Real-Time Systems

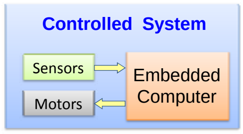
Giorgio Buttazzo
E-mail: buttazzo@sssup.it
Real-Time Systems Laboratory
Scuola Superiore Sant'Anna

http://retis.sssup.it/~giorgio/rts-LE.html

Definition

Real-Time Systems are computing systems that must perform computation within given timing constraints.

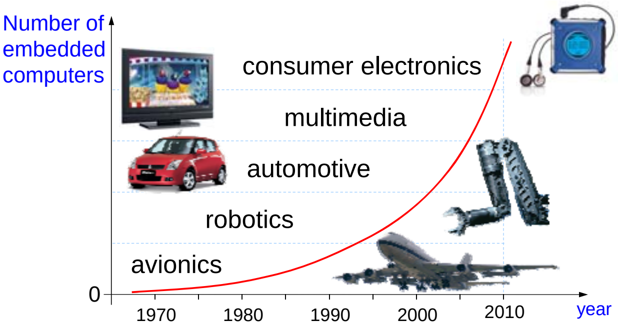
They are typically embedded in a larger system to control its functions:



Real-Time Embedded Systems

Evolution of Embedded Systems

Embedded computing systems have grown exponentially in several application domains:

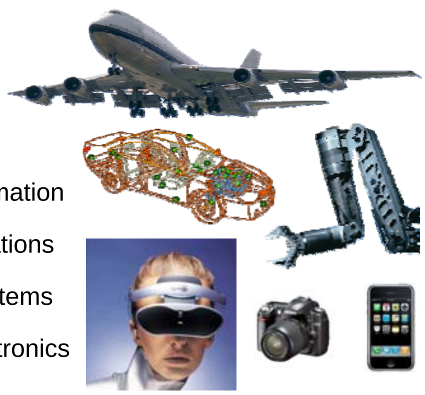


Computers everywhere

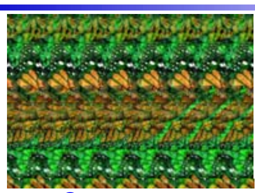
Today, **98%** of all processors in the planet are embedded in other objects:

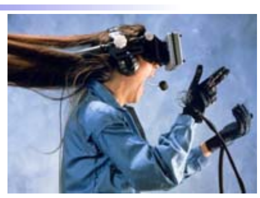
Typical applications

- avionics
- automotive
- robotics
- industrial automation
- telecommunications
- multimedia systems
- consumer electronics



Art & Entertainment


Stereograms

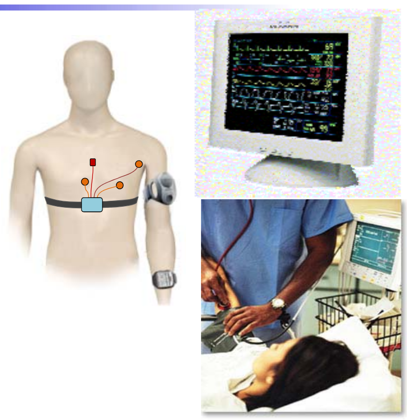

Virtual games


Animation

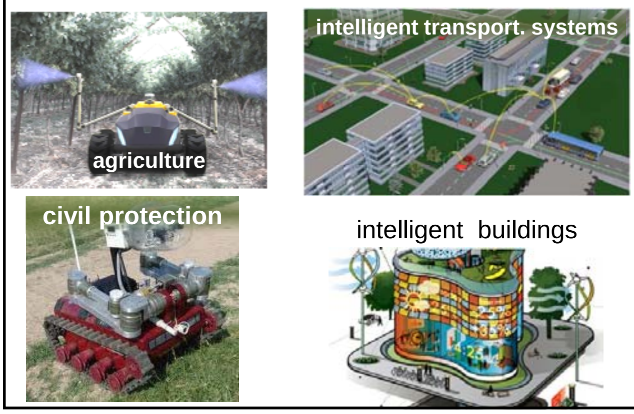

Smart toys

Health Care

- Tele-monitoring
- Tele-rehabilitation
- Assisted Living
- Sport



Emerging applications



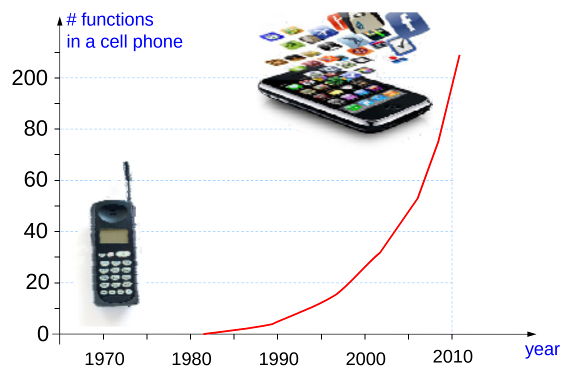
Smart objects

The number of such objects will increase in the future:

