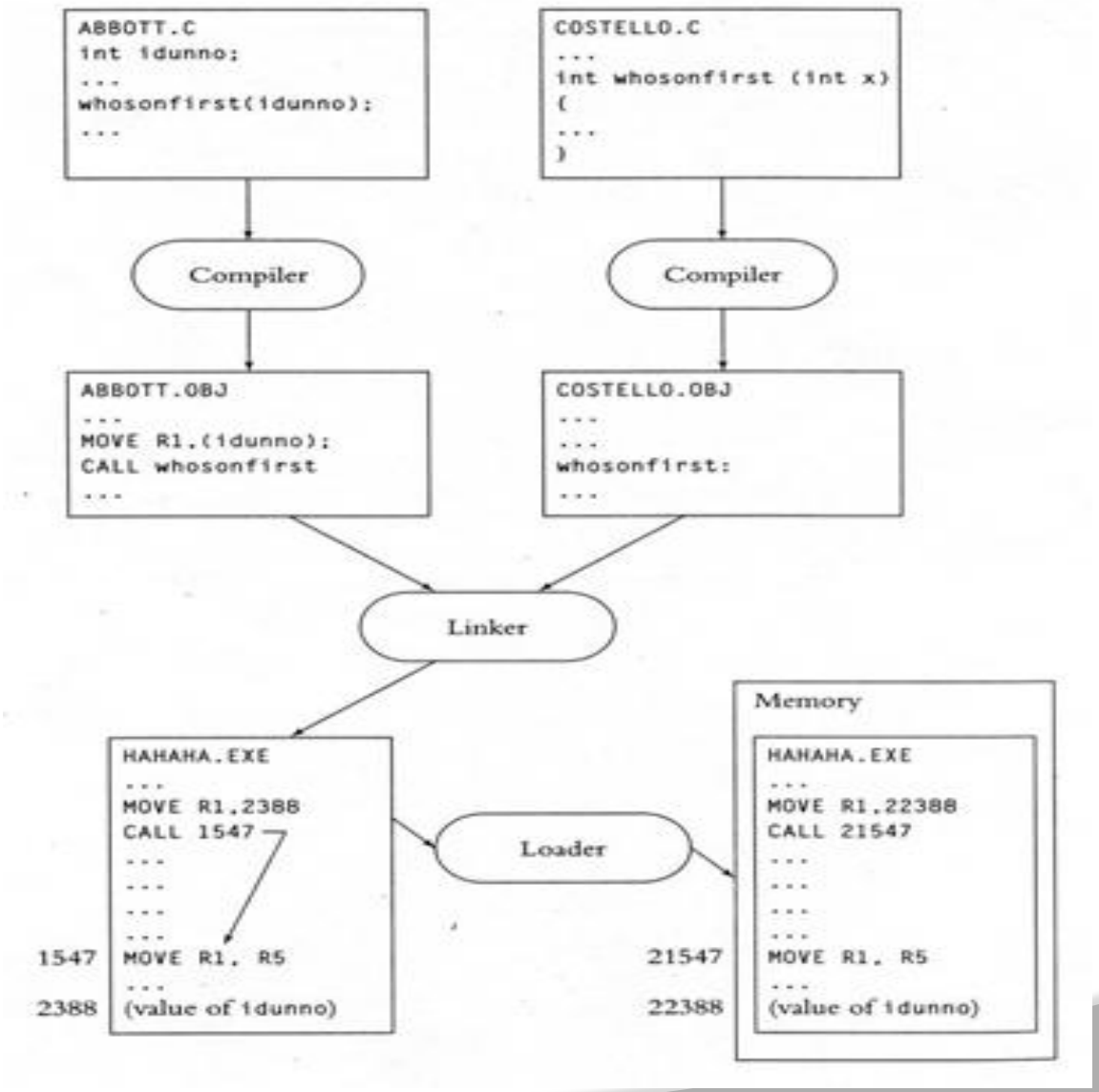
- Native linkers are different from cross-linkers (or locators) that perform additional tasks to ***locate*** embedded binary code into target processors
- Address Resolution –
  - Native Linker: produces host machine code on the hard-drive (in a named file), which the loader loads into RAM, and then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

LINKERS AND LOCATORS

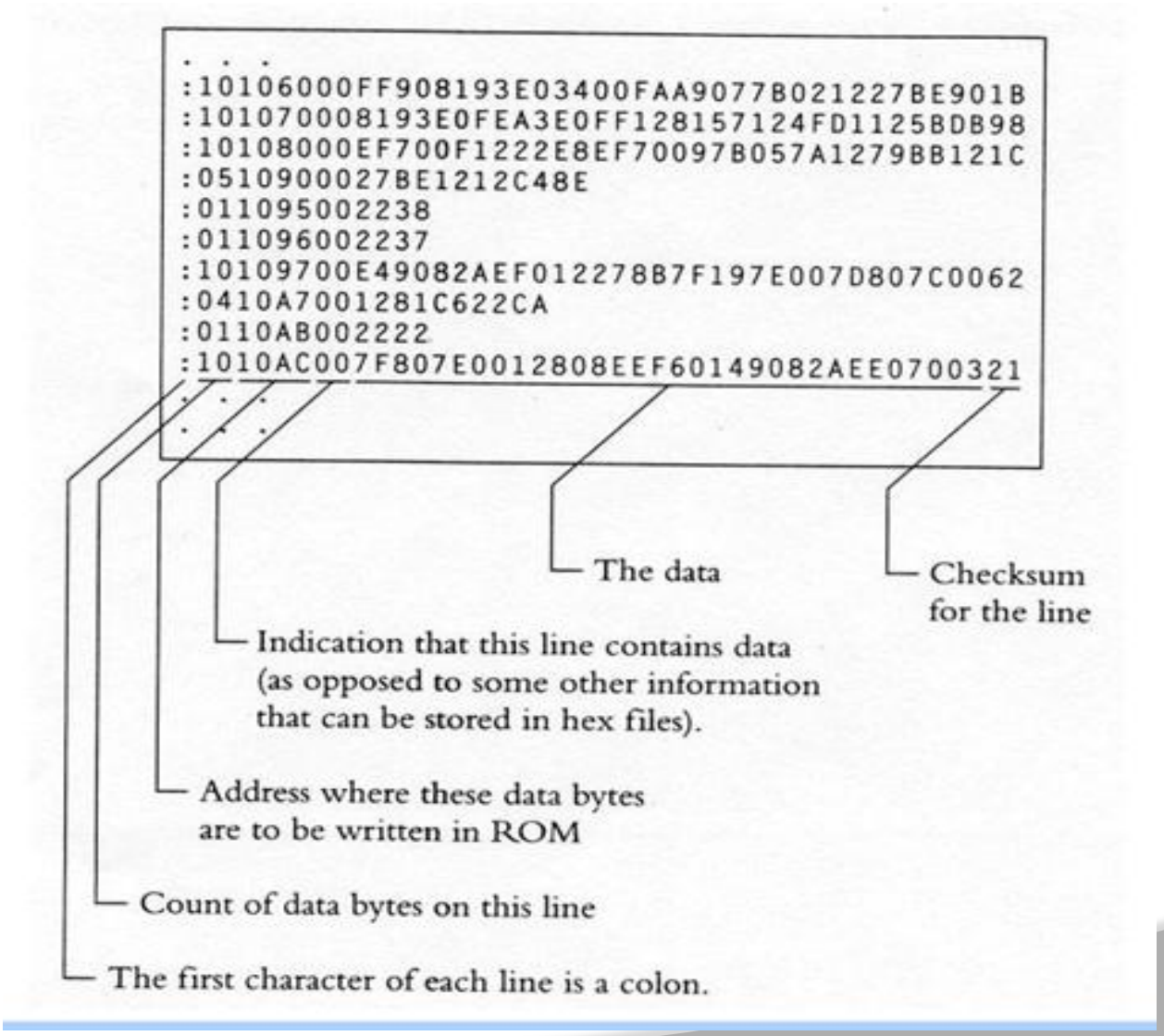
Linker/Locators for Embedded Software:

- 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



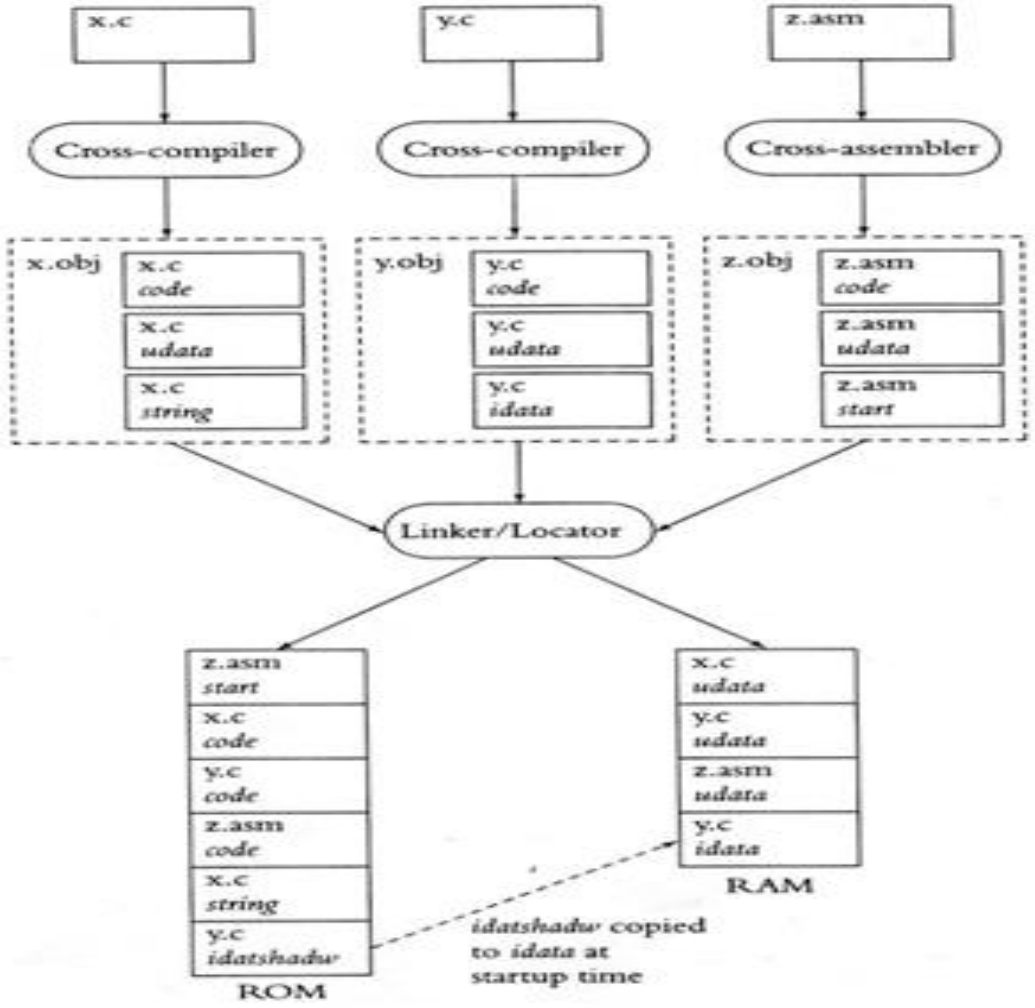
HOST AND TARGET MACHINES



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

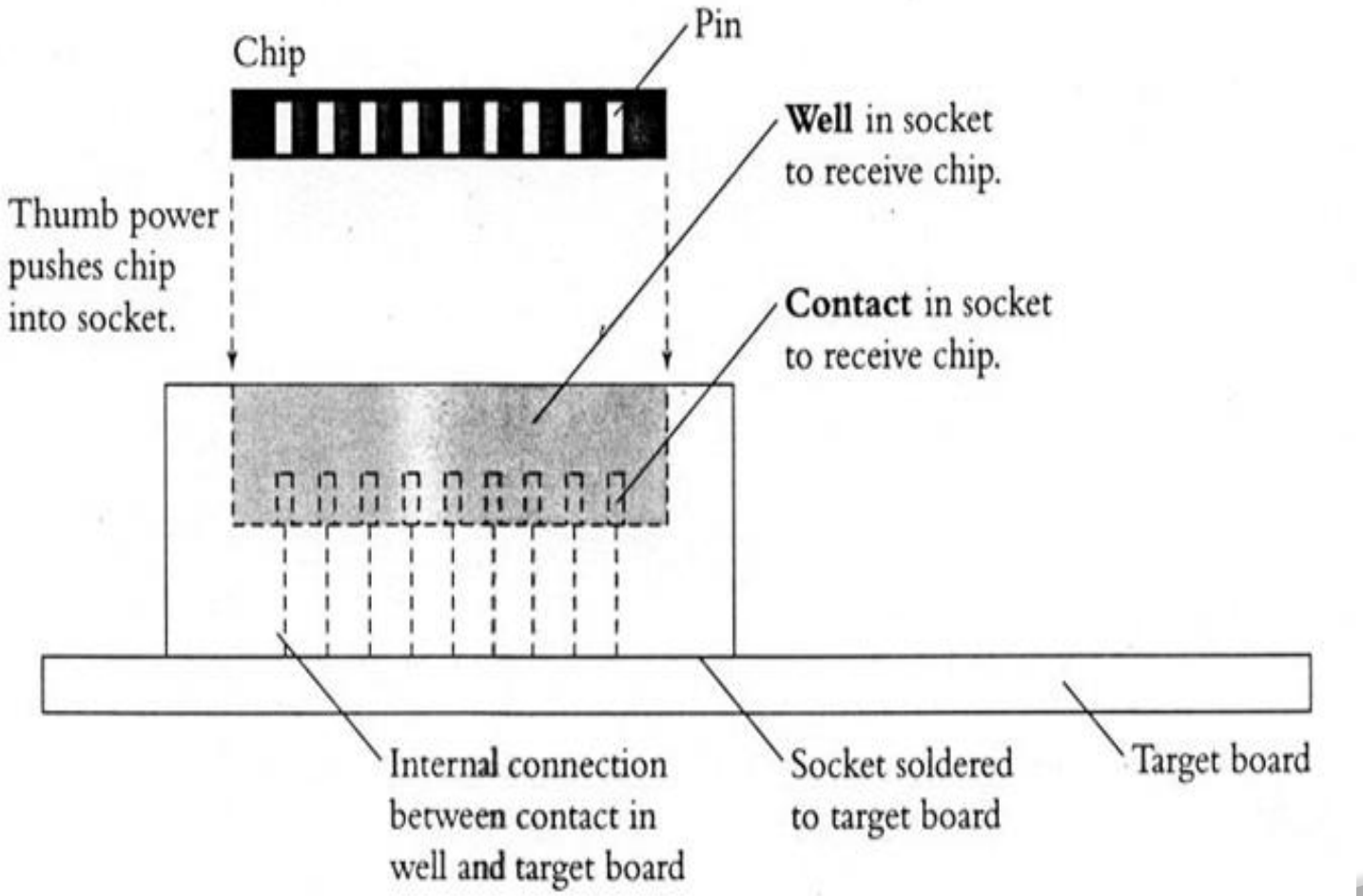
HOST AND TARGET MACHINES



Getting Embedded Software into Target System

- 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

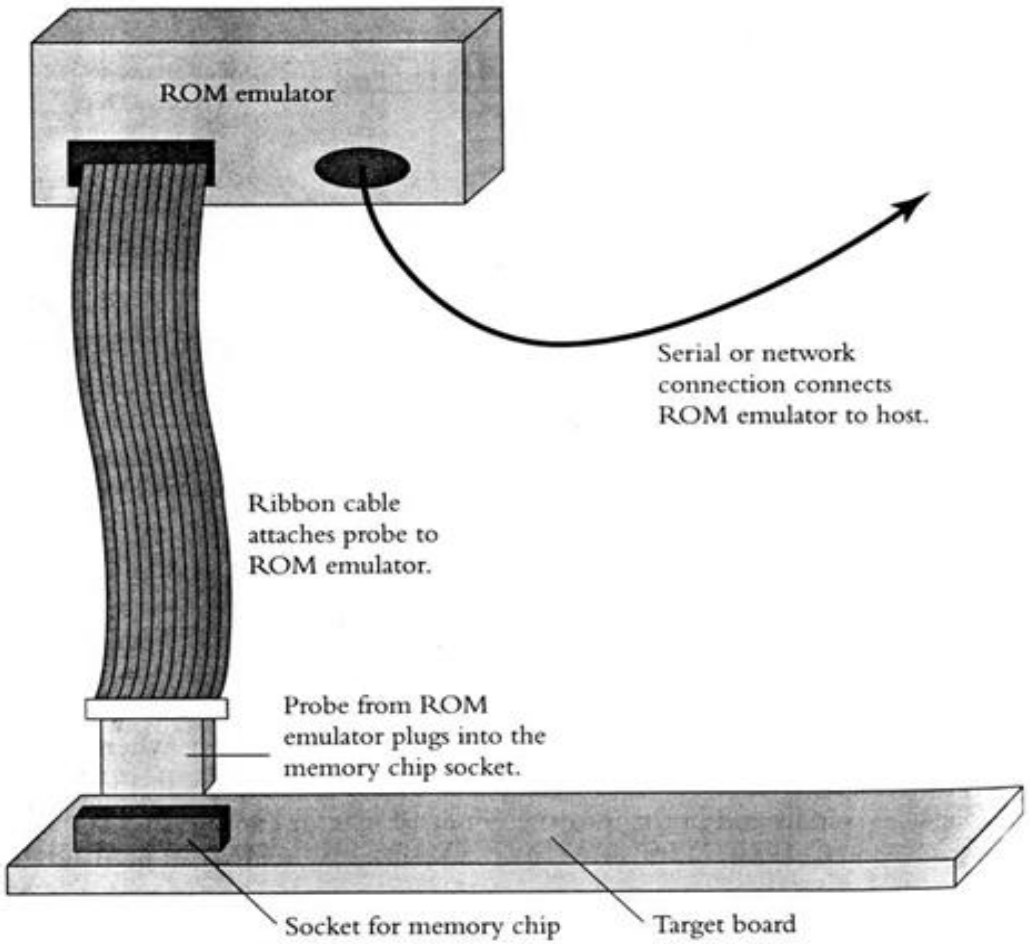
DEBUGGING TECHNIQUES

- **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

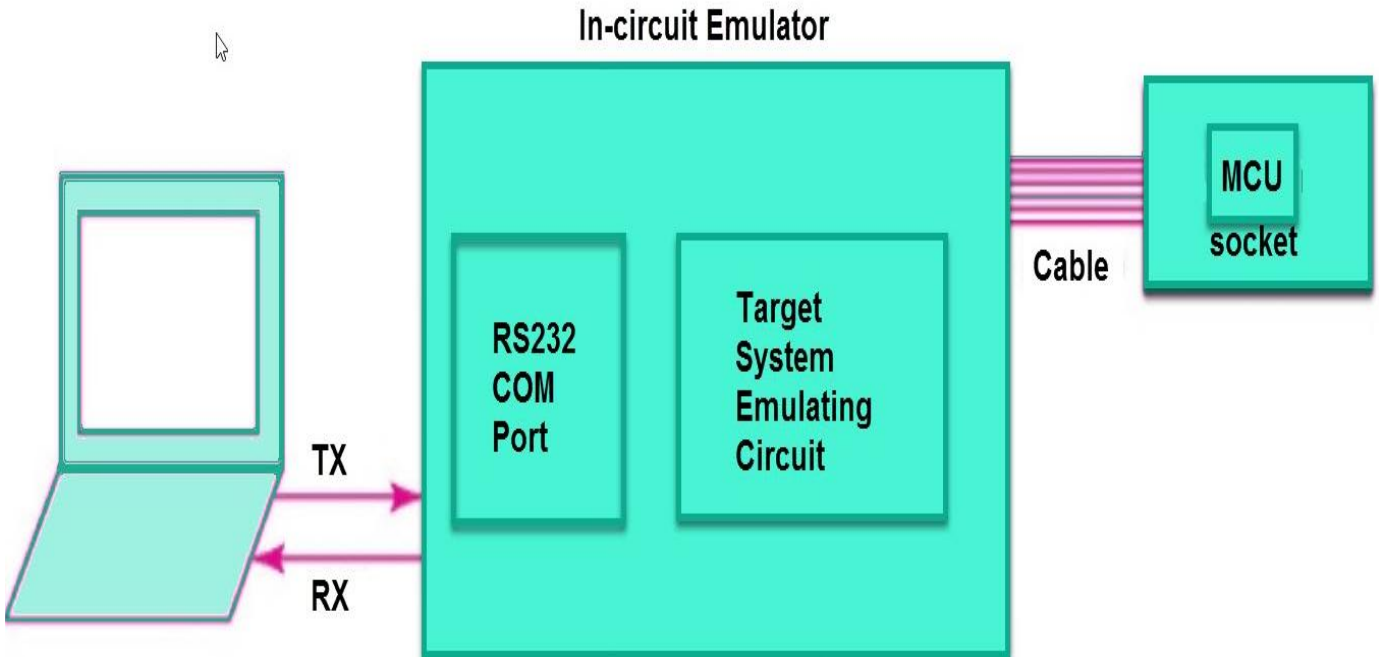
