Design of Embedded Systems

José Costa

Software for Embedded Systems

Departamento de Engenharia Informática (DEI)
Instituto Superior Técnico

2013-02-24
Outline

- Challenges in Embedded Systems Design

- The Embedded System Design Process
  - Requirements
  - Specification
  - Architecture
  - Component Design
  - System Integration

- Formalisms for System Design
  - Unified Modeling Language
Challenges in Embedded Systems Design

- How much **hardware** do we need?
- How do we meet **deadlines**?
- How do we minimize **power** consumption?
- How do we design for **upgradeability**?
- Does it really **work**?
Difficulties in Design and Development

- Complex testing
  - run a real machine to have proper data
  - system must be tested in the embedded machine

- Limited observability and controllability
  - sometimes no keyboard or screen!
  - in real-time systems it’s not easy to stop the system to see what is going on

- Restricted development environments
  - much more limited than in PCs
  - usually compile code in PC and download it to embedded system
Design Methodologies

A design methodology is a procedure for designing a system.

- Understanding your methodology helps you ensure you didn’t skip anything.
- Compilers, software engineering tools, computer-aided design (CAD) tools, etc., can be used to
  - help automate methodology steps
  - keep track of the methodology itself
- Better communication between team members
  - what they are supposed to do
  - what they should receive
  - when they have completed their assigned steps
Levels of Abstraction

- Requirements
- Specification
- Architecture
- Component Design
- System Integration
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
Design Goals

- Performance
  - Overall speed, deadlines
- Functionality and user interface
- Manufacturing cost
- Power consumption
- Other requirements (physical size, etc.)
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
- **verify** that it meets all system goals
  - cost, speed, ...
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

- Requirements end up being the specification
  - containing enough information to begin designing the system architecture
Requirements vs Specification

Consumers:
- are not embedded system designers
- see mostly users’ interactions
- most of the time have unrealistic expectations as to what can be done within their budgets
- have a different language

Translating from requirements to specification (from the consumer’s language to the designer’s)
- capturing a consistent set of requirements from the customer
- massaging those requirements into a more formal specification
Functional vs. Non-functional Requirements

- Functional requirements
  - output as a function of input

- Non-functional requirements
  - time required to compute output
  - cost
  - size, weight, etc.
  - power consumption
  - reliability
  - ...

José Costa (DEI/IST)
Non-functional Requirements

- **Performance**
  - major consideration for the usability of the system and its ultimate cost
  - may be a combination of soft performance metrics and hard deadlines

- **Cost**
  - manufacturing costs (e.g. components, assembly)
  - nonrecurring engineering (NRE) costs (e.g. personnel, designing the system)

- **Physical size and weight**
  - depends on the application

- **Power consumption**
  - important not only in battery-powered systems
  - specified in terms of battery life
Validating The Requirements

- Requires understanding what people want and how they communicate it

- User interface requirements can be refined by using a mock-up
  - may be executed on a PC

- Physical, nonfunctional models of devices can also help
  - better idea of size and weight
Sample Requirements Form (1/3)

- name
- purpose
- inputs and outputs
- functions
- performance
- manufacturing costs
- power
- physical size/weight
Sample Requirements Form (2/3)

- **name**
- **purpose**
  - one- or two-line description
- **inputs and outputs**
  - types of data: analog? digital? mechanical?...
  - data characteristics: periodic? occasional? how many bits?...
  - types of I/O devices: buttons? A/D converters? video displays?...
- **functions**
  - more detailed description of the system
  - when the system receives an input, what does it do?
  - how do interface inputs affect these functions?
  - how do different functions interact?
Sample Requirements Form (3/3)

- **performance**
  - must be identified earlier to ensure that the system works properly

- **manufacturing costs**
  - cost has substantial influence on architecture
  - work with some idea of the cost range

- **power**
  - battery powered? plugged into a wall?

