ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

# TOWARDS MODEL DRIVEN SOFTWARE DEVELOPMENT FOR ARDUINO PLATFORMS: A DSL AND AUTOMATIC CODE GENERATION

Elaborata nel corso di: Ingegneria dei Sistemi Software LM

*Tesi di Laurea di*:
MIRCO BORDONI

*Relatore*:
Prof. ANTONIO NATALI

*Co-Relatore*:
Prof ALESSANDRO RICCI

ANNO ACCADEMICO 2011-2012
SESSIONE I

# KEYWORDS

**Model Driven Software Development**

**Arduino**

**Meta-Modeling**

**Domain Specific Languages**

**Software Factory**

# Contents

# Introduction

The aim of this thesis is the production of a Meta-Model to characterize the main concepts introduced by a particular family of Embedded Systems, named Arduino.

The Meta-Model will be defined trying to capture all the concepts introduced by these kinds of Systems. The reference platform will be Arduino because it is a widely used platform to easily realize hardware systems.

Arduino provides a successful combination of good quality, low complexity and low price. In fact, with an Arduino board and some sensors or actuators everyone can create its own embedded system.

The production of a Meta-Model will lead to the implementation of a Software Factory to enhance fast prototyping and fast implementation of Arduino systems, through automatic code generation. The Meta-Model will be expressed through an EBNF that formally defines a Domain Specific Language. The framework used to define the DSL is XText. Such DSL has then been mapped on the proper Arduino code through automatic code generation. This aims at providing a programming language to write models, instance of the defined meta-model, that will be mapped automatically on an Arduino program.

The thesis goes in this direction because Meta-Modeling and DSLs are one of the main trends of Software Engineering of the last few years and they are gradually changing the way a Software is conceived and produced, speeding up its development.

Once a Meta-Model of an Embedded System is defined, the level of Abstraction has been raised trying to define some of the concepts that characterize a System of different entities, regardless from their nature. All the newly introduced concepts will describe the System Structure and the Interaction with the other components of the System, seen from each entity viewpoint.

The thesis is composed by eight chapters describing in order: what Model Driven Software Development means, an introduction on Embedded Systems, a brief description of Arduino, the defined Meta-Model for an Arduino platform, the definition of the Meta-Model related to the System perspective, a brief overview of implemented code generators, a Use Case and conclusions.

# Chapter 1

# Introduction to Model Driven Software Development

One of the main concept that an Engineer relies on is the concept of Model. A Model is always used as a representation of a known domain or system trying to capture all relevant characteristics and ignoring no relevant ones.
Since the beginning of Computer Science, research has worked to raise the level of abstraction at which softwares are written. Model Driven Development is going in this direction and is the natural continuation of that process.
In this section the concepts of Meta-Modeling and Modeling will be exploited, trying to focus on all their characteristics and on how they can help Software Engineers to solve difficult problems, to better organize their work or to increase development processes quality.
After a first theoretical introduction we will go through the definition of what is meant with Model Driven Software Development and how it is now supported by some Environments, e.g. Xtext.

## 1.1 Models and Meta-Models in Software Engineering

In Software Engineering *Models* are used to introduce the needed level of abstraction and take into account the aspects of interest of a given System.

The aspects of interest can change from case to case, depending on the level of abstraction, on what the model refers to and on which are the important characteristics of the actual abstraction.

At a first sight it can sometimes seems easy to develop a software, following a short process to develop the solution: acquiring knowledge about the problem from experts on that domain, developing a solution and deploying it to the customer. This process does not take care about many problems raised for different reasons:

- Knowledge about the problem can be ambiguous if not correctly formalized. This issue borns from interactions between humans in *natural* language. Natural language is not *formal* at all and can lead to misunderstanding and ambiguity.

- Legacy is one of the main problem in Software Engineering. In fact rarely a software is started from scratch, but it often has to cohabit, collaborate or use previous software, maybe written years before, on different technologies and from unknown developers.

- Complex Softwares are often developed by teams of people. Sometimes the team can be unique and others more than only one. This means that the work should be somehow organized and, following the approach *"Divide et Impera"*, different tasks should be distributed to different people.

- The development process is never a straightforward top-down process. It is often needed to go back in previous phases and modify what was produced before. This approach, even if not the unique, is largely used and behaves like a spiral, continuously reviewing (if needed) what previously produced.

These are the main motivations of models. It is in fact fundamental the production of proper formal artifacts in each phase of the development process and models aims at the formalisation of all the processes and artifacts involved.

So engineers rely on models to understand and formalize complex systems. They can be derived from existing systems or developed to formalize what will be produced.

A Model is defined as an instance of a *Meta-Model*. A Meta-Model is a set of abstractions introduced to produce Models and they can be introduced at different levels of a Modeling hierarchy. This means that a Model is an instance of a Meta-Model and a Meta-Model can be an instance of a Meta-Meta-Model, and so on.

Sometimes it can be useful to traduce a Model to another at an equivalent level of abstraction.

The concept of Meta-Model is well introduced in [1] where it is defined as:

*A metamodel is a model used to model modeling itself.*

The definition emphasizes the concept that a metamodel is itself a model and is used to model, so to formalize, the process of modeling itself. Previous phrase focuses also on the fact that a metamodel can be used to model itself as well as other models or meta models.

## 1.2 Model Driven Development

The introduced definitions of Models and Meta-Models have led to a Software Development process referred as *Model Driven Development*. In this approach the attention is posed on models and not on code.

The primary condition to adopt MDD is having a Meta-Metamodel enough expressive for the modeling of the main concepts introduced by the faced domain. Once that the highest Meta-Metamodel has been defined, we can proceed top-down in the definition of all the needed layers. One of the main objective of this method is the introduction of a set of formal models that can directly be mapped on code, possibly without the need of additional information in the implementation phase. To have an idea of which are the current practices of interaction and synchronization between models and code, we can have a look to a picture introduced in [2], shown in Figure 1.1. The first approach

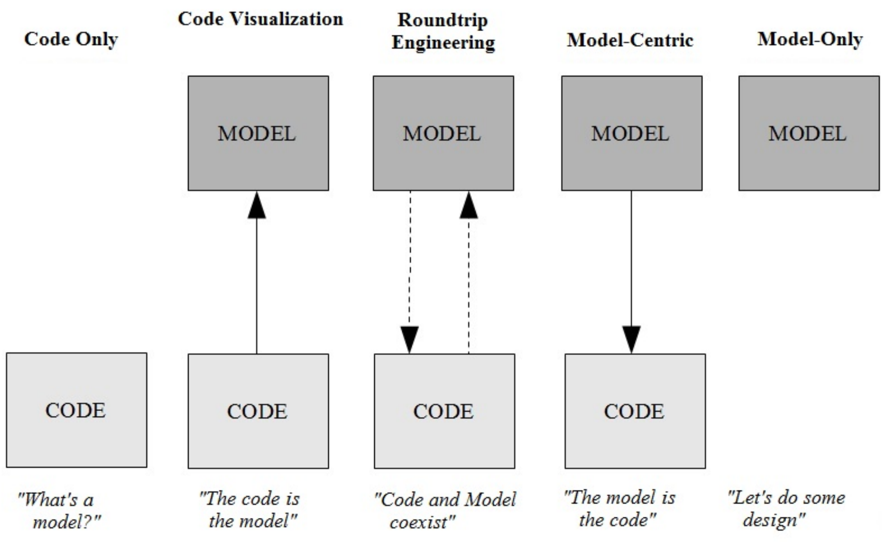


Figure 1.1: Relation between modeling and code

shown in the figure is the classical Extreme Programming approach. There is no model of the system to produce and the developers simply start by coding, each of them having a mental representation of the system and of the problem solution. It is like if the model is represented by the formally defined constructs found in the programming language used. This approach often leads to the so called *Software Crisis* because each developer can have a different idea of the solution and ambiguity is the main issue. When the software is completed it usually has low quality and is not scalable or reusable. The management of the evolution of the software becomes indeed really difficult and often brings to the need of rewriting the whole system or some parts of it.

The second diagram introduces the concept of model, even if still weak. In this case a model is a representation (in a defined notation) of the structure and behaviour of a system. These kind of models are referred to as "code models", because they only represent the written code. Sometimes changes to the system can be made upon the model, but always bringing them also on the code.

*Roundtrip Engineering* is similar to the spiral we previously introduced. Now the developers start from the definition of a model of the system architecture and behaviour. This model tipically reports three main characteristics of the system: Structure, Behaviour and Interaction within the parts. The level of detail is decided by developers and can vary from model to model. Once the model has been validated, a new phase is started called *Model-To-Code* transformation. During the translation of the model into the proper code, the code can show some weaknesses of the model, leading to a model refactoring. These kind of backward steps can be activated also during the modeling phase.
The Model to Code phase can be manual or automatic, depending on the technology adopted for the definition of the model. Typically, in case of automatic generation, the main skeleton of the application is generated and then the developer has to add some little parts related to the business logic.
This is the main approach used in Model Driven Software Development. It is worth noting that MDSD is still not a top-down process, but still iterative.

The Model-Centric approach has the objective to model all the system details so to generate all the needed code without the addition of any information. This means that the model has to include the representation of the persistent and non-persistent data, business logic and presentation elements. All the complexity is indeed brought at the model level.
This is the direction where MDSD is going. In this approach the produced model should be Platform Independent, so to be applicable to each desired platform.

Last method is the development of a system only focusing on model. This is the typical approach of companies that outsource projects implementation and maintenance, while they keep control on the overall models. Developers only produce models of the system, providing a formal description of all the involved details.

The direction of MDD is indeed the development of a Software through the definition of a Model that includes the functionalities and the architecture of the system. Code is the last part of the development phase and often is automatically generated.

Companies are moving in MDD direction because it also promotes a dramatic improvement of productivity and an important decreasing of costs. The improvement of productivity is either short-term and long-term.

Short term productivity is due to the chance of automatically generate a system only providing its model. This means that, once the initial model is produced, a prototype of the working system can be developed quite fast (*Fast Prototyping*).

Long term productivity depends on the primary software artifact's longevity. When a model is well defined and captures all the proper details of the system, it remains valid for a long period.

In the next sections we will describe some of the standard tools introduced to promote modeling, giving also an overview of the main standard meta-models and meta-metamodels.

## 1.3 Model-Driven Architecture

Model-Driven Architecture (MDA) is a style of MDD formalized by the Object Management Group (OMG). It is based on a set of standards to define a set of models, notations and transformation rules. These standards are developed by OMG and among them we find: Unified Modeling Language (UML), Meta-Object Facility (MOF) and Common Warehouse Metamodel (CWM).

To define MDA conceptual framework and vocabulary the OMG has introduced a set of layers: Computational Indipendent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM) and Implementation Specific Model (ISM). A representation of the architecture can be found in Figure 1.2 and is taken from [2]. Having multiple layers, and consequently models, we can think that the development of a system is based on the definition and refinement of models. These operations can be critical and can involve insertion of details or convertion from one model to another.

MDA approach let the developer focus on the essential parts and details of the system, ignoring all the platform characteristics. Models

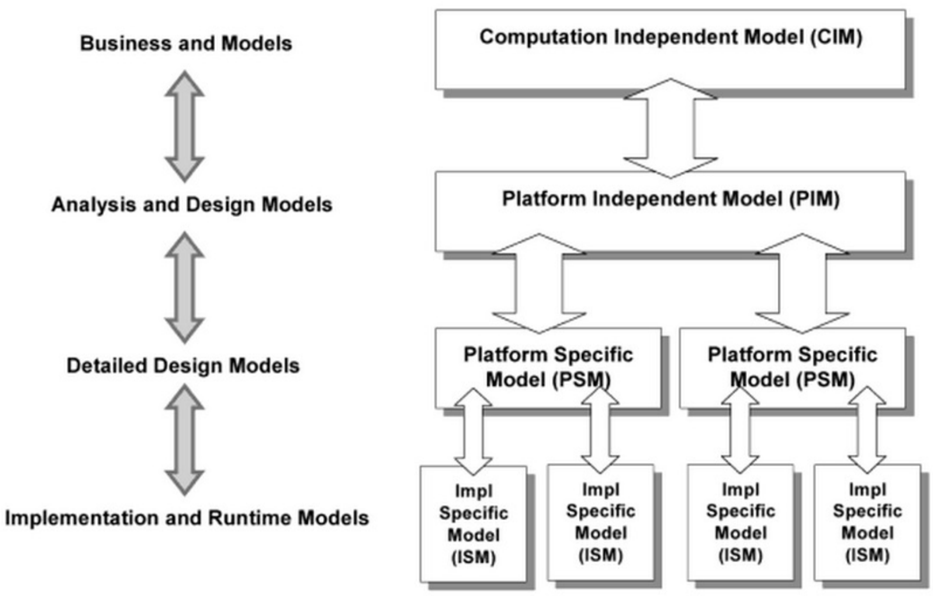

Business and Models
Analysis and Design Models
Detailed Design Models
Implementation and Runtime Models
Computation Independent Model (CIM)
Platform Independent Model (PIM)
Platform Specific Model (PSM)
Platform Specific Model (PSM)
Impl Specific Model (ISM)
Impl Specific Model (ISM)
Impl Specific Model (ISM)
Impl Specific Model (ISM)

Figure 1.2: MDA architecture layers and transformation

become indeed Platform Independent and can be applied to every existing platform. With the world ”platform” we can intend a wide variety of existing technologies, like Operating Systems, environments and so on. It is remarkable the fact that, whatever we intend with platform, the important is to think in term of models at different level of details and abstraction. In this way the transformation between models becomes first-class elements of the development process.

