Embedded Systems:
Characteristics and Constraints

Pascal Raymond
Verimag-CNRS

MOSIG - Embedded Systems

What is an Embedded System?

Minimal definition

A computer *system* dedicated to a *particular function*

- *system:* mix of software/middleware/hardware
- *particular function:* not really/fully programmable, does 1 thing

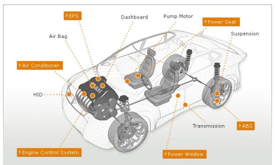
Related/similar notions

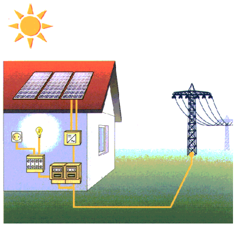
More or less “synonyms” emphasize particular characteristic(s):

- Reactive systems: perform “everlasting” interaction with the environment
- Real-Time systems: must react “instantaneously” with respect of a particular environment
- Cyber-physical: emphasize the difference between the system (digital, discrete time) and its environment (physical, continuous time)
- ... and historically, strongly related to *control engineering systems*.

Computer Systems in Everyday-Life Objects

- Trains, subways, cars ...
- Avionics and space
- Consumer electronics (phones, digital cameras, ...)
- Smart cards
- Household appliances
- Telecom equipments
- Computer Assisted Surgery
- Smart buildings and Energy





Characteristics

Various problems/difficulties ...

- Real time: system must be fast enough to react to prhysics
- Criticity: safety-critical and/or business critical
- Limited resources: memory, processor, energy, space

... whose importance depends on the particular domain. What are the main problems for these domains ?

- Embedded control (train, cars, planes, power plants) ?
- Consumer Electronics ?
- Sensor Networks ?

Let's focus on embedded control ...

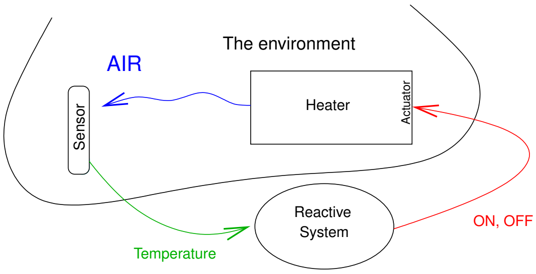
Detailed Example: Embedded Control

- In trains, cars, aircraft, space objects ...
- and also power plants, elevators etc.

Characteristics

- The environment is *mainly physical*, with more or less human intervention
- Submitted to strong *real-time constraints* (often called: hard-real time)
- They are *safety-critical* systems
- The computer system is the implementation of a *control engineering* solution
- The computer system is *reactive*

A (tiny) example: heater control



```
initializations
while (true) {
  --- point (1)
  get inputs
    from the sensors
  compute outputs
    and update memory
  write outputs
    on the actuators
  --- point (2)
}
```

- Real-time: time to execute the code from (1) to (2) must be short enough
- Reactive: output to the environment influence future inputs
- Criticity/Safety: badly controlled outputs may have dramatic consequences e.g., this a (small) part of a nuclear power-plant controller

Real-Time Programming Problems

- Write code that is sufficiently fast (not always possible to "try a faster machine")
- Be able to tell how fast your program is, *in advance* (Worst-Case-Execution-Time static evaluation)
- It's not always possible to write single-loop code, because of the *intrinsic parallelism* of a reactive system.

  e.g., multiple sensor-computing-actuator lines, like temperature and pressure

---

Safety Problems

- Criticity: faults may be irreparable (lives, environment), or just very expensive (e.g., launcher, Ariane 5 flight 501 1996)
- HW failures:
  - ↪ HW Fault-tolerance: a whole domain, mainly based on redundancy (e.g., several sensors + voter, several processors running the same code) necessary since HW may break down.
- SW "failures":
  - ↪ a SW does not "break down", it is buggy
  - ↪ (run-time) SW Fault-tolerance ?