- **physical size/weight**
  - more or less flexibility in the components to use
Beyond The Requirements Form

- The requirements form should be the introductory of a longer document.
- After writing the requirements you should check for internal inconsistency:
  - forget to assign functions to an input/output?
  - considered all modes of operation?
  - unrealistic number of features into a battery-powered, low-cost machine?

## Requirements form

- name
- purpose
- inputs and outputs
- functions
- performance
- manufacturing costs
- power
- physical size/weight
Example: GPS Moving Map Requirements

- Moving map obtains position from GPS
- Paints map from local database

I-78
Scotch Road
lat: 40 13 lon: 32 19
GPS Moving Map Needs (1/2)

- **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
GPS Moving Map Needs (2/2)

- **Cost**
  - € 100 street price = approx. € 30 cost of goods sold

- **Physical size/weight**
  - should fit in dashboard

- **Power consumption**
  - current draw comparable to CD player
name: GPS moving map

purpose: consumer-grade moving map for driving

inputs: power button, two control buttons

outputs: back-lit LCD 400x600

functions: 5-receiver GPS; displays current lat/lon

performance: updates screen within 0.25 sec of movement

manufacturing costs: \(\approx 30\€\) cost-of-goods-sold

power: 100mW

physical size/weight: no more than 5cm \(\times\) 15cm, 350g
Specification

Serves as the contract between the customer and the architects.

- 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
Specification

- Should be understandable enough
  - so that someone can verify that it meets system requirements and overall expectations of the customer

- Should be unambiguous

Problems of unclear specifications

- implementation of wrong functionality
- system architecture may be inadequate to meet the needs of the implementation
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
  - e.g. operating the GPS receiver
Specification vs Architecture Design

- The specification does not say how the system does things, only what the system does.

- The purpose of the architecture is to describe how the system implements the functions.

- The architecture is a plan for the overall structure of the system, it will be used later to design the components.

The creation of the architecture is the 1st phase of the so called “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 receiver
- search engine
- renderer
- display
- database
- user interface
GPS Moving Map Hardware Architecture
GPS Moving Map Software Architecture

position → database search → renderer → pixels

user interface ← database search

timer ← renderer
Designing Hardware and Software Components

- Must spend time architecting the system before you start coding.

- Some components are ready-made
  - CPU, memory chips, ...
  - Some are software components

- Some can be modified from existing designs

- Others must be designed from scratch
  - At least you may have to design the board
  - Custom programming
Identifying Existing Hardware And Software Components

- Much of the design process is composition of existing modules
  - Hardware boards and peripheral interfaces
  - Software modules
    - Operating System or run-time environment
    - Middleware (ex. Communications layers, RFID coding)
    - Applications

- Usually designers try to keep their usual development environment
  - Processors and low-level tools
  - Operating systems or run-time environments
Put together the components (hardware blocks and software modules)

Not as easy as it sounds...  
- Many bugs appear only at this stage
- Debugging facilities are limited

Have a plan for integrating components to uncover bugs quickly, test as much functionality as early as possible
- build up the system in phases
- debug only a few modules at a time
Formalisms For 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
  - encourages design to be described as a number of interacting objects
  - some objects will correspond to real pieces of software

- Object = state + methods
  - State provides each object with its own identity
  - Methods provide an abstract interface to the object
Object Oriented Concepts

- **Objects** represent an entity and the basic building block
- **Class** is the blueprint of an object
- **Abstraction** represents the behavior of a real world entity
- **Encapsulation** is the mechanism of binding the data together and hiding them from outside world
- **Inheritance** is the mechanism of making new classes from existing one
- **Polymorphism** defines the mechanism of one class existing in different forms
UML - Unified Modeling Language

- UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

- UML was created by Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.