IN-CIRCUIT EMULATOR

- 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 PROCESSORS

ARM instruction set

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

ARM versions

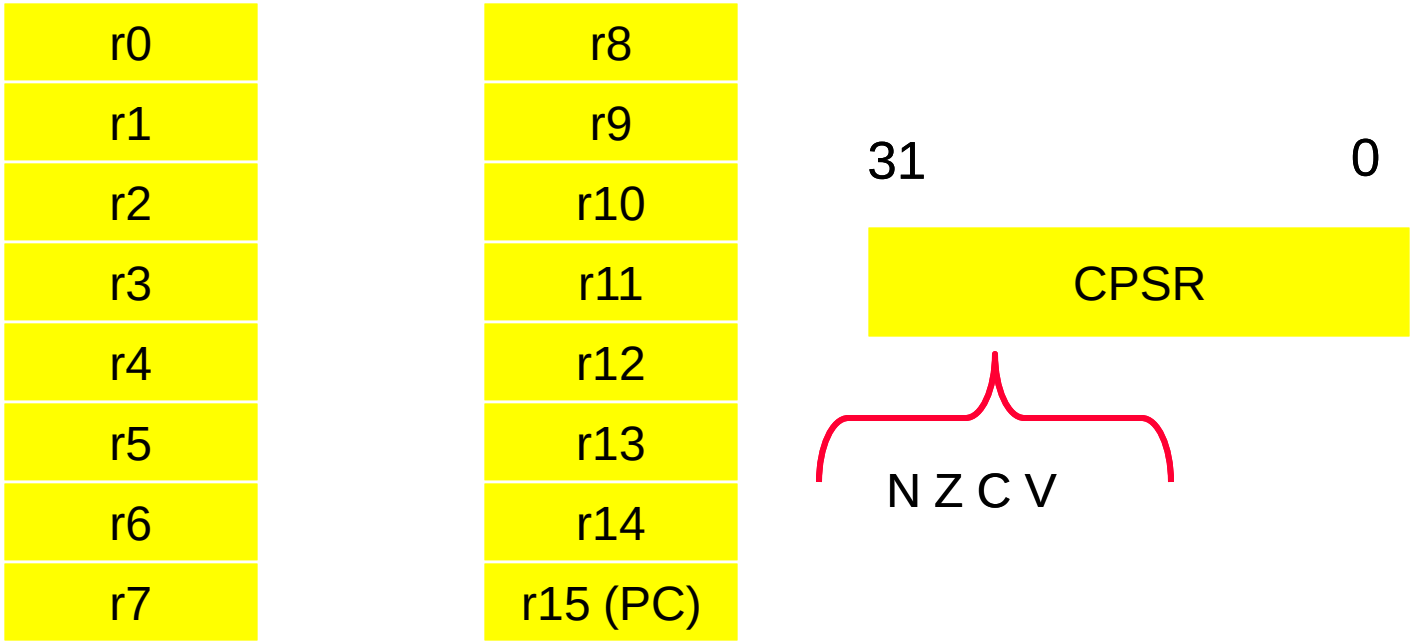
- ARM architecture has been extended over several versions.
- We will concentrate on ARM7.

ARM assembly language

- **Fairly standard assembly language:**

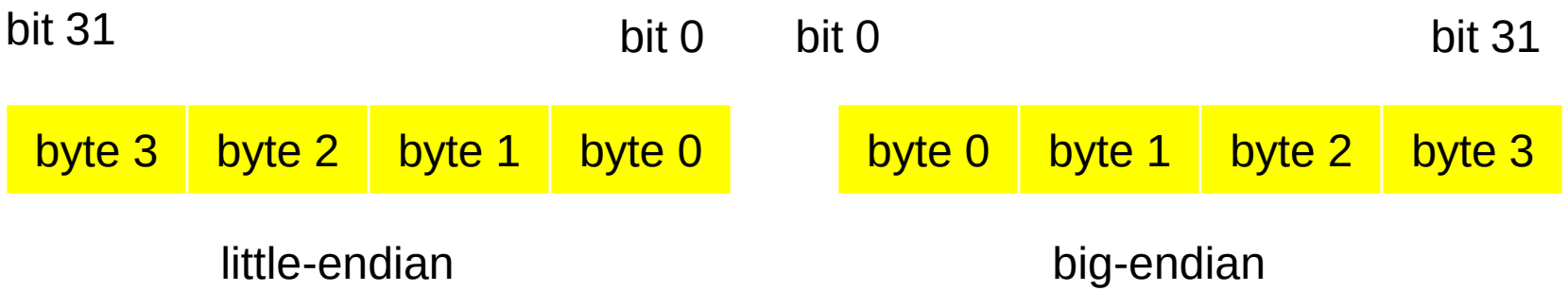
```
LDR r0,[r8] ; a comment
label   ADD r4,r0,r1
```