    e.g, several codes developed independently from the same specification...
  - ↪ (off-line) classical methods to track bugs: programming methodology; intensive testing; when possible: formal verification

Safety and certification

- Critical ES are submitted to Design norms, defined by certification authorities.
- Example in civil avionics: DO178B
  - ↪ a bundle of definitions and rules
  - ↪ classify risks form “none” to “catastrophic”
  - ↪ recommends/imposes design/validation methods depending on the level
  - ↪ basically: an aircraft whose ES is not DO178B certified is not allowed to fly
- Other examples:
  - ↪ Railways: IEC 62279
  - ↪ Automotive: ISO 26262 (concerns safety in general, including SW)

Centralized or Distributed Systems?

- Fact: *centralized systems are far simpler than distributed ones*
- However, distribution is required:
  - ↪ HW fault-tolerance
  - ↪ topology of sensors/actuator lines (several dozens in an aircraft)
  - ↪ sometimes, for efficiency purpose.
- Other fact: *distributed real-time* programming is very hard

Designing Embedded System

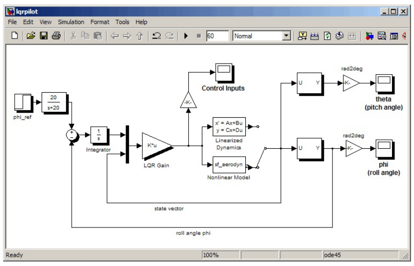
Main Difficulties for the Design of Embedded Systems

- Real-time parallel and distributed programming (choice of a programming language?)
- Relation with control engineering
- Intricate dependency between HW, application SW, and OS or middleware
- Certification authorities
- Several degrees of dynamicity (from simple reconfigurations to mobile code...)

Industrial Practice

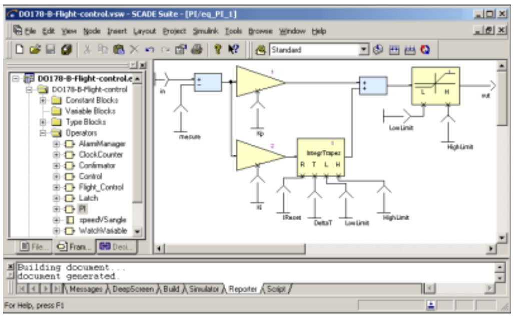
- “Hand-craft”: use general purpose tools and languages (Java/C/assembly...)
- Domain Specific Languages: real-time features (multi-tasks, timers, synchro), specific device operation (sensor/actuator libraries)
- Model-based design: more radical, continuity between high level design (control engineering problem) and implementation

Example: Simulink



- Originally: design/simulation tool for control engineers
- Allows to simulate conjointly:
  - ↪ a continuous-time model of the environment
  - ↪ a discrete-time solution of the controller
- Implementing the controller:
  - ↪ classically manual encoding (Simulink = specification language)
  - ↪ more and more: automat-ic/ized code generation (Simulink = programming language)
- Widely used in automotive, transportation, and all domains where control engineering culture is strong.

Example: Scade



```
Building document...
document generated.
```

- Close to control engineering concepts (block-diagram)
- Formal language, deterministic specification
- Programming language/environment:
  - ↪ Software engineering features (modularity, libraries)
  - ↪ Automatic code generation:
    KCG compiler is DO178B qualified
    in particular, eliminates the need of low-level code testing
  - ↪ Thanks to formal semantics, high-level validation possible (and sufficient cf. qualif.): automated testing, formal verification
- Widely used (imposed) for high-critical systems (avionics, helicopters, power plants)

Summary

Important points/problems

- General purpose vs dedicated languages
- Validity/correctness: functional (no bugs), extra-functional (time)

This course

- Focus on functionality
- How to design/validate safe ES:
  - ↪ programming languages (features? styles?)
  - ↪ code generation
  - ↪ functional validation (formal methods?)
  - ↪ timing validation (Worst Case Exec time ?)
- Based on the so-called “Synchronous approach”