Models tranformation requires a clear and formal way to describe the semantic of a model. So each model has to be described in another model, called meta-model. OMG recognize that for modeling it is fundamental the concepts of metamodels and formal semantics. For this reason they defined a set of meta-modeling levels and a standard language to define metamodels, called Meta-Object Facility (MOF).

## 1.4 From Standard to Domain Specific Modeling Language

OMG has formally defined its standard meta-modeling language, called MOF. The MOF 2 Model can be used to model itself as well as other models and other meta-models (e.g. UML).

UML is based on MOF, in the sense that it is a meta-model defined on the MOF meta-metamodel. A graphic representation of the stack is in Figure 1.3. In this picture we can see the layers we have talked

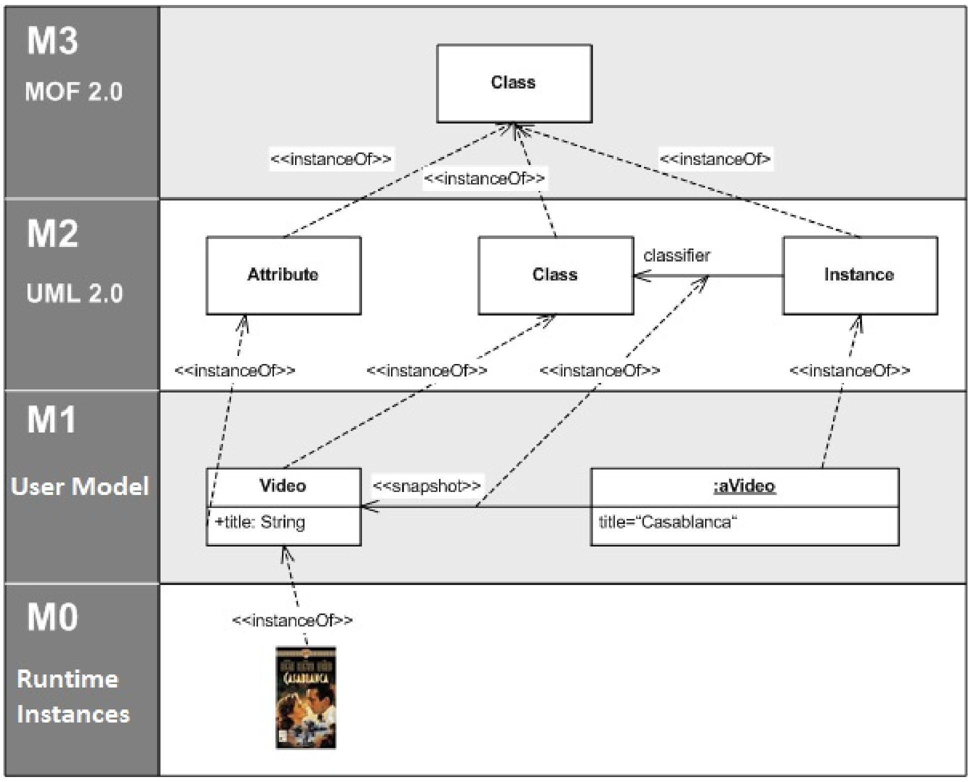


Figure 1.3: UML layers

about earlier. M3 is the meta-metamodel level and formally defines a set of concepts. These concepts are used to define the level M2, the so called metamodel. In this case the metamodel is UML. In M1 we find the model of the system expressed as a set of concepts defined in M2. M0 defines the runtime instances of the designed system.

Actually the MOF 2 Model is made up of two main packages: the Essential MOF (EMOF) and the Complete MOF (CMOF). A deeper description can be found in [1].

UML is a really powerful language, but it includes also some limits.

When a domain presents peculiar characteristics that often appears in problems on that domain, we can start thinking about a specific Meta-Model for that Domain.

The Meta-Model should be based upon MOF and should express the set of concepts that are specific of the observed domain.

We now come to the central point of this chapter: the definition of custom Domain Specific Models by the replacement of the M2 level of Figure 1.3 with a new Meta-Model defined by the developer. Once this Meta-Model is defined, a Model of the system can be expressed in the level M1.

Figure 1.4 reports a scheme of the models architecture to be defined. M3 level is of course the MOF level. We can then say that each application domain has peculiar recurrent requirements and entities, so Domain Specific Models (DSM) can be more useful than UML. These DSM must be defined as instances of (E)MOF so to define a language different from UML.

A Meta-Model expressed in MOF can be seen as the *Abstract Syntax* of a language to define models. MOF itself constitutes a Domain Specific Language (DSL) to define meta-models, as the EBNF notation costitutes a DSL to define grammars.

## 1.5 Ecore and EMF

Ecore is the version of EMOF implemented in Eclipse IDE. It defines the following concepts:

- EPackage: represents a container of information.

- EClass: represents a class. Can have zero or more attributes and zero or more references.

- EAttribute: represents an attribute with a name and a type.

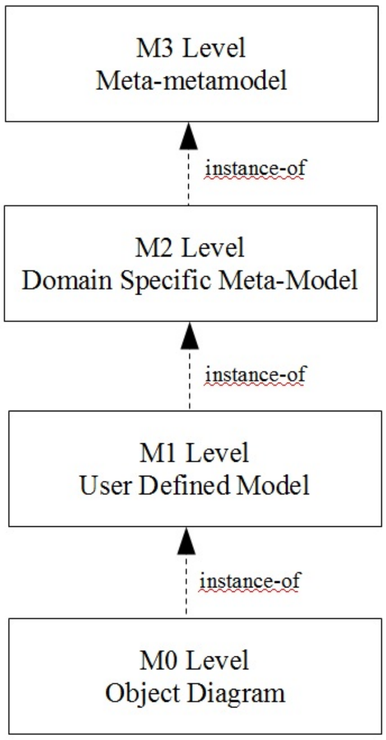


Figure 1.4: Architecture to be achieved when a Domain Specific Model is defined

- EReference: represents an association between classes. It has a name, a boolean to indicate containment and a reference target type that is a second class.

- EDataType: to represent a type.

- EENum: to represent an enumerative type.

- EOperation: represents an Operation in a EClass.

Eclipse IDE implements also a modeling tool called Eclipse Modeling Framework (EMF) to define models instances of the ECore Meta-metamodel. Actually EMF is now considered as de facto standard for Java Model Driven Software Development (MDSD) on Eclipse because it automatically generates the code related to a define ECore model (Model to Code translation). Actually the defined model should be instance of a Meta-Model that in turn is instance of the ECore Meta-metamodel.

EMF can also generate plugins that can be used as syntax driven editors and also a graphical representation of the defined model.

## 1.6 Domain Specific Languages

A Meta-Model can be seen as the Abstract Syntax of a language. Since MOF is also a Meta-metamodel and can be used to define Meta-models, we can use MOF as a reference language for the definition of these specialized languages.

When we talk about good quality software, we refer to softwares built following a specific organization and devided into three main parts:

- *generic part*: a part that could be shared in all future applications.
- *schematic part*: a part organized according to known schemas reusable in different contexts. This can be the "skeleton" of the application.
- *specific part*: specific to the application, also called *application logic* or *business logic.*

The definition of a software system through Domain Specific Languages can introduce several benefits, related to the quality and reusability.

The definition of a DSL starts from the recongnition of the domain common concepts that can be expressed by specific rules in the language. In this way analysts do not have to start each time from the beginning, having defined all common features of the explored domain in the DSL.

Designers can focus on the problem and not on the platform, developing Platform Independent projects.

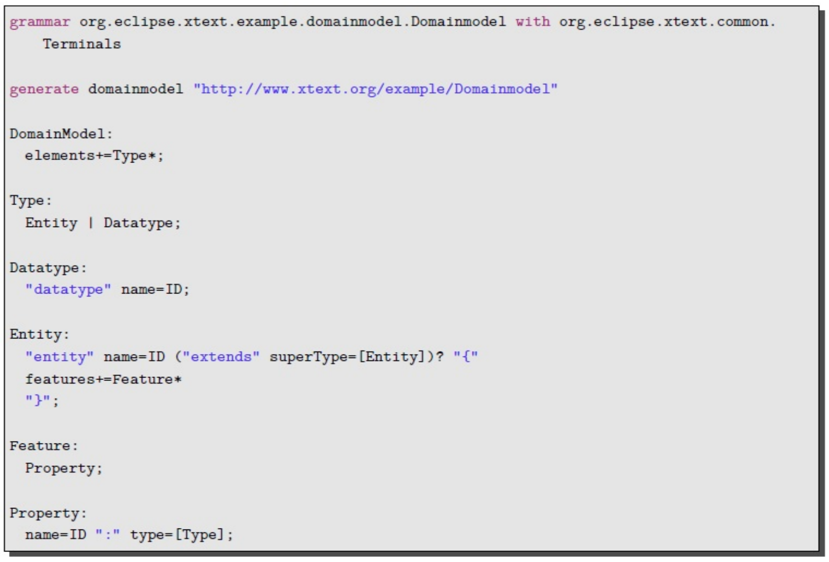
Programmers (or Application Designers) can use automatic code generators developed by System Designers to automatically generate the code related to a model defined as an instance of a meta-model.

The definition of DSLs aim at the creation of *Software Factories* related to specific domains. A company that often has to cope with problems in the same domain should think about creating its own DSL, to automate the coding phase and to have an important saving on producing costs.

Of course the definition of a DSL and proper code generators can be quite expensive, but it is a one time task and its costs can then be spread on all future applications on that domain.

Actually, one of the main framework developed to define DSL is XText.

## 1.7 XText and XTend2 Brief Introduction



```
grammar org.eclipse.xtext.example.domainmodel.Domainmodel with org.eclipse.xtext.common.
    Terminals

generate domainmodel "http://www.xtext.org/example/Domainmodel"

DomainModel:
  elements+=Type*;

Type:
  Entity | Datatype;

Datatype:
  "datatype" name=ID;

Entity:
  "entity" name=ID ("extends" superType=[Entity])? "{"
  features+=Feature*
  "}";

Feature:
  Property;

Property:
  name=ID ":" type=[Type];
```

Figure 1.5: Example of Xtext specification

XText is a framework that allows the definition of a Meta-Model through the Abstract Syntax of a language expressed in EBNF notation.

It has born as part of the framework OpenArchitectureWare (OAW), coupled with the language XTend, but now it has become opensource and part of the Eclipse community. In fact XText comes as a feature to the Eclipse IDE and supports EMF, so the parsing of an XText defined language is automatically mapped on an Ecore meta-model. In Figure 1.5 there is an example of how a simple XText specification looks like.

XText builds a bridge between the world of Programming Languages and the world of (meta)modeling. Before the definition of EMOF there was no standard language for modeling.

Furthermore, other XText tools are available to perform Model-to-Code or Model-to-Model transformations, like for example XTend2.

XText not only generates a parser for the defined grammar (as normally done by standard parser generators), but also a Model for the Abstract Syntax Tree and a customizable Eclipse syntax driven editor, related also to the concrete syntax of the language.

Without this tool the mapping of the concepts introduced by the Meta-Model on the proper language constructs should be done manually by the developer, so the time and costs reduction is evident.

XText also includes the chance to write custom validator for the written model, that dinamically control if specific rules are observed. These controllers are written in the form of Java functions.

The XText version used in this thesis is the 2.3.

# Chapter 2

# Embedded Systems

An Embedded System is a computer system built to accomplish specific tasks possibly inside a larger system. These systems are often provided with various mechanical parts, like sensors, actuators and other hardware.

Embedded systems often include supports for real time computing, meaning that they usually support computational models that are strictly coupled with the concept of time and that some functions have to be executed respecting proper time constraints. Real-Time computing is a key feature in controllers, automation etc.

Embedded Systems are special purpose computers. They are in fact in contrast with the definition of general purpose machines, like PCs, that are much more flexible and can be used for a wide variety of functions.

Nowadays we can find Embedded Systems inside a lot of common devices e.g. printers, cars, ovens, washing machines, dishwashers etc.

In this chapter some of the main characteristics of an Embedded System will be introduced, giving a general overview of the argument.

## 2.1 Main Features

Embedded Systems always incorporate a Microcontroller or a Digital Signal Processor (DSP). In general, microcontrollers used are quite simple and often are not even comparable to what we find in PCs. The motivation of this choice is that these systems often have to continuously accomplish simple tasks and in a complex system we can find more than only one; so there is no reason to have much powerful microcontrollers that unnecessarily raise their price. In fact the other main reason is the cost. As in every computer system costs must be proportioned with the tasks it will have to execute. For this reason their cost is kept quite low.

In general it is not possible to define all the architectures and characteristics of Embedded Systems because they can be really different each other. So in this section we will discuss some key features common to every embedded system.

Embedded Systems are designed to execute some specific task, rather than dedicated to multiple tasks like general purpose computers. Some of them can also have strong real-time constraints to be met, while others may have not. Real-time constraints are introduced to ensure safety and performances.

As we saw earlier, an Embedded System may be not a standalone device. It can be integrated within a larger system, like in the automotive field.