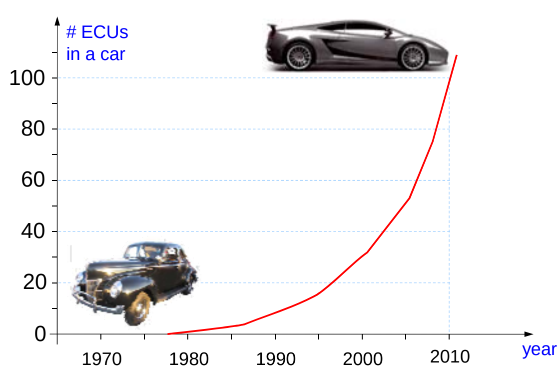
Inside body

Computers will be embedded even in our body:

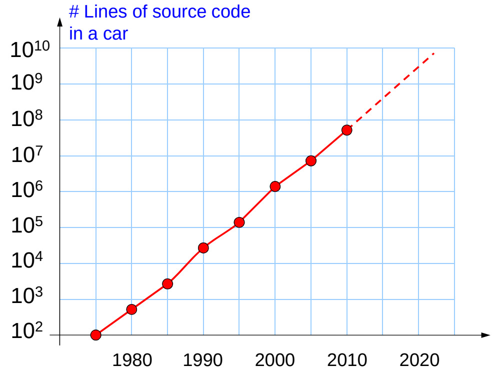
Increasing complexity



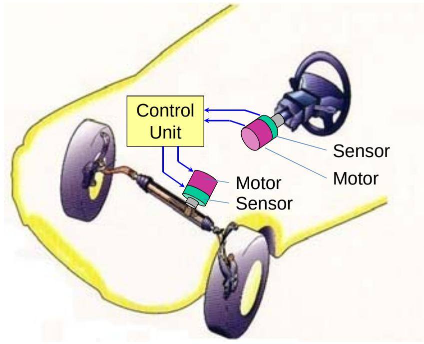
ECU growth in a car



Software evolution in a car



Steer by Wire

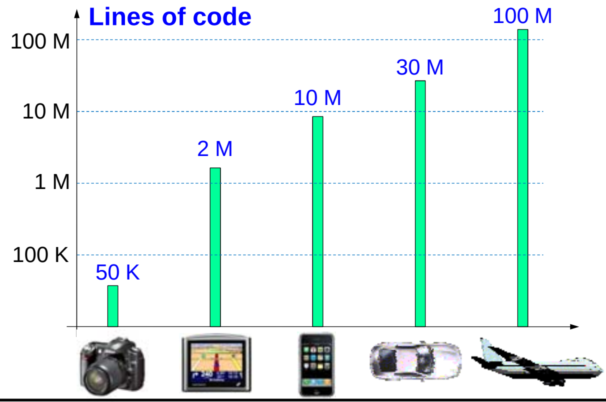


Software in a car

Car software controls almost everything:

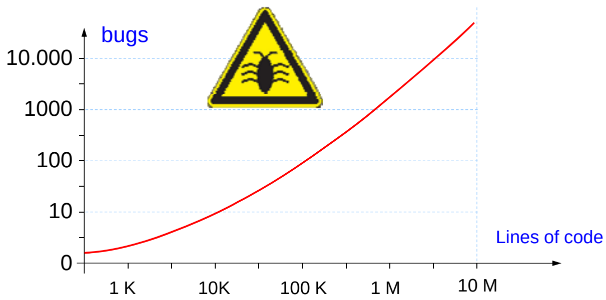
- Engine: ignition, fuel pressure, water temperature, valve control, gear control,
- Dashboard: engine status, message display, alarms
- Diagnostic: failure signaling and prediction
- Safety: ABS, ESC, EAL, CBC, TCS
- Assistance: power steering, navigation, sleep sensors, parking, night vision, collision detection
- Comfort: fan control, heating, air conditioning, music, active light control, noise control & cancellation, regulations: steer/lights/sits/mirrors/glasses…