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 address for d
    LDR r1,[r4] ; get value for d
    SUB r0,r0,r1 ; compute a-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

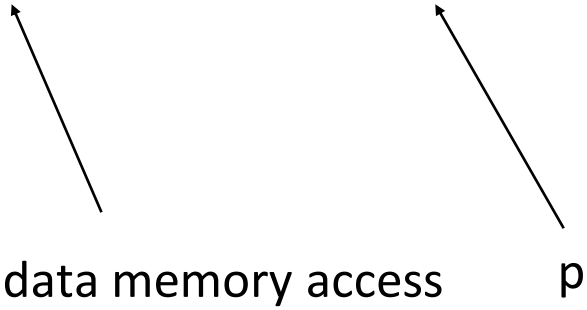
- **Register files:**

  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:

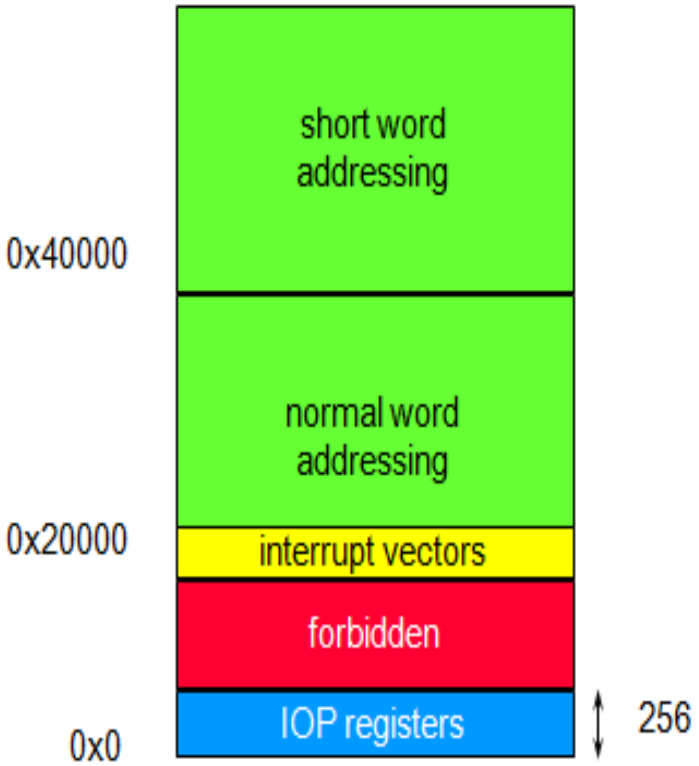
```
R1=DM(M0,I0), R2=PM(M8,I8); ! comment
label: R3=R1+R2;
```



data memory access

program memory access

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:

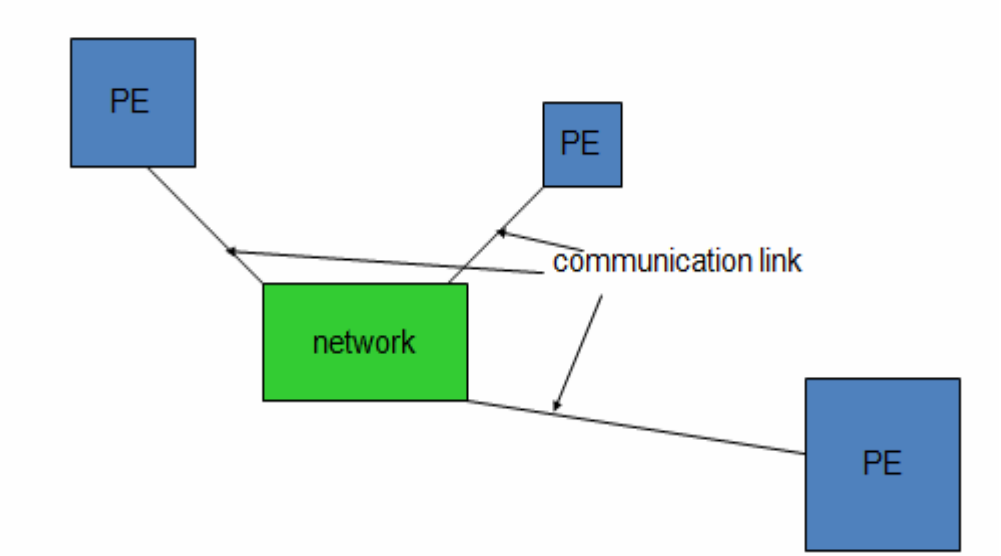
- instruction cache;
- PC stack;
- status registers;
- loop logic;
- data address generator;

Networking for Embedded Systems

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

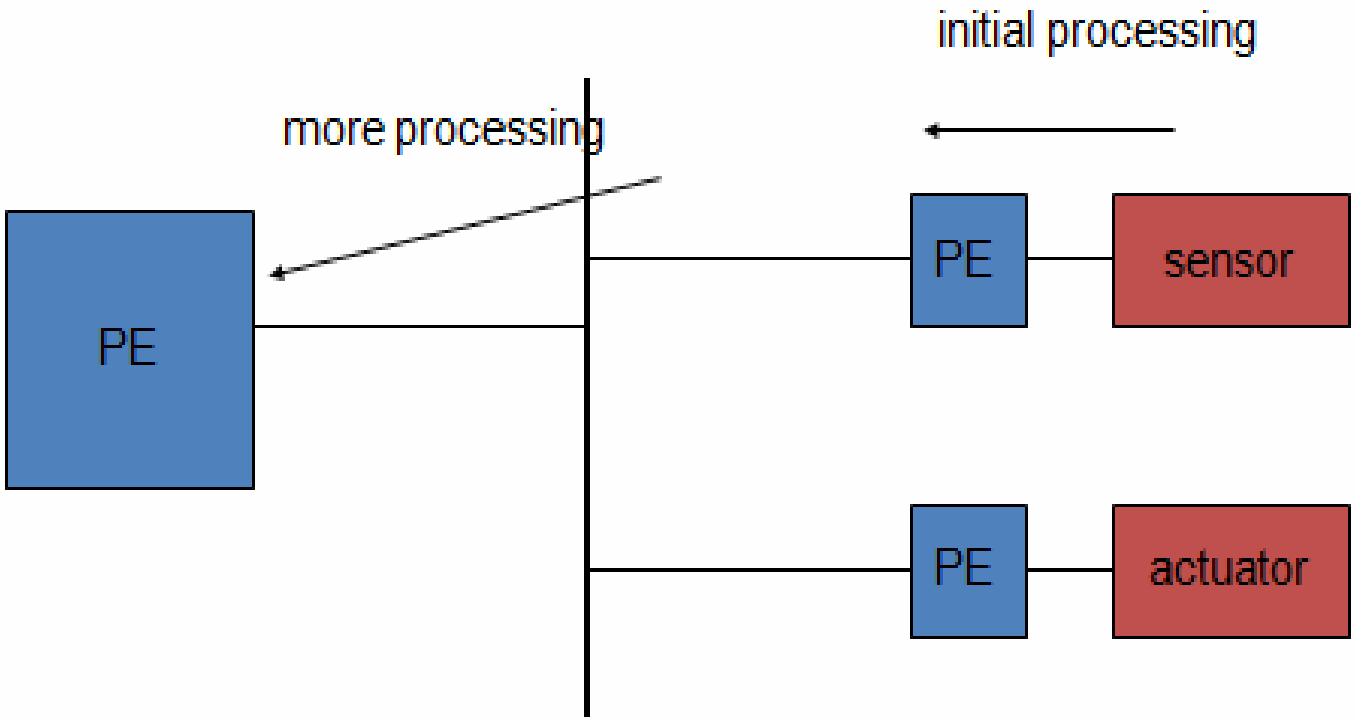
Network elements

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.