Interface with the user or with other devices can be done in several ways: hardware components, LCDs, Ethernet, Serial etc.

Hardware interaction can be reached through the introduction of special buttons and other means to check the result of an interaction, like for example LEDs. More advanced techniques concern the use of LCDs to display some information or to interact with the hardware.

Some Embedded Systems come with the support to Serial protocols (via e.g. RS232, I2C, USB) or network protocols (e.g. Ethernet). Of course these supports let the definition of more powerful user interfaces, like, for example an interface on a PC.

**Processors** In embedded systems there is always the need of a computational unit. The most common are microprocessors, microcontrollers and DSP. The architecture of these units can vary from case to case and includes: Von Neuman, various degrees of Harvard architectures, RISC, non RISC and VLIW.

Bus parallelism can go from 4 to 64 bit, even though most common is from 8 to 16 bit.

**Peripherals** Many peripherals are supported to communicate with other systems, supporting a wide range of protocols.

The main supported protocols and hardwares are:

- Serial: it can be synchronous or asynchronous.

- Univesal Serial Bus (USB)

- Network: like Ethernet, XBee etc.

- Multimedia Storage: like SD, Flash etc.

- Fieldbuses: like CANBUS, PROFIBUS etc.

- Discrete IO

- Analog to Digital and Digital to Analog converters

**Reliability** These kind of systems are often hidden to the user, or sometimes completely inaccessible. This means that they have to guarantee an high degree of reliability, even because they usually have to correctly work 24/7.

Reliability concerns either Hardware and Software. Hardware reliability is reached in the design phase, while software one is much more difficult to reach. To have a good software it has to be well designed and well tested and debugged. Sometimes these softwares should be capable of react to some occurred error and recover themselves.

To cope with software errors a variety of techniques have been introduced. One is the introduction of a watchdog timer: if the system does not periodically notify the timer, it means that some problem happened, so the watchdog reset it.

To have an higher reliability, redundant circuits can be added, like power supplies, backups etc. In these systems redundancy is not considered as a problem because it can solve reliability problems with relatively low prices.

## 2.2 Software

The software produced and installed in these systems is usually called *Firmware*. A firmware is intended as a software born to be run on special purpose devices and it includes all the needed features. It in fact has to implement all the registers management operations, all the communication protocols needed and all the computational operators.

The word "firmware" reminds a non modifiable software installed on a read only memory. In present devices, firmwares are installed on ROMs or Flash Memory, always persistent memories.

### 2.2.1 Software Tools

The implementation of a firmware always requires a compiler and a way to transfer the generated program on the Embedded System memory.

Usually a program is written in an IDE, using an high level language. Actual embedded systems are infact programmed in high level languages like C, C++ etc. while first firmwares were written directly in assembly or other low level languages.

Once a program is written, it is compiled in the supported low level language and then uploaded in some way on the proper memory, e.g. via USB.

All the tools needed to write, debug and upload the program have to be provided by the producer.

### 2.2.2 Programming Languages

There are many supported programming languages for embedded systems, depending on their characteristics and on the producers. Here an overview of which are the main languages is provided.

#### C Language

C is one of the most used programming language for these systems, including also all its evolutions, like C++.

Usually supports for special purpose operations, like hardware ones, are introduced through libraries.

### Statecharts Diagram

The Statechart diagram is used to model the evolution and the behaviour of a system. It is a set of states, representing the actual state of the system, events, actions and guards. States can be initial, intermediate or final and represent a defined moment in the life cycle of the system.

The system evolves from one state to another thanks to the happening of an Event; a transition is activated from an event, and the action is the consequent event. The other main component of statcharts are guards. They are further conditions to be met to execute an action.

Statecharts diagrams have also been introduced in the UML standard.

### VHDL and Verilog

VHDL is the acronym of VHSIC (Very High Speed Integrated Circuits) Hardware Description Language and it is, with Verilog, the most used language for the design of digital electronic systems.

Even though this language presents the classical structure of a programming language (control statements, loops etc.) it is used as a model to describe the structure of the hardware components of the system. It also supports the definition of concurrent parts, intended as parts of the software with their own control flow.

VHDL allows also the definition of the interaction between the functional blocks that constitute the system. The exchanged information are essentially control signals and data. Each functional block is also descripted by a input-output relation.

The language Verilog supports the design, test and implementation of digital circuit using a C-like syntax. Coupled with the VHDL it can be considered one of the few languages used in the industrial design and simulation.
Also this language supports parallel and concurrent processes.

All the sequential operations are written inside a block delimited by a begin and an end. All the concurrent instructions and all the begin/end blocks are executed concurrently. It is also possible to define a hierarchy of modules.

### PLC Programming Languages

PLCs are special purpose computers, originally used in industrial plants to manage production processes. They are capable of executing programs and interact with sensors and actuators.

Now, with the continuos reduction of size and costs we find them in smaller systems too, also in houses.

PLCs are composable and extremely reliable. In fact they are often used in environments with high electrical interference, high temperature, high umidity and so on.
The structure is always adapted to the functions covered and they can be programmed in a wide range of languages. Usually programs are written in PCs on specilized softwares that promote also the upload of the program on the memory of the on board CPU.

PLCs programming languages are divided into to families: *graphic* and *textual* languages.
Among graphic languages we find:

- *Ladder Diagram*: it is based on graphic diagrams of electronic circuits based on relay logic hardware. It was the main programming language for PLC because it maps directly electronic circuits into software.
- *Function Block Diagram*: it uses block diagrams to model the behaviour of the system. Each block describes a function between input and output variables.
- *Sequential Function Chart*: a program is a sequence of steps, each one with associated actions. The other main component is the transition, each coupled with a condition and directed linked to steps.

Textual languages are instead:

- *Instruction List*: it is a low level language, similar to assembly. It can give the full control of the PLC, but it is quite difficult to use.
- *Structured Text*: it is an high level programming language, really similar to Pascal. It supports conditional statements, loops and functions.

### 2.2.3 Embedded Software Architectures

There are many kinds of software architecture defined in the Embedded Systems field and here a brief summary of them is presented.

#### Simple Control Loop

In this approach the software is based on a continuous loop. All the functions are called in the loop and the system repeatedly executes the same functions. Each function manages an aspect of the software or hardware.

#### Interrupted Controlled Systems

Some Embedded Systems are mainly controlled through Interrupts. An Interrupt is the execution of a task caused by the happening of a predefined event, for example a timer or a rising edge generated by a sensor.

Event handlers should always be as simple and as fast as possible to introduce low latency and guarantee a good level of reactivity and reliability.

#### Cooperative Multitasking

A nonpreemptive multitasking system is really similar to the control loop architecture where the loop is hidden in an API. The program is composed of tasks, each one executed in its protected environment. Tasks are executed in a queue and when they are not in execution they become idle.

#### Preemptive Multitasking or Multithreading

In this type of systems a low level program switches between all the defined tasks or threads using a timer. With this approach it seems that tasks are executed in parallel, even if it is not like that.

To switch between tasks the scheduling algorithm becomes a really important part of the ”operating system kernel” in execution on the embedded system. This behaviour promotes interleaving between different control flows, so softwares should be well defined to avoid inconsistency problems.

This architecture is often implemented in Real-Time operating systems, so the developer can implement programs without the need of taking care of the scheduling algorithm.

### 2.2.4 Wiring

Wiring is an opensource programming framework for microcontrollers based on C++. It allows writing cross-platform software to control devices attached to a wide range of microcontrollers boards.

Currently it supports the Wiring hardware and all the hardware based on the AVR atmega processors.

The software architecture used in Wiring is the Single Control Loop. Each program (called Sketch) has to be organized in two main functions: *setup* and *loop*. The *setup* procedure is executed only once during the initialization of the board, while the loop procedure is repeatedly executed infinite times.

Arduino is based on the ATmega328P and Wiring is used as a framework to program it. Wiring introduces libraries to communicate with hardware devices, while removes other features introduced by the C++ (e.g. processes instantiation).

Further details on Wiring can be found in citel.

# Chapter 3

# Introduction to Arduino

Arduino is an Open-Source Project born in Italy with the goal of enhancing fast-prototyping and development of general-purpose hardware.

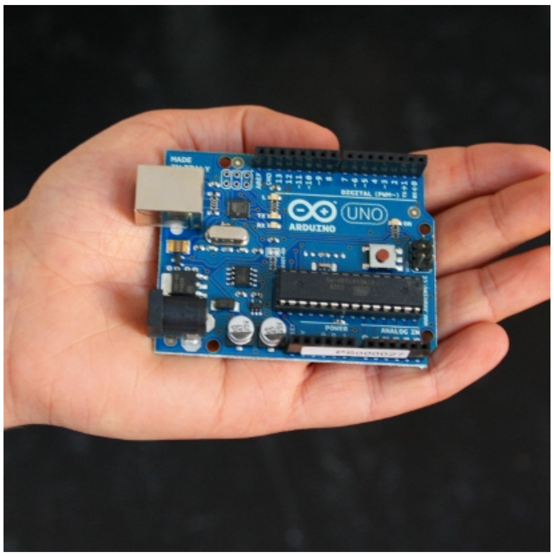


Figure 3.1: Arduino UNO Board

It is an Embedded System that provides a wide set of hardware pins conceived to directly interact with different kinds of hardwares in a simple and fast manner. It is easily extensible by exploiting the stack mechanism it is conceived with. In fact many features are introduced on different hardware boards, called *Shields*, that can be stacked on the main Arduino board.

In this chapter we will go through a brief overview of the features implemented in Arduino UNO, starting from the Software viewpoint and then proceding to the Hardware. Some features will be described in following chapters because they are strict related to what will be introduced there.

## 3.1 Arduino Software Introduction

Arduino is programmed in C++ using a special framework called Wiring. Wiring is an Opensource programming framework for microcontrollers that allows to write cross-platform software to control devices attached to a wide range of microcontroller boards.

Wiring is a C++ based framework, where some features of the language have been removed (due to resource limitations of embedded systems) and other added (with specific libraries) to interact with hardware. For example it does not provide multi threading support, but it provides functions to read from hardware pins.
Wiring has a predefined software structure composed by two functions:

```
void setup(){}

void loop(){}
```

The *Setup* function is executed only once during board initialization. *Loop* is instead repeatedly executed.

The computational model implemented is indeed a single control flow that repetedly executes a sequence of instructions written in the *loop* function. The control flow pass through the procedure *setup* only once during the initialization of the program. A sequence diagram that describes this behaviour is in Figure 3.2.

It is a classical *Single Control Loop* approach, even though it provides also a support for interrupts. The diagram in Figure 3.2 refers to the control flow of a Wiring program that is not affected by *Interrupts*. Arduino supports in fact two kinds of Interrupt: External

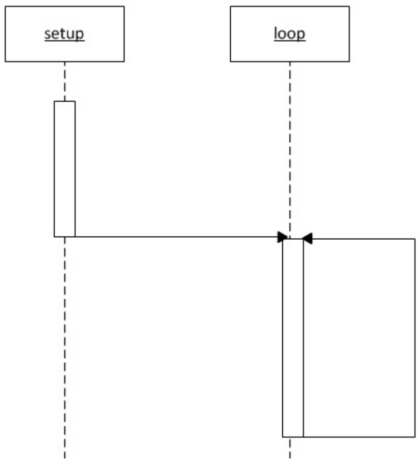


Figure 3.2: Basic Sequence Diagram of a Wiring Program

and PinChange; a deeper description of them will be provided in the following sections.
By now we will only focus on the general concept of *Interrupt* and on its semantic. An Interrupt, in an Embedded System like Arduino, is an Asynchronous event raised by an electronic signal that interrupts the normal control flow of the program to execute a predefined routine. This routine is an handler of the event and can modify the state of the system.

We can indeed say that Arduino supports a unique control flow that can however be perturbed by the occurrence of certain events. To graphically explain better the concept let's have a look to the Picture 3.3. In the picture the *loop* procedure is interrupted by a certain event and the control flow passes to the handler of the *Interrupt.* When the routine is completed the control returns to the next instruction still to be executed of the procedure *loop.* In the picture the control flow starts again the *loop* function when it completes, but, during the next execution, it will not pass through the handler, unless an interrupt is raised again.

In [10] the reader can find a good overview on the relation between Wiring and C++.