A picture is worth a thousand words.
Conceptual Model of UML

- Building Blocks
  - Things
  - Relationships
  - Diagrams

- Rules

- Common Mechanisms
  - Specifications
  - Adornments
  - Common Divisions
  - Extensibility Mechanisms
UML Building Blocks

**Things**
- Structural
- Behavioral
- Grouping
- Annotational

**Relationships**
- Dependency
- Association
- Generalisation
- Realization

**Diagrams**
- Class Diagram
- Object Diagram
- Use Case Diagram
- Sequence Diagram
- Collaboration Diagram
- Statechart Diagram
- Activity Diagram
- Component Diagram
- Deployment Diagram
Structural Things

Structural things are the static parts of the system.

- **Class** - an abstraction of a set of things in the problem domain that have similar properties and/or functionality

- **Interface** - a collection of operations that specify the services rendered by a class or component

- **Collaboration** - a collection of UML building blocks (classes, interfaces, relationships) that work together to provide some functionality within the system

- **Use Case** - an abstraction of a set of functions that the system performs; a use case is “realized” by a collaboration
Active Class - a class whose instance is an active object; an active object is an object that owns a process or thread (units of execution)

Component - a physical part (typically manifests itself as a piece of software) of the system

Node - a physical element that exists at run-time and represents a computational resource (typically, hardware resources)
Behavioral Things

Behavioral things are usually the dynamic parts of the system.

- **Interaction** - some behaviour constituted by messages exchanged among objects;

- **State machine** - a behaviour that specifies the sequence of “states” an object goes through, during its lifetime.
Grouping Things

Grouping things provides a higher level of abstraction.

- **Package** - a general-purpose element that comprises UML elements - structural, behavioral or even grouping things
Annotational Things

Annotational things add information/meaning to the model elements.

- **Note** - a graphical notation for attaching constraints and/or comments to elements of the model
Relationships articulates the meaning of the links between things.

- **Dependency** - a semantic relationship where a change in one thing causes a change in the semantics of the other thing
- **Association** - a structural relationship that describes the connection between two things
- **Generalisation** - a relationship between a general thing and a more specific kind of that thing, such that the latter can substitute the former
- **Realization** - a semantic relationship between two things wherein one specifies the behaviour to be carried out, and the other carries out the behaviour.
Diagrams are the graphical presentation of the model.

- **Class Diagram** - models the static view of the system

- **Object Diagram** - models the instances of things contained in a class diagram

- **Use Case Diagram** - models what the system is expected to do

- **Sequence Diagram** - models the flow of control by time-ordering
Diagrams

- **Collaboration Diagram** - models the interaction between objects, without the temporal dimension

- **Statechart Diagram** - shows the different state machines and the events that leads to each of these state machines

- **Activity Diagram** - shows the flow from activity to activity

- **Component Diagram** - shows the physical packaging of software in terms of components and the dependencies between them

- **Deployment Diagram** - shows the configuration of the processing nodes at run-time and the components that live on them
Elevator - Use Case

- Passenger
  - Process Car Calls
  - Move/Stop the Car
  - Indicate Moving Direction
  - Process Hall Calls
  - Indicate Car Position
  - Open/Close the Doors
  - Trigger Emergency Brake
Class Diagram - Object Construction

```
Class: Door
  2
  control

Class: ElevatorControl
  1

Class: Car
  1
  control

Class: Button
  *
  communicate & control

Class: CarCallButton

Class: HallCallButton

Class: Safety

Class: Indicator
  *
  control

Class: CarPositionIndicator

Class: CarLantern
```

José Costa (DEI/IST)
Modeling should capture the system in its entirety

Modeling facilitates quick and efficient analysis and design and helps communicate the overall system architecture unambiguously

Principles of modeling

- model must be chosen well
- model should encapsulate different granularities
- models can make simplifying assumptions, but not hide important facts
- no single model can capture all dimensions of the complexity
Outline

- Challenges in Embedded Systems Design

- The Embedded System Design Process
  - Requirements
  - Specification
  - Architecture
  - Component Design
  - System Integration

- Formalisms for System Design
  - Unified Modeling Language
Next Class

- Architecture of Embedded Systems