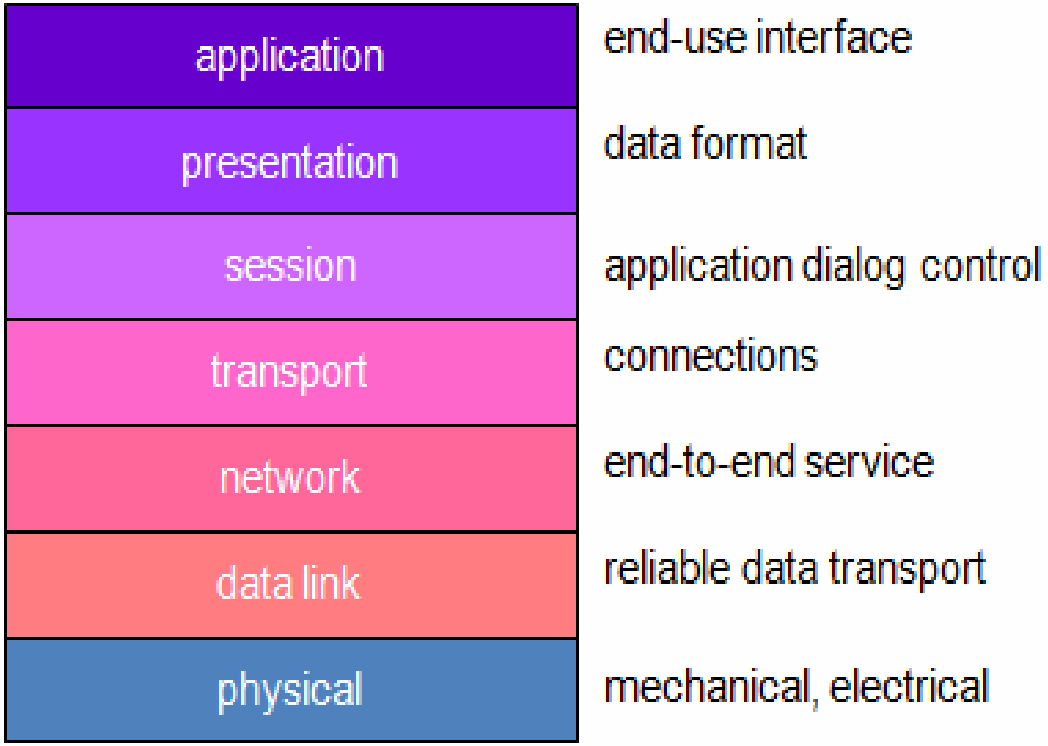
Network abstractions

- 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

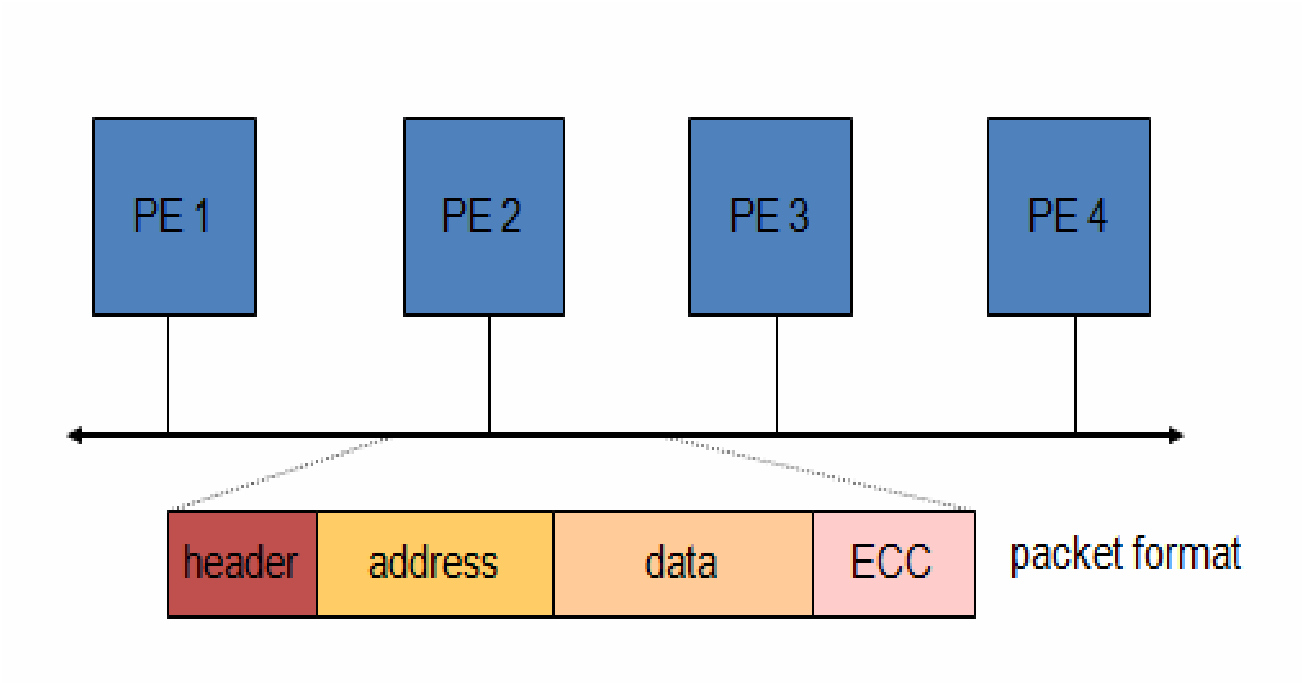

application
presentation
session
transport
network
data link
physical
end-use interface
data format
application dialog control
connections
end-to-end service
reliable data transport
mechanical, electrical

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.

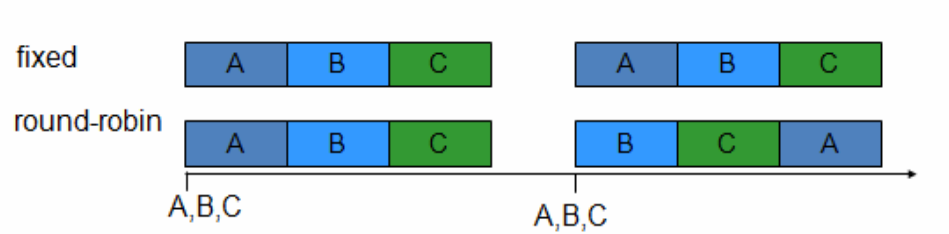
Bus networks

- Common physical connection:

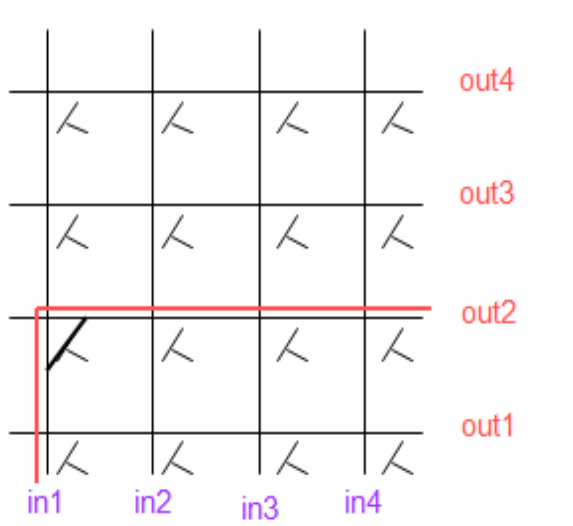


Bus arbitration

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



Crossbar



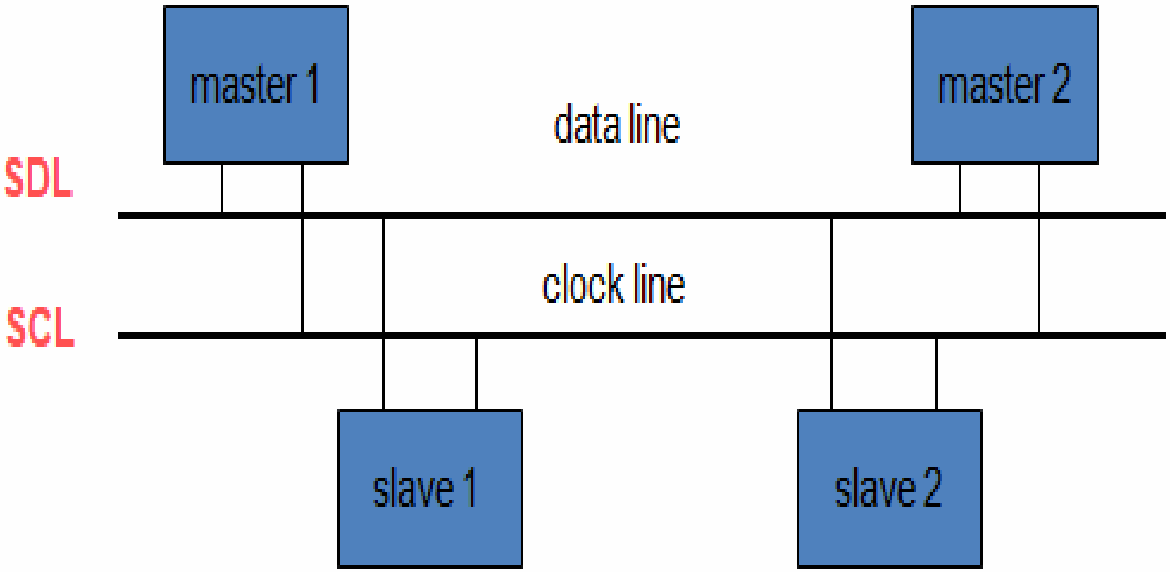
Crossbar characteristics:

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

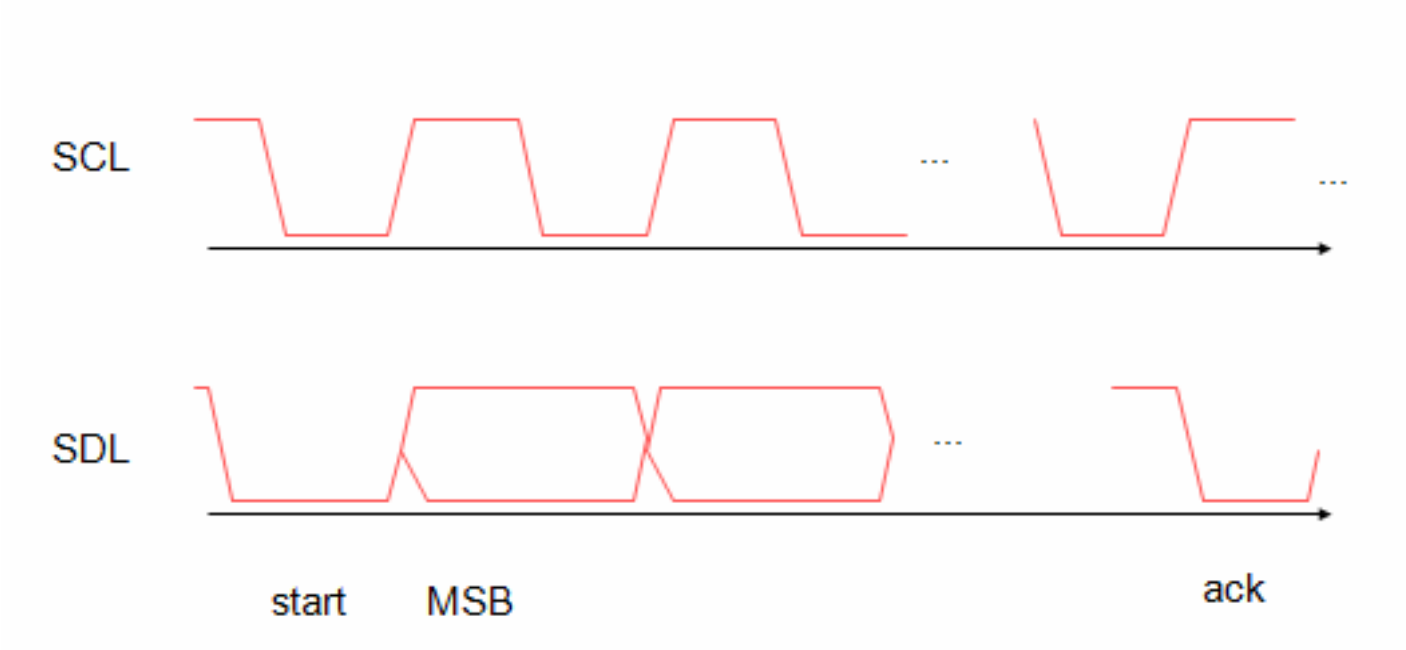
I2C bus

- 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.