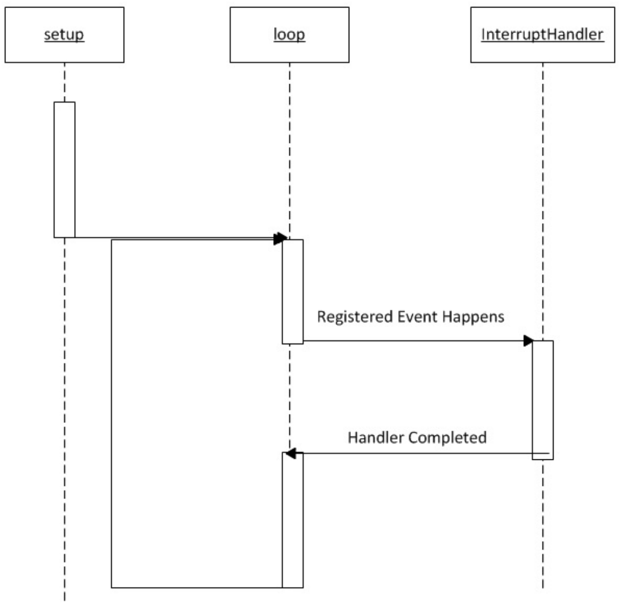


Figure 3.3: Sequence Diagram of a Wiring Program interrupted by an Interrupt

## 3.2 IDE

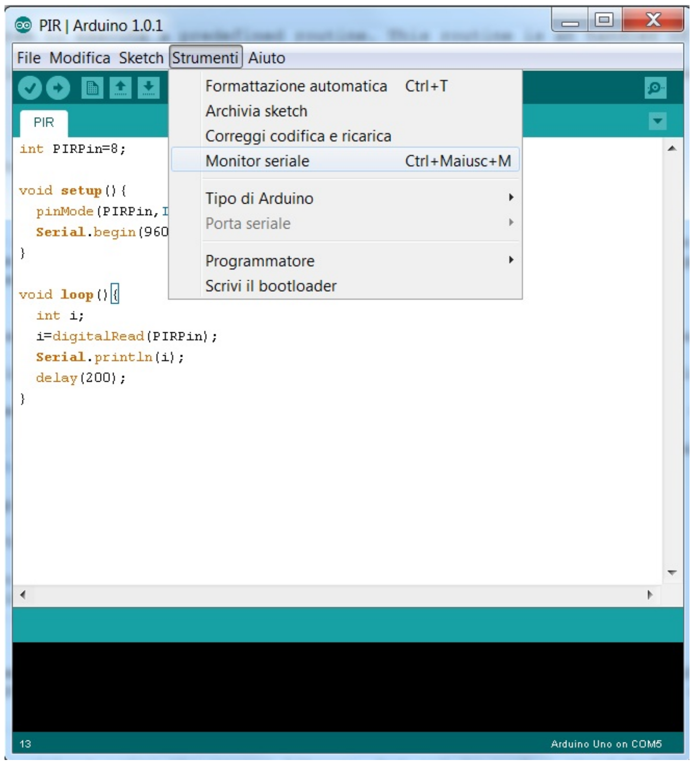
The Development Environment is based on *Processing* and let the user write Arduino programs and upload them on the board.

The code is uploaded to Arduino through the USB port and, in case of multiple boards, the user can select to which board upload the code changing the destination USB port.

The IDE provides also a Serial Monitor to show what has been received on the USB. This is the fastest way to debug the code or to see the results of the application.

The only way to send data to the Serial Monitor is through the Serial. The Arduino program should first initialize its Serial with the proper Baudrate, the default in Serial Monitor is 9600, and then send data to the PC using the function *Serial.print*.

A screenshot of an Arduino IDE is in Figure 3.4.


PIR | Arduino 1.0.1
File Modifica Sketch Strumenti Aiuto
PIR
Formattazione automatica Ctrl+T
Archivia sketch
Correggi codifica e ricarica
Monitor seriale Ctrl+Maiusc+M
Tipo di Arduino
Porta seriale
Programmatore
Scrivi il bootloader
13
Arduino Uno on COM5

```
int PIRPin=8;

void setup(){
  pinMode(PIRPin,I
  Serial.begin(960
}

void loop(){
  int i;
  i=digitalRead(PIRPin);
  Serial.println(i);
  delay(200);
}
```

Figure 3.4: Arduino IDE

## 3.3 Libraries

Arduino comes with a wide range of supported libraries, implemented to support various features. A brief list follows:

- EEPROM: manage Storage and Retreiving of Data in the EEPROM Memory
- Ethernet: introduce support to Ethernet networks
- Firmata: implementation of the Firmata protocol to communicate with other Software
- LiquidCrystal: implements APIs to use LCDs compatible with the Hitachi HD44780 driver

- PinChangeInt: this library is used to easily take advantage of interrupts on each pin of the board

- SD: used to read and write on SD cards

- Servo: ease the usage of Servo motors

- SoftwareSerial: introduce a support for Serial communication on each pin of the board, instead of only on pin 0 and 1

- SPI: allows the communication with SPI devices. SPI (Serial Peripheral Interface) is a synchronous serial protocol used by microcontrollers to communicate with other peripherals

- Stepper: used to control Stepper Motors

- Wire: implements I2C communication within Arduino and other devices

Obviously all the introduced features could be reached also without using the proper library, but not in such a straightforward manner.

## 3.4 Arduino Hardware Introduction

Arduino UNO is based on the microcontroller ATmega 328p and its pin mapping is shown in Figure 3.5. Each pin of the microcontroller has a well defined function and features.

The pure electrical pins (power supply and so on) are 7, 8, 20, 21 and 22. Pin 7 is the 5V output. Pins 8 and 22 are connected to the Ground. Pin 20 is the AVCC, that is the supply voltage pin for the A/D Converter. Pin 21 is the AREF pin and is used to configure the reference voltage for analog input.

The remaining pins are mapped on three different *Ports*:

- B (digital pin 8 to 13)
- C (analog input pins)
- D (digital pins 0 to 7)

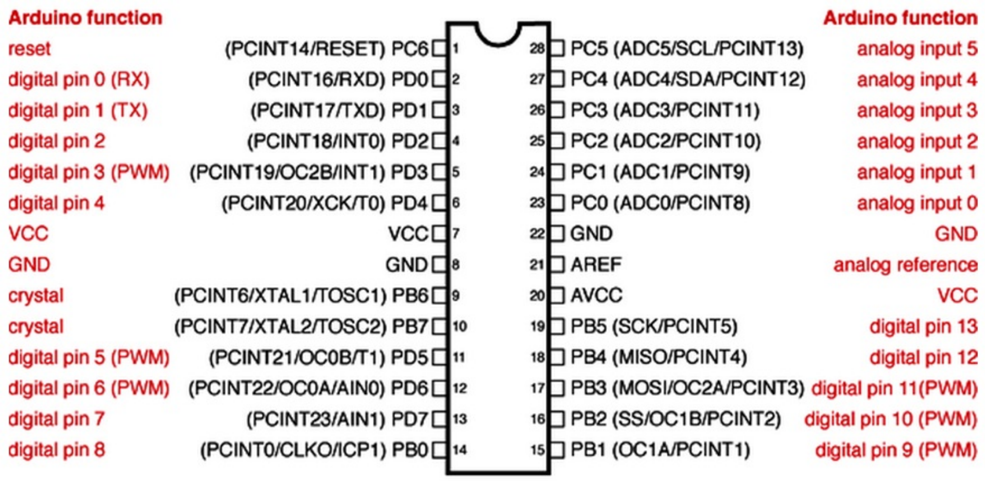


Figure 3.5: ATmega 328p Pin Mapping

In the picture they are indicated respectively by the prefixes PB, PC and PD. A Port is only a concept introduced to identify a register on the microcontroller. Actually not only one register for each port, but three registers for each one of them. Indicating with $X$ the letter that identifies each port, the description of each register follows:

- *DDRX* - Data Direction Register X: each bit of this register is associated to a pin belonging to port X. This register records if a pin is configured as Input or Output. A pin configured as an Input is stored as 0, while a pin configured as an Output is stored as 1.

- *PORTX*: this register is used to set the state of the output pins. So for example if a pin is configured as an Output (1 in DDRX) and the corresponding bit in the PORTX register is set to 1, on that pin that will be the value 1. This register can be either written or read.

- *PINX*: this is the Input register. It stores the value of each Input pin and let the user read the value of a group of pins at the same time. So for example if a pin is configured as an Input (0 in DDRX) and the corresponding bit in the PINX register is set to 1, it means that on that pin there is a logical high value. This register is read only.

Arduino supports Digital Input and Output and Analog Input. Analog Output is not supported, but it is simulated through PWM outputs.

In Figure 3.5 there are two additional columns reporting the Arduino function associated to each pin of the microcontroller. Following these functions we can talk about two different categories of pins:

- Digital Pins: can be used either as Digital Input or Digital Output. Some of them are marked as *PWM* meaning that they can be used as PWM outputs. This feature is introduced to simultate Analogic Output and is described later in this document.
- Analog Inputs: can be used to read an Analogic value. They can also be used as Digital Inputs or Outputs.

Each pin can be set as Input or Output, not both at the same time.

The Microcontroller ATmega 328p supports a 16 MHz clock and provides quite limited resources, especially for memory. It in fact provides:

- Flash Memory : 32KB. It is used to store the Bootloader and the Sketch. It can be used to store varables too, declaring the variable with the keyword *PROGMEM*.
- SRAM (Static RAM): 2KB. This is a volatile memory that stores variables at runtime.
- EEPROM: 1KB. This memory is used to store long-term information.

# Chapter 4

# Arduino DSL and Semantic Mapping

In this chapter will be shown all the steps followed to reach the definition of a Meta-Model for Arduino.

The defined meta-model will abstract from the kind of hardware, trying to capture the main features provided by the framework Wiring. In this way it is possible to translate a Model into the appropriate code, specifing the hardware where it will run on.

To give a first sight of what it means developing an Arduino program we will start from a first example to introduce the main concepts of the platform. Then we will go through the definition of the Meta-Model and of some new abstractions.

We will then go up in abstraction and start talking about the concept of System composed by a set of Entities communicating each other.

The introduction of a metamodel for these kind of systems is also motivated by the need to bring at the software level some concepts that normally are part of the operating systems, like interrupts, event management and so on, and by the need to abstract from the hardware viewpoint and bring the computation on the software side.

At the end of the chapter is proposed the Model of the example introduced at the beginning. The Model is an instace of the introduced Meta-Model. In this way the reader can have an idea of which are the main advantages of Meta-Modeling and how the classical software development cycle is affected by the introduction of Meta-Models.

As introduced in Chapter 1, a Meta-Model can be formalized through the definition of a set of EBNF rules that define a Domain Specific Language. In this work the Meta-Model will be specified using a DSL.

## 4.1 A First Domotic System

Now that an introduction to Arduino Software and Hardware has been provided, we can face a first example to further understand it.

Suppose we want to realize a simple Domotic System to monitor our apartment. The requirements are:

- When the temperature is over 30 Celsius a red LED has to turn ON. It has to stay OFF otherwise.
- When it is getting dark a green LED has to turn ON. Otherwise it has to stay OFF.
- When the entrance doorknob is rotated, previous LEDs have to blink three times at the same time.

Let's start from the electronic components we should have to implement this simple domotic system. We need to sense the temperature and the light, so we will have to introduce at least a *Thermistor* and a *Photoresistance*. We also have to buy two *LED*s, one red and one green.
To react to the doorknob rotation we need a *Tilt* sensor.

In the Analysis phase, we also consider that it is not necessary to continuously check all the sensors, but it is enough to check them every 400ms. This is also thought to avoid high power consumption. Of course everything will be connected to an Arduino UNO board.

The complete circuit should be like the one shown in Figure 4.1. Of course this is not the only possible implementation, but just one of the many.

The code that implements the system has been organized with a modular approach and it follows:

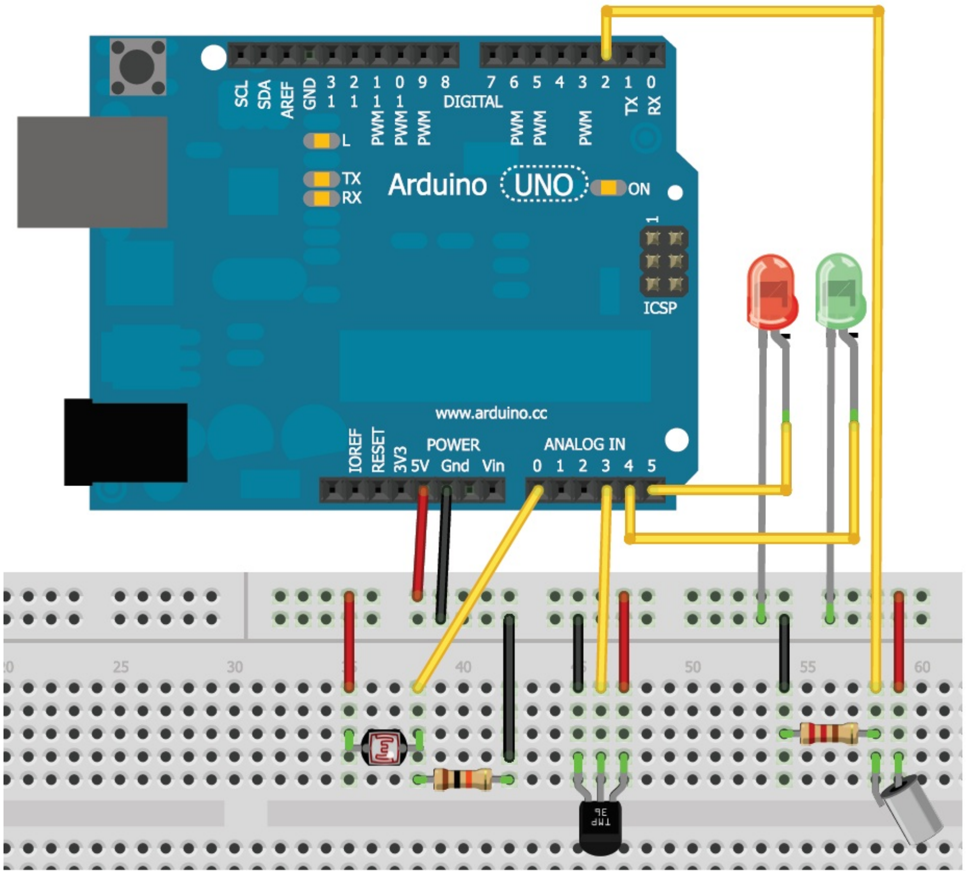


Figure 4.1: First simple domotic example ciruit

```
#include "Arduino.h"

int redLEDPin=A5;
int greenLEDPin=A4;
int thermistorPin=A0;
int photoresistancePin=A3;
int tiltPin=2;
int lightVal;
int tempVal;


void setup(){
  // Configure LED pins as OUTPUT
  pinMode(redLEDPin,OUTPUT);
  pinMode(greenLEDPin,OUTPUT);
  // Configure other pins as INPUT
  pinMode(thermistorPin,INPUT);
  pinMode(photoresistancePin,INPUT);
  pinMode(tiltPin,INPUT);

  // Attach interrupt routine to pin 2.
  // Raised on rising edge!
  attachInterrupt(0,tiltHandler,RISING);
}

void loop(){
  pollLightVal();
  pollTemperatureVal();
  delay(400);
}

// Check the value of the light sensor
void pollLightVal(){
  // Read analog sensor value
  lightVal=analogRead(photoresistancePin);
  // If it is getting dark (val<550) turn ON the Green LED
  if(lightVal<550)handleGettingDark();
  // Otherwise turn it OFF
  else handleSunIsShining();
}

// Check the value of the temperature sensor
void pollTemperatureVal(){
  // Read analog sensor value
  tempVal=analogRead(thermistorPin);
  // If temperature > 30Â°C (tempVal>164) turn ON the Red LED
  if(tempVal>164)handleTempOver30();
  // Otherwise turn it OFF
  else handleTempUnder30();
}

void handleGettingDark(){
  digitalWrite(greenLEDPin,HIGH);
}

void handleSunIsShining(){
  digitalWrite(greenLEDPin,LOW);
}

void handleTempOver30(){
  digitalWrite(redLEDPin,HIGH);
}

void handleTempUnder30(){
  digitalWrite(redLEDPin,LOW);
}

void tiltHandler(){
  int i;
  // Blink both LEDs three times
  for(i=0;i<3;i++){
    digitalWrite(greenLEDPin,HIGH);
    digitalWrite(redLEDPin,HIGH);
    digitalWrite(greenLEDPin,LOW);
    digitalWrite(redLEDPin,LOW);
  }
}
```

The *loop* function is only composed by three procedures calling, *pollLightVal*, *pollTemperatureVal* and *delay*. As the names suggest, the first two will implement a Polling and the third one will suspend the loop for 400ms.

The polling procedures check the actual value of a pin and, if a condition is met, execute the proper function. In both pollings, the procedures executed when a condition is met are called with a name that starts with the keyword *handle*. This means that their aim is handling a particular event. Following what is declared in the requirements, they act on the proper pins to achieve those *desiderata*.

The Arduino examples proposed in the Web are almost never organized in such a modular way. We would have found a bunch of code written in the loop function, with no other procedures, except the interrupt one that has to be declared necessarily.

This code structure is proposed to give an idea of the good quality code that will result from the Software Factory developed in this thesis.

From this little example we can also start to capture some of the main concepts of Arduino.

We observe that some of the pins are set as input and others as output using the procedure *pinMode*. This suggests that input pins are linked to some kind of <u>*Sensors*</u>, while <u>*Actuators*</u> are linked to output pins. Sensors can be Digital (like the *tilt* sensor) or Analog (*thermistor* and *photoresistance*). Digital pins are read with procedure *digitalRead*, while analog pins are read with *analogRead*.

In the *setup* procedure there are all the initialization operations, while in the *loop* procedure we find the logic of the application. Particularly interesting is the implementation of the behaviour related to light and temperature values. What is done here is a continuous <u>*Polling*</u> on the proper Sensors, taking the right decision depending on their values.

The behaviour related to the doorknob is managed through an *Interrupt*. As we will see later in this document, Arduino supports External Interrupts only on pin 2 and 3. Related interrupts are indicated respectively with 0 and 1. In the setup phase, the procedure *attachInterrupt* is called to attach an interrupt routine to the event *Rising Edge* on pin 2. When this event happens (transition 0 to 1) the normal control flow is interrupted and a *Handler* named *tiltHandler* is called. The Rising Transition happens when the doorknob is rotated. When the Handler has completed its execution the control flow returns where it was before the interrupt.

It is worth noting that integer numbers are used to check analog sensors values (instead of float), like in the lightVal condition. This is because float values read from analogic pins are converted in integers mapped in the range 0-1023. Further details will be found in the semantic mapping of Sensors.

## 4.2 Basic Arduino Meta-Model

The previous example shows some high level concepts (highlighted in the discussion that follows the example), but also that an Arduino program can basically be modeled using lower level abstractions (based on C++), like:

- Set and Get of a Variable: a pin can also be intended as a variable, so providing this concept at the meta-model level, we have covered all the interactions with external devices and with data.

- Procedure Calling: Arduino is programmed in C++, so a basic construct that it expresses is the Procedure.

- Delay: one of the main concept in Arduino loops are delays. They are used to suspend the control flow for a specific amount of time.

- Basic Control Structures: control structures like *if* and *while*. The structure *for* can be derived from the *while*.

- Class instantiation: this can be optional because all the needed behaviours could be modeled also using basic procedures.

This can be a basic meta-model, but it is really close to the Arduino programming level.

The purpose of this thesis is not the definition of such a low level meta-model, but the creation of an higher level meta-model that captures the concepts identified in the example above. Then it can be mapped (with a Model to Model tranformation) on the basic meta-model.

This process can asymptotically lead to the definition of high level meta-models, specific of certain domains, that can then be mapped on the meta-model defined in this thesis.
So for example we can think about the definition of a meta-model for the Domotic Domain. The main entities could be the fridge, the oven and so on. A model, instance of such a meta-model, can then be mapped with a Model to Model transformation on the meta-model defined in this thesis, or on other lower level meta-models.
The aim of this process is the creation of a hierarchy of meta-models, one for each needed level of abstraction.

In the next sections we will describe the metamodel introduced in this work and define it through a DSL.

## 4.3 DSL Syntactic Definition

Arduino Meta-Model has been defined through a DSL using XText 2.3.
The entry point rule of the DSL is *Sketch*. This rule defines the meta-model of a Sketch (this is how an Arduino program is called) and its definition has been splitted in the three dimensions Structure, Behaviour and Interaction.

```
Sketch :
    'Sketch' name=ID
    'Hardware:' hardware=('Arduino UNO' | 'Arduino MEGA
    2560' | 'Arduino FIO'|'Arduino MEGA ADK' | 'Arduino
    PRO' | 'Arduino NANO'|'Arduino BT' | 'Arduino MINI' |
    'Wiring S')

    // Structure
    (devices+=AbstractDevice)*
    (handlers+=Handler)*
    (tasks+=Task)*

    // Behaviour
    (interrupts+=Interrupt)*
    (pollings+=Poll)*
    ('execution cycle:' (loop+=LoopItem)* ';')?

    //Interaction

;
```

### 4.3.1 Structure

The *Structure* defines the structure of an Arduino program and it is composed by: Devices, Handlers and Tasks.

#### Devices

An AbstractDevice represents a pin of the board and can be either a Sensor, an Actuator or an I/O Device.
Let's define them:

```
AbstractDevice:
    Sensor | Actuator | IODevice
;

Actuator:
    'Actuator' name=ID 'pin' pin=STRING ';'
;

Sensor:
    'Sensor' name=ID 'pin' pin=STRING (analog?='analog')?
    (pullup?='pullup')? ';'
```

```
;

IODevice:
    'I/O Device' name=ID 'pin' pin=STRING
    (analog?='analog')? (pullup?='pullup')? ';'
;
```

Actuators, Sensors and IODevices all have a name and a pin number. An Actuator represents a pin set as Output, a Sensor represents a pin set as Input and an IODevice is a pin that can change its direction (Input or Output) at runtime.

A Sensor can be either Digital or Analog (Arduino provides A/D Converters for some pins) and it can be pulled up.

An IODevice is modeled as a Sensor and can be analogical or pulledup.

The *pin* attribute is not defined as an Integer, but as a String, because some pins are expressed with the name *Ax* where the 'A' refers to the fact that they are part of the Analogic pins of the board, while 'x' express the number. In Arduino UNO we find 6 Analog pins, from 'A0' to 'A5'. A Sensor or IODevice can be defined as attached to a pin of the kind 'Ax', but with the attribute *analog* set to *false*. This means that the device is read as digital, even if linked to an analogical pin.

### Handler

An Handler is a function executed to take care of certain events. It is identified by a name and can use other Devices to keep its own decisions or act on the environment.

It is defined in the meta-model as an entity with the name as a unique attribute:

```
Handler:
    'Handler' name=ID
    ';'
;
```

It expresses the reactive part of an Arduino program, that is, the part that is activated by a raised event.

### Task

A Task is intended as a sequential operation executed in the Arduino *Loop* function in a well defined position of the loop. As we already said, an Arduino program is characterized by the cyclic execution of a set of operation. A task is thought to be the representation of such operations.

This is how the Proactive part of an Arduino program is modeled. The proactive part of a program can be also modeled through Procedure Calling, but it is a too basic way. For this reason, to enhance a modular organization of the application, the concept of Task has been introduced as a first class abstraction in the Arduino concept space.

It is identified by a name and can use Sensors, Actuators and IODevices to interact with the environment. It is defined as a set of actions executed if some preconditions are met or in each cycle of the loop.

A Task can be defined as follows:

```
Task:
  'Task' name=ID (external?='external')?
  ';'
;
```

The *external* attribute refers to the fact that a Task can also be defined in other boards of the System. A deeper description of this feature is in Chapter 5, where the abstracion is raised at the System level.

## 4.3.2 Behaviour

The *Behaviour* defines several different concepts:

- Interrupt: interrupt represents the interrupt concept that follows from Embedded Systems theory, that is, a mechanism that starts a predefined function when an hardware event is perceived. The function executed is of the type *Handler* and its execution stops the main control flow of the program, restoring it when *Handler* completes.

- Poll: this is another way to check the value of a particular sensor. A polling function reads the value of a sensor once for each cycle of the *Loop*.
- Loop: this represent the *loop* cycle introduced by the platform.

In this work the behaviour is conceived as the specification of how each component of the structure behaves in reaction to events or during the execution of the process.

### Interrupt

The first concept we define is *Interrupt*:

```
Interrupt:
    'Interrupt' name=ID 'kind' interruptKind = ('External' |
    'PinChange') 'on' sensor=[Sensor]
    'event kind' eventKind=('Change' | 'Rising' | 'Falling')
    'handled by' handler=[Handler]
    ';'
;
```

An Interrupt can be of two types: External or Pinchange. An External interrupt has a higher priority than PinChange interrupts, but it is supported only on a few pins of the board. PinChange interrupt is intead supported on each pin of the board.

The Interrupt is raised by a Sensor when a particular event happens. This event can be of three types: Change (0 -> 1 or 1 -> 0), Rising (0 -> 1) or Falling (1 -> 0). When the event is perceived, an handler has to be executed.

### Polling

Another important concept found in the Domotic example is *Polling.*

```
Poll:
    'Poll' sensor=[Sensor]
  'if' (type='CHANGE' | type='RISING' | type='FALLING' |
  type='BETWEEN' l=INT '&' h=INT)
  'call' handler=[Handler]
  ';'
;
```

A Polling can be executed on a particular Sensor to investigate its value and consequently decide if a particular event is happened. An event can be of the kinds: Change, Rising, Falling and Between two thresholds. The last event type concerns only analogical Sensors and can not be captured by previously defined Interrupts. This is one of the main reason that cause the introduction of Pollings.

If Interrupts could also capture events on Analogic sensors, probably that would be no need of pollings. Furthermore pollings generates continuous wordload for the microcontroller, decreasing the battery life, that is a key feature for these systems.

When an event is perceived it is handled by a predefined Handler.

### Loop

To model which Task has to be executed and in which position of the cycle, a new abstraction called *LoopItem* has been introduced. A *LoopItem* is expressed in the DSL from:

```
LoopItem:
    'exec task' task=([Task]) ('preconditions:{'
    (precondition = Precondition) '}')?
;
```

A *LoopItem* declares indeed each Task to be executed in the loop and the preconditions to be satisfied for each one. Preconditions can be: a check on the value of a Sensor or an empty condition. They can be a conjunction or a disjunction of conditions.

Preconditions are defined in the Meta-Model as:

```
Precondition:
     pre1=Precondition1 op=('&&'|'||') pre=Precondition |
     pre1=Precondition1
;

Precondition1:
     (pre=SensorValuePrecondition | pre=EmptyPrecondition)
```

```
;

EmptyPrecondition:
     name=ID
;

SensorValuePrecondition:
     sensor=[Sensor] cond=('==' | '<=' | '>=' | '!=')
     value=PossibleValues
;

PossibleValues:
     Double | INT
;
```

Empty preconditions becomes part of the Application Logic.

### 4.3.3 Interaction

Wiring is a C++ based framework, so Interaction between entities of the program is expressed by the classical interactions provided by Object-Oriented programming languages. In this languages an object can communicate with other objects through procedure calling or shared memory.
The interaction between the components declared in the structure part of the meta-model has not been brought to the meta-model level because the system is concentrated on a unique platform and each component shares memory with all the others. This means that it is possible to share data and also call procedures from one object to another.