Comparing Software Complexity



Complexity and bugs

Software bugs increase with complexity:

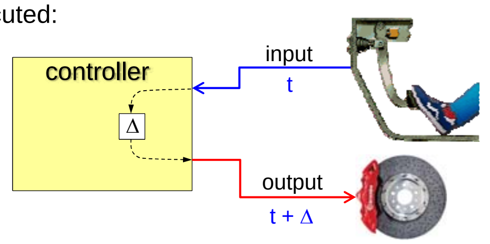


Software reliability

When aircraft control depends on a program with 100 million instructions, reliability is a primary objective.

Software reliability

Reliability does not only depend on the correctness of single instructions, but also on when they are executed:



A correct action executed too late can be useless or even dangerous.

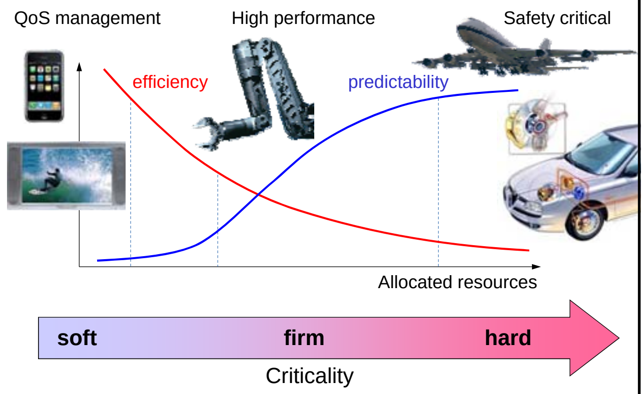
Real-Time Systems

Computing systems that must guarantee bounded and predictable response times are called **real-time systems**.

Predictability of response times must be guaranteed

- for each critical activity;
- for all possible combination of events.

Predictability vs. Efficiency



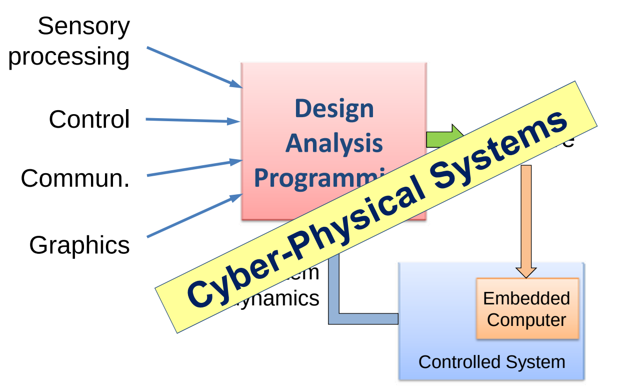
What's special in Embedded Systems?

FEATURES		REQUIREMENTS
Scarce resources (space, weight, time, memory, energy)	➡	**High efficiency** in resource management
High concurrency and resource sharing (high task interference)	➡	**Temporal isolation** to limit the interference
Interaction with the environment (causing timing constraints)	➡	**High predictability** in the response time
High variability on workload and resource demand	➡	**Adaptivity** to handle overload situations

Aim of the Course

- Studying software methodologies for supporting **time critical** computing systems.
- We will not consider how to control a system, but only how to provide a predictable software support to control applications.

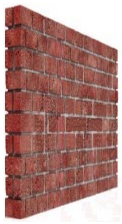
Main focus: predictable software



Control and implementation

Often, control and implementation are done by different people that do not talk to each other:

$\dot{x} = Ax + Bu$



```
if (b != 0) y = a/b;
else printf("error\n");
```

Control guys typically assume a computer with infinite resources and computational power. In some case, computation is modeled by a fixed delay Δ.

Control and implementation

In reality, a computer:

- has limited resources;
- finite computational power (non null execution times);
- executes several concurrent activities;
- introduces variabile delays (often unpredictable).

Modeling such factors and taking them into account in the design phase allows a significant improvement in performance and reliability.

Specific course objectives

- Study software methodologies and algorithms to increase predictability in computing systems.
- We consider embeddded computing systems consisting of several concurrent activities subject to timing constratints.
- We will see how to model and analyze a real-time application to predict worst-case response times and verify its feasibility under a set of constraints.