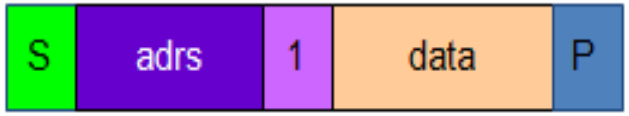
I2C bus arbitration

- 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



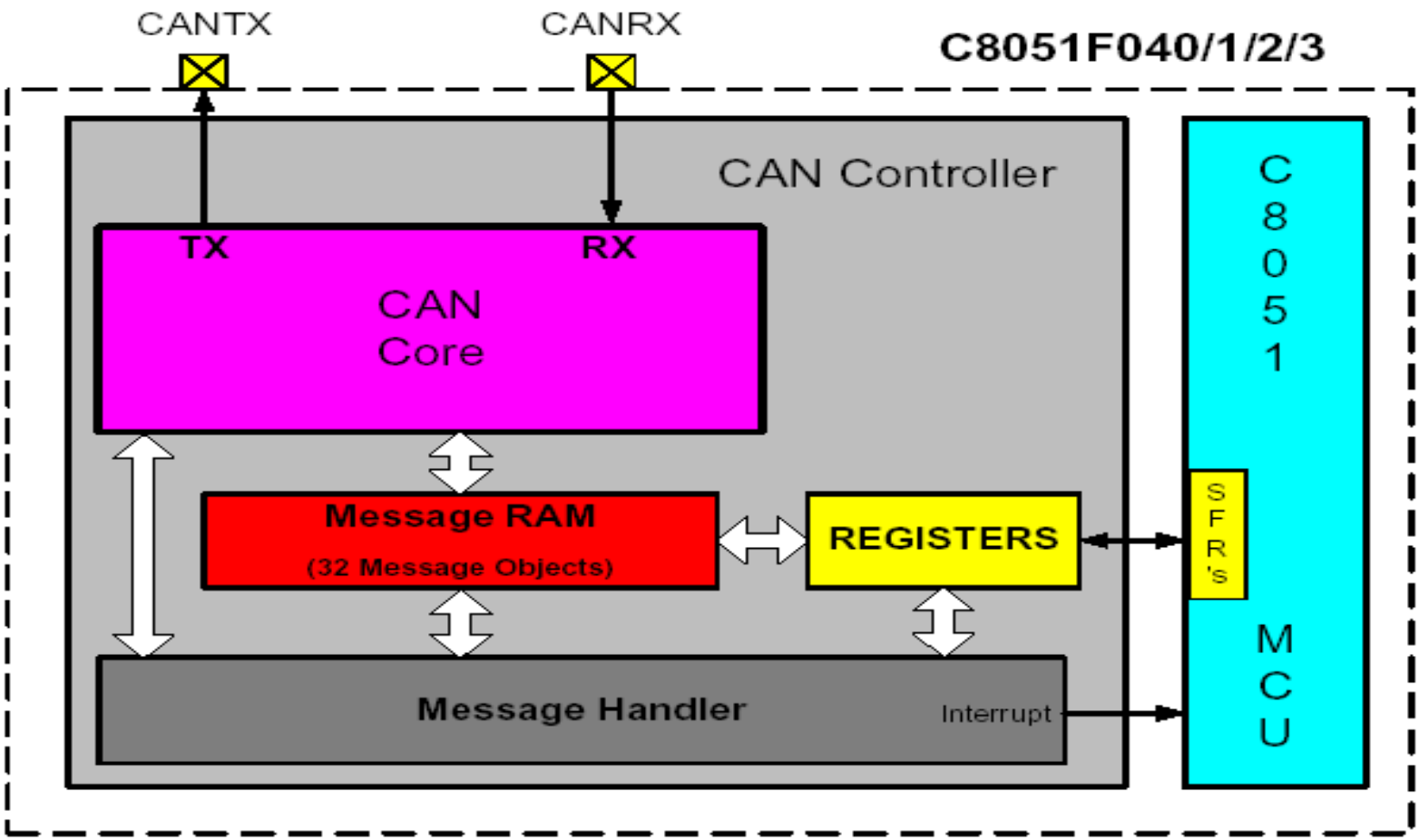
write, then read

CAN BUS

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

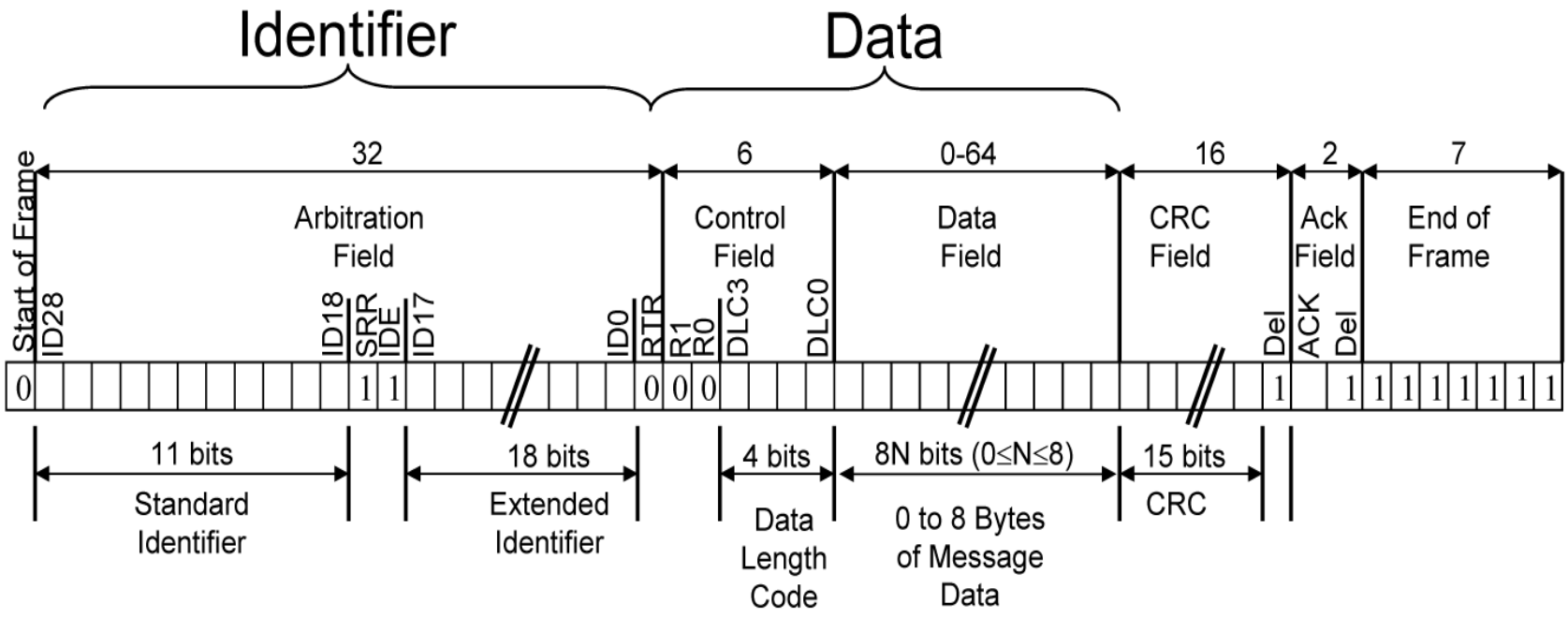
- It was originally designed for use in the automotive industry but is used today in many other systems (e.g. home appliances and industrial machines).

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.

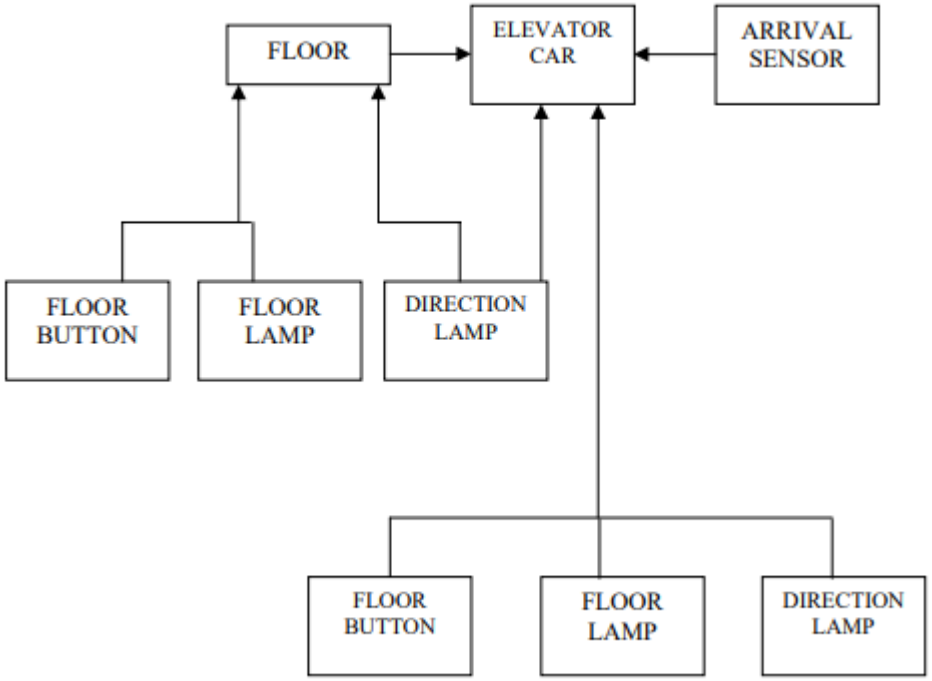
1.High end laser printers use internet protocols to receive print jobs from host machines.

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