So the interaction part is left to the Application Designer.

Now that all the concepts have been introduced we can pass to the description of their mapping on the Wiring framework.

## 4.4 Semantic Mapping

To give a formal semantic to the defined Meta-Model, we need to map each concept on an already formally defined entity. The mapping is done with reference to Wiring because its Semantic is already formally defined.

Wiring uses C++ as a reference languange, so we can take advantage of all its Object-Oriented features.

### 4.4.1 Structure

#### Sketch

A *Sketch* is translated into the entry point of a Wiring program, that is, a file with extension *.ino* and with the appropriate name (specifiend in the model) that declares the two main functions of the framework: *Setup* and *Loop*.

#### Sensor

What is meant with the concept of *Sensor* is a pin configured as an Input and that can be Digital or Analogical. To configure a pin as input or output Wiring provides the function:

```
void pinMode(uint8_t pin, uint8_t mode)
```

where *mode* can be the defined constants *INPUT*, *OUTPUT* or *INPUT_PULLUP*. At the beginning of page 34 we can see an example of the definition of three input pins. A Sensor has indeed to be set as an INPUT. It is possible to enable an internal 20KOhm pullup resistor configuring a pin as an INPUT_PULLUP. Writing a LOW value on the pin will disable the pullup. An input pin can be read as Digital or Analogical. Arduino boards provide well defined Analog pins where Analogical inputs are converted into Digital integer values in the range 0-1023. The functions used to read from a pin are:

```
int digitalRead(uint8_t pin) // Read a digital value
int analogRead(uint8_t pin)  // Read an analog value
```

We now have all the tools we need to define a Sensor class and all its properties. The class is implemented in an header file that will look like:

```
class Sensor{
  protected:
    uint8_t pin;       // Pin number
    boolean analog;    // Analog or Digital?
    boolean pulledUp;  // True if pullup resistor enabled
  public:
    Sensor(uint8_t p, boolean pullup, boolean an);
    int getValue();    // Get the pin value (Dig or Analog)
    void setAnalog(boolean a); // Change the pin analog
                               // configuration
    void pullup();
    void disablePullup();
};
```

Only certain pins of Arduino boards provide A/D conversion and they can be read either as Digital or Analogical. For this reason has been introduced the function *setAnalog* to change at runtime the way the pin is read.

The pullup resistor can be also enabled or disabled at runtime respectively with the functions *pullup* and *disablePullup*.

The implementation of each method can be found in the same header file.

### Actuator

The semantic of the concept Actuator is: a pin configured as an OUTPUT and that can be used to write a particular value.

The function *pinMode* allows to configure a pin as an output, like what we did for Sensors but changing the *mode* value.
To write a value on a pin two functions are provided:

```
void digitalWrite(uint8_t pin, uint8_t value);
void analogWrite(uint8_t pin, int value);
```

The function *digitalWrite* sets the pin *pin* to *value*, either 0 or 1.

Arduino does not support analogical output, but it allows to write an analog value (PWM wave) to a pin. Calling *analogWrite*, the pin will generate a steady square wave of the specified duty cycle until the next call to *analogWrite* (or a call to *digitalRead* or *digitalWrite* on the same pin). The frequency of the PWM signal is approximately 490 Hz. The range of value accepted from *analogWrite* is from 0 to 255. Figure 4.2 shows some examples of PWM signals.

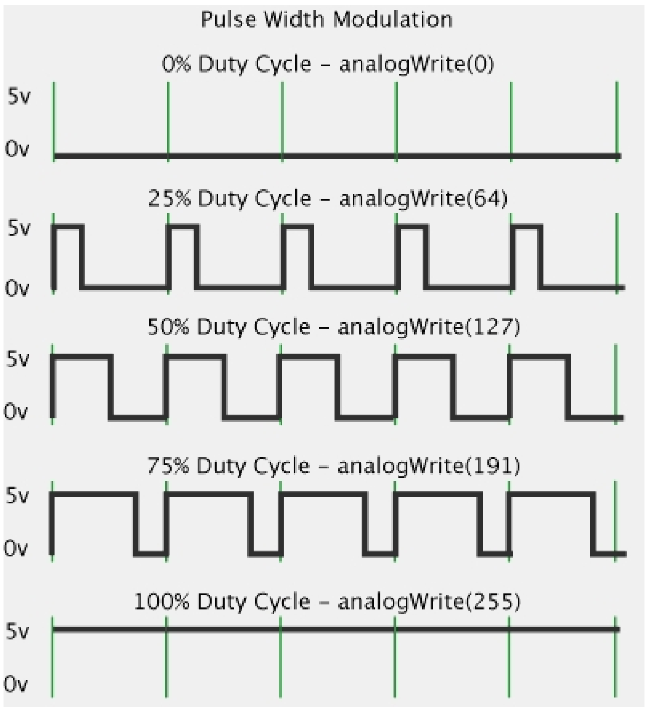

Pulse Width Modulation
0% Duty Cycle - analogWrite(0)
25% Duty Cycle - analogWrite(64)
50% Duty Cycle - analogWrite(127)
75% Duty Cycle - analogWrite(191)
100% Duty Cycle - analogWrite(255)
5v
0v

Figure 4.2: PWM Samples

The header file that defines an Actuator is:

```
class Actuator{
  protected:
    uint8_t pin;     // Pin number
  public:
    Actuator(uint8_t p);
    void writeDigitalVal(uint8_t val);
    void writeAnalogVal(uint8_t val);
};
```

The introduced methods are self explained by their names: *writeDigitalVal* simply sets *val* as digital output, while *writeAnalogVal* sets *val* as PWM output.

### IODevice

In this Meta-Model with IODevice is intended a pin that can change its mode configuration at runtime, that is, it can pass from Input to Output and viceversa during the execution of the program.
An IODevice can be indeed either a Sensor or an Actuator. Using the mechanism of Multiple Inheritance allowed by C++, we can define an appropriate hierarchy to reuse what we already have.

The implementation of the header file that defines an IODevice is:

```
class IODevice: public Sensor, public Actuator{
  private:
    boolean currentlyInput;
  public:
    IODevice(uint8_t pin, boolean p, boolean a):
    Sensor(pin,p,a), Actuator(pin){currentlyInput=false;};

    void writeDigitalVal(uint8_t val);
    void writeAnalogVal(uint8_t val);
    int getValue();
    void pullup();
    void disablePullup();
};
```

It inherits all the methods and properties defined in the father classes and redefines them to take care of the actual configuration of the pin. For example, if the pin is set as Output and the method getValue is called, it has first to be reconfigured as Input and then read.

### System Resources

To have a complete access to each Structural Resource of the System (i.e. Sensors, Actuators and IODevices) it can be useful to have a place where all those resources are instantiated and can be retreived. For this reason an header file called *SystemResources.h* is implemented and it instantiates all the resources of the system providing also a *getter* for each one of them.

### Task

A Task is defined as an operation to do at a certain point of the *Loop*. To further modularize the architecture of a program each task is mapped on a class, so to encapsulate in it its behaviour. A super class called *Task* has been implemented, containing the name and an abstract method *doJob*. A defined task will extend the class Task and implement the method *doJob* where the Application Designer will write the Business Logic of the task.

The definition of the superclass Task follows:

```
class Task{
  private:
    String name;
  public:
    Task(String n){
      name=n;
    }
    // Abstract method doJob
    virtual void doJob();
    String getName(){return name;}
};
```

A method *getName* is already implemented in this class.

Tasks can access all the Devices needed as if they were declared as global variables. Actually Devices are not global, but can be used by tasks through the mechanism of the *#include* directive. Tasks may need Devices (Sensors, Actuators or IODevices) to perform their job and to take all the proper decisions

A defined task will be generated as a new class that extends the class Task:

```
class <SpecificTask> : public Task{
  public:
    <SpecificTask>(String name):Task(name){}
    void doJob();

};
```

The Application Designer has to implement the method *doJob* of each defined task.

### Handler

An Handler is a function called when an event is perceived. It can be called as an Interrupt routine or from a Polling.

The Application Designer can, in this way, separate the Business Logic related to the pure reactive behaviour of the system from the one related to the normal cyclic behaviour.

As for Tasks, Handlers can access all the Devices needed as if they were declared as global variables, simply calling them with their name.

The Semantic Mapping of a Handler in Wiring involves several steps. As already seen for Tasks, a specific handler extends a base class called *Handler* and defined as:

```
class Handler{
  private:
    String name;
  public:
    Handler(String n){name=n;}
    virtual void doJob(){}
};
```

A handler will then extend the previous class and implement the method *doJob*:

```
class <SpecificHandler> : public Handler{
  public:
    <SpecificHandler>(String name):Handler(name){}
    void doJob();

};
```

In the *doJob* function the Application Designer will write the Business Logic of the Handler.

### 4.4.2 Behaviour

#### Interrupt

Arduino supports two kinds of Interrupt: External and PinChange Interrupt. To turn interrupts ON and OFF Wiring implements the commands:

```
void sei();   // Turn ON Interrupts
void cli();   // Turno OFF Interrupts
```

By default when an Interrupt is perceived, a *cli* command is executed so the board becomes "blind" to external events. Nested interrupts are indeed not supported. This configuration can be of course changed.

External Interrupts are supported only by few pins of the board (e.g. pin 2 and 3 in Arduino UNO) and they have the highest priority. The routines called by this kind of interrupts can be installed with the avrgcc preprocessor macro "ISR". Specifying the name of the interrupt vector it is possible to install the appropriate routine in this way:

```
// Install the interrupt routine.
ISR(INTX_vect) {   // X stands for the number
                   // of vector (INT0 or INT1)
  // Insert here the Interrupt Routine
}
```

To declare which event has to trigger the interrupt we have to set the proper bits on register MCUCR. External Interrupts can be triggered on events Change, Rising, Falling or Low. In Arduino UNO there are two bits dedicated in MCUCR respectively in position 0 and 1. This is how events are mapped on those bits:

| | | |
|---|---|---|
| 0 | 0 | low |
| 0 | 1 | change |
| 1 | 0 | falling slope |
| 1 | 1 | rising slope |

To enable interrupts the proper bit has to be set in the register GICR (General Interrupt Control Register): bit 7 for INT1 and bit 6 for

INT0.
Now everything is set to react to particular events.
Actually there is no need to manually act on registers because Wiring provides a simple operation to manage External Interrupts and install the right handler routine. The function is:

```
void attachInterrupt(uint8_t int, void (*)(void) routine,
                          int mode);
```

The variable *int* is the number of the interrupt (i.e. for Arduino UNO it is 0 for interrupts on pin 2 and 1 for the ones on pin 3). The function that has to be executed when interrupt is triggered is specified with parameter *routine*, while *mode* selects the kind of event. It can be expressed with the defined constants: *RISING*, *FALLING*,*CHANGE* or *LOW*.

PinChange interrupts are instead available on each pin of the board, but they have less priority than External ones. To enable these interrupts we can use the PinChange library, available on Arduino website.
Using the library we can set proper routines in a really simple manner. The only static function used is:

```
void attachInterrupt(uint8_t pin, PCIntvoidFuncPtr
                userFunc, int mode);
```

where *pin* is the pin where the interrupt is enabled, *userFunc* is the routine and *mode* is the kind of event (RISING, FALLING or CHANGE).

The main problem of interrupts is that electronics components that generate them are not ideal. This means that e.g. when a component generates a transition from 0 to 1, it is not istantaneous, but may be affected by Bouncing problems. So, these bouncings, can lead to multiple execution of interrupt handlers. To solve this problem, when an interrupt is received, the system checks if that interrupt has been received less than 10 milliseconds before; if not, interrupts are disabled, the handler is called and, when it completes, interrupts are re-enabled. If the same interrupt was received less than 10 milliseconds before, the system simply does not execute the handler.
The timeout has been pragmatically determined after several tests.

### Polling

Another way to check the current value of a pin is through Polling. It involves the reading of the value of a pin once for each *Loop* cycle and execute an handler in case of a certain event.

Pollings can intercept more events than interrupts because interrupts for analogical values are not supported. For this reason, in the Polling entity of the Meta-Model, the event *between* two values has also been defined.

In the generated code the order in which pollings are executed is the same of pollings declaration in the Model. Pollings are mapped on procedures that check the value of a Sensor and, if an event happened, call the proper handler *doJob* method.

In the *loop* function pollings are executed before tasks.

### Loop

The *execution cycle* defined in the meta-model is a ordered set of tasks, optionally with preconditions.

It is mapped on a set of calling to the method *doJob* of the specific task, specified as a *LoopItem*. If the *LoopItem* also declares preconditions, the generated code becomes like:

```
void loop(){
  if(<list of preconditions>)<Specific Task>.doJob();
}
```

The list of preconditions is a conjunction or disjunction of conditions on Sensors, or empty preconditions mapped on procedures to be implemented by the Application Designer.

Preconditions are optional. If they are not specified in the model, in the code we will only find the calling to the *doJob* method.

## 4.5 Domotic Example Model

Now that the DSL that describes a Meta-Model of Wiring has been introduced, we can provide a model of the previous domotic example.