Specific course objectives

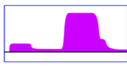
How to model a real-time system

How to combine control & programming

How to analyze timing properties

How to monitor real-time applications

How to program safety-critical code



How to manage load variations

Course outline - 1

1. Basic concepts and terminology
2. Sample applications
3. Problem identification
4. Modeling real-time activities
5. Deriving timing constraints
6. Worst-case reasoning
7. Managing periodic tasks
8. Scheduling algorithms
9. Schedulability analysis

Course outline - 2

10. Problems introduced by resource sharing
11. Resource access protocols
12. Estimating worst-case blocking times
13. Handling asynchronous (aperiodic) tasks
14. Handling execution overruns
15. Managing overload conditions
16. Real-time communication mechanisms

Course outline - 3

Programming real-time applications

- Processes and threads in Linux
- Thread creation and activation
- Linux schedulers
- Time management
- How implement periodic threads
- How to structure RT applications
- How to use a graphics library
- How to simulate RT control systems

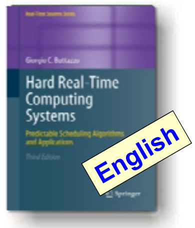
Pthread library

Teaching material

Course homepage

http://retis.sssup.it/~giorgio/rts-MECS.html

Books:



Third Edition
Springer, 2011

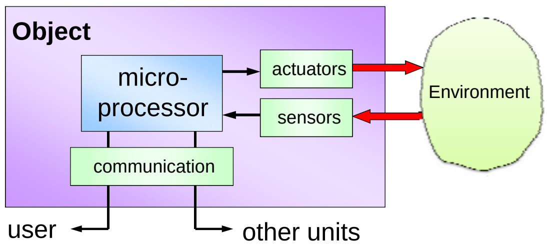
Third Edition
Pitagora, 2006

32

Definitions and sample applications

Embedded systems

They are computing systems hidden in an object to control its functions, enhance its performance, manage the available resources and simplify the interaction with the user.

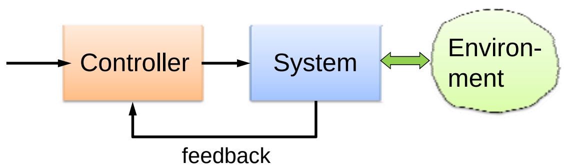


Control system components

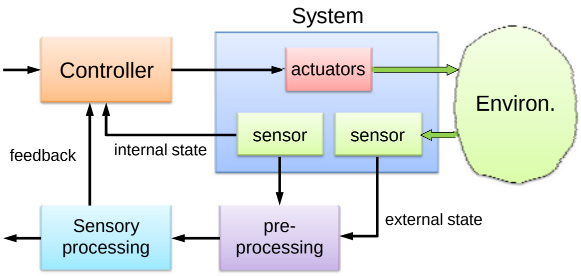
In every control application, we can distinguish 3 basic components:

- the **system** to be controlled
  - it may include sensors and actuators
- the **controller**
  - it sends signals to the system according to a predetermined control objective
- the **environment** in which the system operates

A typical control system



Detailed block diagram



Other activities

filtering, classification, data fusion, recognition, planning

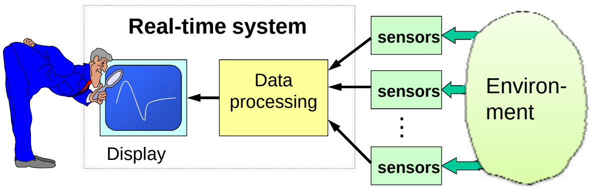
Types of control systems

Depending of the system-environment interactions, we can distinguish 3 types of control systems:

- **Monitoring Systems**
  - do not modify the environment
- **Open-loop control systems**
  - loosely modify the environment
- **Closed-loop control systems**
  - tight interaction between perception and action

Monitoring Systems

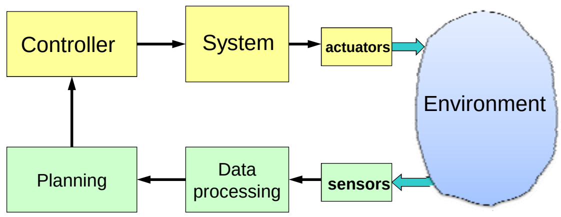
- Do not have actuators
- Do not modify the environment



Examples: Environmental monitoring, surveillance systems, air traffic control

Loosely-coupled control systems

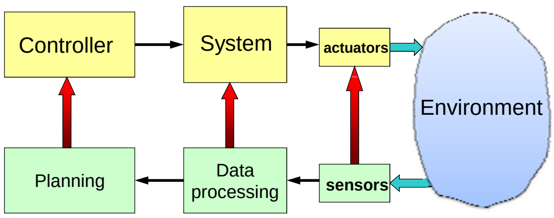
Modify the environment, actions are mostly pre-programmed, so loosely coupled with the current state of the environment:



Examples: painting robots, assembly robots, sorting robots

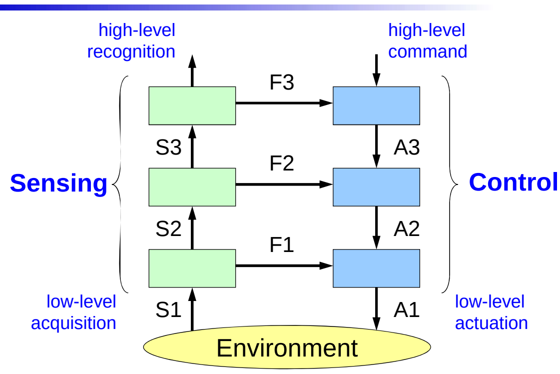
Tightly-coupled control systems

Sensing and control are tightly coupled and occur at different hierarchical level:



Examples: flight control systems, military systems, advanced robots, living beings

Hierarchical control



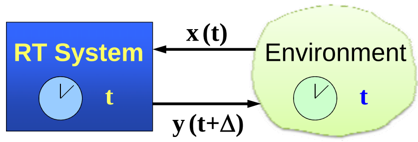
Implications

- The tight interaction with the environment requires the system to react to events within precise timing constraints.
- Timing constraints are imposed by the performance requirements and the dynamics of the system to be controlled.

➡ The operating system must be able to execute tasks within timing constraints.

Real-Time System

It is a system in which the correctness depends not only on the output values, but also on the **time** at which results are produced.



REAL means that system time must be synchronized with the time flowing in the environment.

Typical objection

It is not worth to invest in RT theory, because computer speed is increasing exponentially, and all timing constraints can eventually be handled.

Answer

Given an arbitrary computer speed, we must always guarantee that timing constraints can be met. Testing is **NOT** sufficient.

Real-Time ≠ Fast

- A real-time system is **not** a fast system.
- Speed is always relative to a specific environment.
- Running faster is good, but does not guarantee a correct behavior.

Speed vs. Predictability

- The objective of a real-time system is to guarantee the timing behavior of each individual task.
- The objective of a fast system is to minimize the average response time of a task set. But …

Don't trust the average when you have to guarantee individual performance

Sources of non determinism

- **Architecture**
  - cache, pipelining, interrupts, DMA
- **Operating system**
  - scheduling, synchronization, communication
- **Language**
  - lack of explicit support for time
- **Design methodologies**
  - lack of analysis and verification techniques

Traditional (wrong) approach

In spite of this large application domain, most of RT applications are designed using empirical techniques:

- assembly programming
- timing through dedicated timers
- control through driver programming
- priority manipulation

Disadvantages

1. Tedious programming which heavily depends on programmer's ability
2. Difficult code understanding

$$\text{Readability} \propto \frac{1}{\text{efficiency}}$$

Disadvantages

3. Difficult software maintainability
   - Complex appl.s consists of millions lines of code
   - Code understanding takes more that re-writing
   - But re-writing is VERY expensive and bug prone
4. Difficult to verify timing constraints without explicit support from the OS and the language

Implications

- Such a way of programming RT applications is very dangerous.
- It may work in most situations, but the risk of a failure is high.
- When the system fails is very difficult to understand why.

→ **low reliability**

Accidents due to SW

- Task overrun during LEM lunar landing
- First flight of the Space Shuttle (synch)
- Ariane 5 (overflow)
- Airbus 320 (cart task)
- Airbus 320 (holding task)
- Pathfinder (reset for timeout)

Lessons learned

- Tests, although necessary, allow only a partial verification of system's behavior.
- Predictability must be improved at the level of the operating system.
- The system must be designed to be fault-tolerant and handle overload conditions.
- Critical systems must be designed under pessimistic assumptions.