Reminding that a Model is an instance of a Meta-Model, we can express the model through the DSL implemented. So the Model will be expressed by:

```
Sketch FirstDomotic Hardware: Arduino UNO

// Sensors Declaration
Sensor Thermistor pin "A0" analog ;
Sensor PhotoResistance pin "A3" analog ;
Sensor Tilt pin "2" ;

// Actuators Declaration
Actuator redLED pin "A5" ;
Actuator greenLED pin "A4";

// Handlers Instances (One for each event to manage)
Handler tiltHandler ;
Handler gettingDark ;
Handler sunIsShining ;
Handler tempOver30 ;
Handler tempUnder30 ;

// Attach event Rising of sensor Tilt to handler tiltHandler
Interrupt tiltInterrupt kind External on Tilt
event kind RISING handled by tiltHandler;

// Pollings declaration
Poll PhotoResistance if BETWEEN 550 & 1023 call sunIsShining;
Poll PhotoResistance if BETWEEN 0 & 549 call gettingDark;
Poll Thermistor if BETWEEN 164 & 1023 call tempOver30;
Poll Thermistor if BETWEEN 0 & 163 call tempUnder30;
```

Once the model has been implemented, it will automatically be translated into the skeleton of the application, leaving to the Application Desinger the implementation of the business logic. In this example what he/she should do is creating a file called *handlersImplementation.h* and then implement the *doJob* method of each Handler:

```
void gettingDark::doJob(){
  greenLED.writeDigitalVal(1);
}

void sunIsShining::doJob(){
  greenLED.writeDigitalVal(0);
}

void tempOver30::doJob(){
  redLED.writeDigitalVal(1);
}

void tempUnder30::doJob(){
  redLED.writeDigitalVal(0);
}

void tiltHandler::doJob(){
  int i;
  // Blink both LEDs three times
  for(i=0;i<3;i++){
    greenLED.writeDigitalVal(1);
    redLED.writeDigitalVal(1);
    greenLED.writeDigitalVal(0);
    redLED.writeDigitalVal(0);
  }
}
```

Each procedure specifies the business logic of each handler, in fact when an event happens, the proper handler *doJob* method is executed.

From the Software Engineering viewpoint we have a much more modular view of the whole system and the translation of concepts identified in the Problem Analysis becomes immediate.

When a model has been specified, what is left to do for the Application Designer is the implementation of the business logics of: Handlers, Tasks and empty preconditions of tasks.

# Chapter 5

# Towards a System perspective

In the previous chapters we focused on the definition of the concepts that characterize a microcontroller (based on Wiring) considered as an isolated entity. In the following sections we will go up in abstraction and identify some of the features of Systems composed by a set of entities. We will then integrate these features in the DSL.

There are many definitions of *Software System*. In this work we define a *Software System* as a set of interacting heterogeneous parts organized to constitute a "whole" whose properties are not directly attributable to the sum of the properties of each part.

The introducion of the concepts related to a Software System is actually not related to the meta-model of Arduino introduced in the previous chapter. It is instead part of the traditional model of a distributed system. The necessity of this concepts comes from a series of application domains that require the introduction of multiple Arduino systems, like for example the control of some key features in farms or factories. When the complexity of the system grows the developer can decide to use more than one Arduino, possibly coupled with other kinds of elaboration units.

For this reason the concepts introduced are not only related to the Arduino field, but will try to capture all the features that characterize a distributed system in general.

In the definition of the Meta-Model, a SW System is considered as seen from the Embedded System viewpoint. As part of the SW System, each Arduino board is interested in *Who* it is interacting with and *How* it can interact with it.

Observing the SW System from one component viewpoint we can define in order:

- Who is the component and which communication supports it provides.
- Which messages the component needs to exchange.
- How those messages are exchanged.

It is worth noting that each component does not need to know *What* is the component it is interacting with. This derives from the definition of SW System provided before, where it is emphasized the concept of heterogeneity of the parts.

To define all this concepts in the Meta-Model it is necessary the introduction of a new part in the sketch definition:

```
// System Definition
('define System{'
  systemDefinition=SystemDefinition
'}')?
```

Defining then the rule *SystemDefinition*:

```
SystemDefinition:
  'Communication Parameters:' mydata+=CommunicationParams
    (',' mydata+=CommunicationParams)* ';'

  (messages+=Message)*
  ( operation+=HighLevelOperation)*
;
```

It is an optional part and defines in order: which supports it provides for the communication, which kind of messages it will exchange and how they will be exchanged.

Of course the definition of all previous properties is optional, meaning that the embedded system does not need to be integrated in a SW

System, but it can stand by itself and carry on its own functionalities. From here on, the word *system* is used with the meaning of a single Arduino board, while the word *System* means a set of distributed interacting parts.

## 5.1 Provided Supports

The *CommunicationParams* rule defines which supports the component will provide and their main configuration parameters.

An embedded system like Arduino can have a wide range of communication supports. By now we only consider rules to define the Ethernet and Serial parameters.

The rule is described with the following specification:

```
CommunicationParams:
    type="Ethernet" 'mac=' mac=STRING ('ip='ip=STRING
    ('dns='dns=STRING ('gateway='gateway=STRING
    ('subnet='subnet=STRING)?)?)?)?  | type="Serial"
    'baudrate=' baudrate=INT
;
```

Ethernet is characterized by a MAC number and optionally by an IP, DNS, Gateway and a Subnet. Each one of them can be specified in the model only if the previous has been already defined.

A Serial has only one parameter that describes the number of symbols transmitted per second, also known as *Baudrate*.

## 5.2 Structure of the System

Each component of the System does not need to know the structure of the whole System and each entity that constitutes it.

As we already said before, in this Meta-Model the *Structure* of the System is defined from the component viewpoint and in this sense each component only needs to know which are the entities it has to communicate with.

The entities that interacts within the System can be identified from the same concept of Task introduced in the Arduino meta-model. For this reason they incapsulate the attribute *external*, meaning that the Task is not in the system defined by the model, but on another board.

The interaction part of the Meta-Model (messages and operations) has been taken by the DSL named *Contact* introduced in the course *Software Systems Engineering*. A deeper overview can be found in [11]. This is a further proof of how a platform independent meta-model can be applied to every specific platform that supports some of the protocols introduced.

We can say that a Task represents the same concept of a *Subject* in Contact and *HighLevelOperations* is the way tasks communicate each other.

## 5.3 Messages

In this section it will be explained which kind of messages have been introduced to exchange information between the parts of the System and their semantic.

As already said, defined messages are inspired by *Contact*.

In this work messages are divided into two main families: OutOnlyMessages and OutInMessages.

OutOnlyMessages can only be sent and no answer is expected. OutInMessages are instead messages that are sent, but that expect an answer.
All the defined messages are intended as asynchronous, in the sense that, when a message is sent, the sender does not have to wait for a reply, even for OutInMessages where the sender sends a message and than can check if the counterpart has sent an answer, but always without the need of remain blocked.

In the DSL, messages are defined in this way:

```
Message: OutOnlyMessage | OutInMessage  ;

OutOnlyMessage :   Dispatch ;
OutInMessage:     Request | Invitation ;

Dispatch:   "Dispatch" name=ID   ";";
Request:   "Request" name=ID   ";";
```

```
Invitation:  "Invitation" name=ID  ";" ;
```

Each message has a name that tries to capture some application level communication features. In this way the Application Designer can refer to different semantics mapped on proper messages. In the DSL each message is characterized only by a name because all the information that describe the exchange phase are specified in the *HighLevelOperation* rule.

The introduction of three different messages needs a brief explanation, even though not in a formal language.

- Dispatch: it is a message that one can *forward* to one specific receiver, with the expectation that it will *serve* it

- Invitation: an invitation is a message that an entity can *ask* to one or more receivers with the expectation to acquire zero or more *ack*. The receiver can *accept* the invitation and send back an acknowledge.

- Request: a Request is a message that one can *demand* to one or more receivers with the expectation to acquire zero or more *response*. The receiver can *grant* the request and send back the response.

Of course more kind of messages could have been modeled, but, for the purpose of the thesis, the previous set of messages is enough.

## 5.4 High Level Operation

The exchange of messages is described in the DSL by the rule *HighLevelOperation*. This rule says that, from the Arduino system viewpoint, there are, as expected, two directions of exchange: in and out.

```
HighLevelOperation:
  OutOperation | InOperation
;
```

*InOperation* will characterize how to receive all the incoming messages, while *OutOperation* describes how to send all outgoing messages. Let's start from the first rule:

```
InOperation:
  InAcquireOperation
;

InAcquireOperation :
  ServeDispatch |  GrantRequest | AcceptInvitation
;

/*  DISPATCH */
ServeDispatch:
    receiver=[Task] "serve"  serve=[Dispatch]
    ("support="  support=SupportSpecification)? ";"
;

/* REQUEST */
GrantRequest:
  receiver=[Task] "grant" grant=[Request]
  ("support=" support=SupportSpecification)? ";"
;

/*  INVITATION */
AcceptInvitation:
  receiver=[Task] "accept" accept=[Invitation]
  ("support="  support=SupportSpecification)? ";"
;
```

In InOperation we find the description of how to receive each possible incoming message and, as said before, they can be of the kinds: Dispatch, Request and Invitation. We also find some keywords highlighted in the former Messages description: a Dispatch has to be *served*, a Request is *granted* and an Invitation is *accepted*.

Each operation can specify a support. This is how the receiver can be reached.

```
SupportSpecification: TCP | Serial;
```

```
Serial: supportType="Serial";
TCP: supportType="TCP" info = SupportData  ;
SupportData : ExplicitSupportData;
ExplicitSupportData : "[" "host=" host=STRING
                          "port=" port=INT "]" ;
```

Of course the SupportSpecification can declare more than two types of support. In Contact we also find UDP, HTTP, etc. In this meta-model the supported protocols are only TCP and Serial, but other supports can be added.

*OutOperation* are thought to declare which messages are sent to which receivers and its definition in the meta-model follows:

```
OutOperation:
   ForwardDispatch | DemandRequest | AskInvitation
;

/* REQUEST */
DemandRequest:
  sender=[Task]
  "demand" demand=[Request] "to"
  receiver+=[Task] ( "," receiver+=[Task] )* ";"
;

/*  DISPATCH */
ForwardDispatch:
    sender=[Task]
    "forward" forward=[Dispatch] "to" receiver=[Task] ";"
;

/*  INVITATION */
AskInvitation:
  sender=[Task]
  "ask" ask=[Invitation] "to"
  receiver+=[Task] (  ","  receiver+=[Task]  )* ";"
;
```

Also for these rules we find some keywords highlighted in the messages description: a Request has to be *demanded*, a Dispatch is *forwarded* and an Invitation is *asked*.

A Request has a sender and one or more receivers, just like the Invitation. The Dispatch instead can have only one receiver.

## 5.5 Semantic Mapping of System Perspective Concepts

Once defined all the concepts to describe the interaction with other entities of the System, they have to be mapped on Wiring, so to formally define them.

Actually the support to communication is introduced in Arduino through special libraries, so it is not a native feature. This can also be deduced by the fact that to support TCP it is needed a new hardware stacked on the Arduino board.

First of all is provided an overview of how Wiring implements the communication supports introduced in the Meta-Model.
In this work we will focus on communication through TCP/IP networks and Serial, even though Arduino supports other means like I2C, XBee or Bluetooth.
Networking in Arduino is supported either on TCP and UDP.

### 5.5.1 TCP Support

TCP communication is based on the concepts of Server and Client and they become fundamental to establish a bi-directional communication channel between two boards.

The Ethernet support in Arduino is provided by the library *Ethernet* and it uses the hardware *Ethernet Shield*, shown in Figure 5.1. The class *Ethernet* has to be initialised through the method *begin*, whose prototypes are:

```
int begin(uint8_t *mac_address);
void begin(uint8_t *mac_address, IPAddress local_ip);
void begin(uint8_t *mac_address, IPAddress local_ip,
            IPAddress dns_server);
void begin(uint8_t *mac_address, IPAddress local_ip,
```

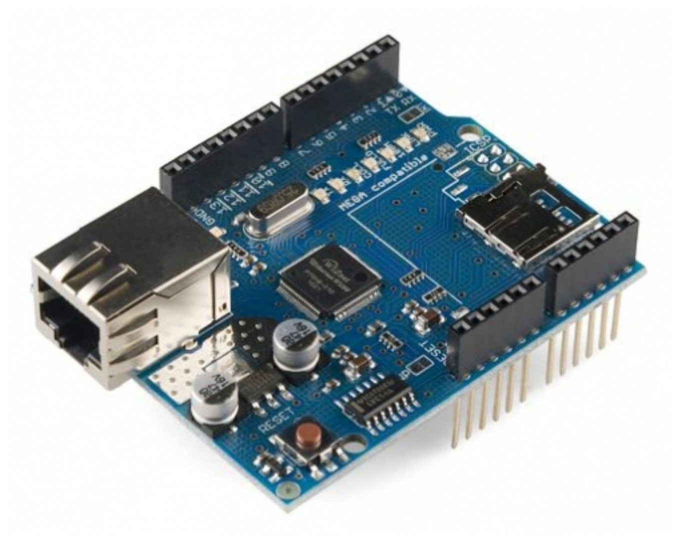


Figure 5.1: Ethernet Shiled

```
            IPAddress dns_server, IPAddress gateway);
void begin(uint8_t *mac_address, IPAddress local_ip,
            IPAddress dns_server, IPAddress gateway,
            IPAddress subnet);
```

Looking to previous prototypes we understand the reason why in the *CommunicationParams* rule of the DSL each attribute can be defined only if the previous was already defined. In fact, each prototype contains a limited number of parameters in the same order defined in the DSL.

### 5.5.2 Server

A Server is a socket used to receive incoming connection. Arduino can handle a maximum of 4 simultaneous connections and does not

provide blocking operations to read data.
A Server is created as an instance of the class *EthernetServer* and started with the method *begin*. The constructor of *EthernetServer* only accept a Port number as parameter:

```
EthernetServer(uint16_t port);
```

To check if data have been received on the Server, Wiring provides the method *Available*. This method returns an *EthernetClient* object that can be used to read or write to the connected client.

A typical scheme followed to use Ethernet is:

```
#include <SPI.h>
#include <Ethernet.h>

byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
byte ip[] = { 10, 0, 0, 177 };
byte gateway[] = { 10, 0, 0, 1 };
byte subnet[] = { 255, 255, 0, 0 };

// Server initialised to port 23
EthernetServer server = EthernetServer(23);

void setup()
{
  // initialize the ethernet device
  Ethernet.begin(mac, ip, gateway, subnet);

  // start listening for clients
  server.begin();
}

void loop()
{
  // if an incoming client connects, there will be
  // bytes available to read:
  EthernetClient client = server.available();
  if (client == true) {
    // read bytes from the incoming client and write
    // them back to all clients connected to the server:
    server.write(client.read());
  }
}
```

An EthernetServer object does not provide the chance to write to a specific connected client. If a write is executed, it sends the specified data to all clients connected to the Server.
It is not even possible to understand which client has sent data just received because EthernetClient class does not implement a method to retreive the IP address of the counterpart.
Another missing feature is a method to get an EthernetClient reference to a just connected Client. If a Client connects to a Server, the EthernetServer object does not have a way to get a reference to an EthernerClient object connected to that Client. It becomes possible only if that Client sends some data to the Server.

All these limits will influence the way TCP communication is implemented.

### 5.5.3 Client

A Client is represented by an instance of the class *EthernetClient*.

EthernetClient can connect to a Server using the method *connect* that can be in the form:

```
int connect(IPAddress ip, uint16_t port);
int connect(const char *host, uint16_t port);
```

The return value indicates success or failure.
When a connection has been established, a data flow can be started through the methods *read* and *write*.

The method *connect* is the only blocking method implemented and it blocks until a connection is established or until a timeout expires. Timeout is hard coded in the EthernetClient library and it is about 30 seconds. I changed this value to 2 seconds, so to leave to the application the control of the desired behaviour.

To check if data are available in the buffer it implements the method *available*, really similar to the one in *EthernetServer*.

A Client socket can be used in each Arduino to connect to other components of the System and to send data to them. As written before, to receive data from other components the program should check the proper Server socket.

It is important to know that Arduino UNO supports only a maximum of four simultaneous connections.

## 5.6 UDP Support

Arduino provides a really limited UDP library. It is only possible to use it to send datagrams to only one destination, and it is not reconfigurable.

For this reason I decided to not map the UDP support in the DSL on this library and leave it to future development.

## 5.7 Serial Support

Arduino supports Serial communication on digital pins 0 and 1, respectively Receiver Pin (RX) and Transmitter Pin (TX).

Of course if those pins are used for serial communication, they can not be used for digital input or output.

The initialisation of Serial is achieved through the method *begin*.

```
void begin(uint16_t baud_count)
```

passing the *Baudrate* as argument.

Arduino implements proper procedures to send and receive data through Serial. They are similar to what we found in the Ethernet library: *print*, *println*, *read*, *available* etc.

Hardware Serial is supported only on digital pins 0 and 1. To fill this lack of serial pins, a library has been implemented. It is called *SoftwareSerial Library* and allows the usage of each pin of the board as a Serial pin. This library introduces some big limitations too. In fact if the system uses more than one serial port, only one at a time can receive data (as we can expect because of the single control flow constraint). Furthermore it makes a massive usage of *Pin Change Interrupts* to be aware of arrived bits. This means that if the

Application Designer decides to use this library, it can not use pin change interrupts at all. And this is quite a big constraint.

For the reasons just explained the serial communication introduced in the Meta-Model will not be mapped on *SoftwareSerial Library*, but only on *Hardware Serial* pins.

This represents a good tradeoff between the reactive behaviour that an Application Designer would like to model and the communication with other entities of the System.

## 5.8 Communication Params Mapping

The first concept mapped on Wiring is the communication parameters list. Ethernet parameters are mapped on the call to the proper method *Ethernet.begin* depending on the number of parameters specified (MAC,IP,DNS,Gateway and Subnet).

Serial support is mapped on the procedure *Serial.begin* with the right Baudrate.

## 5.9 Message Exchange

Each exchange rule is mapped on a procedure with well defined name syntax.

The Xtext syntax is:

```
ProcedureName:
    ReturnType Keyword '_' (Sender | Receiver) '_'
    MessageName ('_' Receiver)? '_' Proto
;
ReturnType:
    'void' | 'String'
;
Keyword:
    DispatchKey | RequestKey | InvitationKey
;
DispatchKey:
    'forward' | 'serve'
```

```
;
RequestKey:
    'demand' | 'grant'
;
InvitationKey:
    'ask' | 'accept'
;
MessageName:
    String
;
Receiver:
    String
;
Sender:
    String
;
Proto:
    'TCP'
}
```

Each procedure can return a void or a String. InOperation always has a String return type, while OutOperation always has a void one. The Keyword depends on the kind of message.

Depending on the direction of the message, it can have a Sender or a Receiver. Out messages will have a Sender, while In messages will have a Receiver. The MessageName is the name of the message defined in the model.

In case of an Out message, a receiver has to be specified. Actually this is true only when there is a unique receiver, like in the Dispatch. Requests and Invitations have multiple receivers, so none of them are reported in the statement of the operation.

The Proto rule describes which protol has been chosen. By now the only supported are TCP and Serial.

The Application Designer can communicate with other Tasks of the System simply calling the right procedure.

So, for example, if the actual Arduino wants to *grant* a request called *request2* from Task *task2* on TCP support, the name of procedure will be:

```
String grant_request2_ent2_TCP()
```

Messages exchanged are mapped on Strings with a well defined syntax too:

```
Content:
    MessageName '(' Sender ',' MessageContent ')'
```

where *MessageName*, *Sender* and *MessageContent* are mapped on Strings.

If an Application Designer wants to forward from Task *MyTask* to Task *Task1* a dispatch with name *dispatch1* with content *HelloWorld* on TCP support he has to call the operation:

```
int forward_MyTask_dispatch1_Task1_TCP("HelloWorld")
```

and the dispatch will contain the String:

```
dispatch1(MyTask,HelloWorld)
```

## 5.10 TCP Mapping

Semantic mapping between messages and TCP depends on the direction of the communication.

For outgoing messages there is only the need of one client socket, that connects to the destination entity and sends the message.
The syntax of client name is:

```
ClientName:
    'client_' Sender '_' MessageName ('_' Counterpart)*
;
```

For incoming messages syntax is defined in the same way, but now we need to declare two objects: a Server socket to receive the message and a Client socket initialized when a message is received.

Both client and server, in case of incoming messages, have the name of the Receiver task instead of the Sender.

Server sockets start listening for messages during the setup phase of the system.

Procedures introduced to get incoming messages are non blocking and always return a String. This String can be empty too, meaning that a message has not arrived yet.

As the reader might have guessed, each message exchanged through TCP is mapped on a separate connection, bringing to a 1:1 mapping between messages and connections. This assumption is motivated by the limited hardware resources and hereinafter it will be explained better.

The original idea was to associate to each task a well defined IP address and port, so the actual Arduino could send every desired message to each entity only on one proper port. This implies that in the receiver entity there should be the way to maintain all the messages received for example in a (perhaps dynamic) Software Buffer or tuple space. There should be one software buffer for each incoming connection and they should store all the messages received. So, if e.g. an entity is waiting for a Response message, it calls the procedure to get the response and this procedure should receive all messages from the ethernet buffer, store them in software buffer and then check if a response has been received. In this way all messages received (that can be more than one and of different kinds) are stored in the buffer and can be retreived from proper procedures implemented in the TCP stub.
If the entity has received more than one message of the same kind, when it will check for that kind of message, it will get all the received messages in the right order.

I have implemented this mechanism, but the result was a really fast Out of Memory. The problem with this kind of embedded systems is that they provide a really limited data memory. Memory provided by Arduino UNO has been introduced in the Hardware overview.

Data can be stored on Flash Memory too, but I decided to do not use it for this purpose because the amount of code that a user may want to generate is initially unknown and using it to store data would have meant introduce a further limitation to programs. Given this limitations, I decided to map 1:1 connections and messages, meaning that an Arduino can exchange a maximum of four different messages with other entities of the System.

Actually a new connection is established when the message has to be sent and then is terminated when the exchanging session is completed. So exchanged messages could be more than four, but only if the communication is designed properly.

To bypass this limit the Application Designer could also map several information on one kind of message, assigning the right semantic at the application level.

## 5.11 Serial Mapping

Serial communication is mapped on Hardware Serial, as said before, keeping the implementation quite simple.

The initialization of Serial support is made in the setup phase of the system, with the Communication Parameters semantic mapping.

Once setup, the only two procedures needed to send data through Serial support are: *print* and *read*.

The convention about procedure names syntax is still valid, so it becomes easy for the Application Designer to identify what a procedure has been implemented for and to use it properly.

## 5.12 Utility

When an entity receives a message it needs to get the proper information from it. For this reason a utility file has been implemented, and some functions are provided:

```
String getMessageType(String Message)
String getMessageSender(String Message)
String getMessageContent(String Message)
```

As suggested by the names of the operations, each function takes a message expressed in the syntax introduced before and returns a part of it: the Type, the Sender and the Content.

# Chapter 6

# Code Generators Overview

The aim of this work is creating a Software Factory for applications in the Embedded Systems domain. Once defined a meta-model and the corresponding DSL through the framework Xtext, we can proceed with the implementation of the *Code Generators.*

The language used for this purpose is Xtend2.

When an Xtext DSL has been defined it is possible to automatically generate all the artifacts (Ecore Java classes) by running the workflow created by the IDE in the file *Generate"Language".mwe2.* The word *Language* has to be changed with the name of the language specified when the Xtext project was created in the IDE. In this case it is *"arduino".*

It is worth noting that the code is generated in Eclipse, but to upload it to Arduino it has to be opened with the Arduino IDE.

In this chapter I will give an overview of the code generators architecture so to give an idea of the roadmap followed and of the motivations behind the decisions taken.

## 6.1 Generators Access Point

After the creation of the DSL and the execution of the workflow, every needed artifact is generated and the syntax driven editor can be started. This editor is actually a new instance of the Eclipse IDE. To create a new model based on the defined meta-model (the DSL), a new Eclipse project has to be created. In the *src* folder of the new project we must create a new file, with extension "*.language_name*", where "language_name" is the name of the language defined when the Xtext project was first set up.

Now a new model can be specified, taking advantage of the syntax driven editor. When the model is saved, the Xtext plugin automatically runs the method *doGenerate* of the class *language_nameGenerator* that represents the access point of the code generator, in my case it is *ArduinoGenerator*. Generated files will be put, by default, in the source folder named *src-gen* that should be manually created in the just instantiated project.
This method has a reference to the Abstrax Syntax Tree (AST) representing the defined model and also a reference to an object (*fsa*) used to access the file system to generate proper code files.

We can now start to navigate the AST and generate all the needed code.

To store the reference to the *fsa* I have implemented a utility class called *GenUtils* that contains also a method to generate a file, given the name, the extension and the content.

The only thing that *doGenerate* does is calling a method called *main* in the class *MainArduinoGenerator*.

## 6.2 MainArduinoGenerator Behaviour

I tried to keep the implemented generators as modular as possible, so to have a simpler view of how they behave.

The method *main* of the class *MainArduinoGenerator* executes in order a list of generators, passing to each one a reference to the root node of the AST and a reference to the utility file. Actual generators are related to the hardware *Arduino UNO*, so they are called only if the specified hardware in the model is that one. The generators called are, in order:

- *MainStructureGenerator*: generates the main structure of the Arduino program, that is, the Sketch file with the *setup* and *loop* functions.
- *BasicLibrariesGenerator*: generates a set of header files that represent the basic libraries of the application. Among them we

find the definition of the class *Sensor*, *Actuator*, *IODevice*, *Handler* and *Task*. Here there is also the definition of the previously introduced file *SystemResources* containing the instantiation of all the devices introduced in the model.

- *HandlerGenerator*: defines all the handlers classes.
- *TaskGenerator*: defines all the tasks classes.
- *CommunicationGenerator*: generates all the communication stubs.
- *UtilityGenerators*: generates the utility class that contains all the methods to process incoming messages.

In the following sections we will have a look at what is inside each generator, giving a brief view of what is the used logic.

### 6.2.1 MainStructureGenerator

This class generates the main structure of the Sketch, starting from the inclusion of the needed header files. To save memory if a file is not needed it is not included. This is the reason why in the code we find several calling to methods with statement similar to *checkIfSomethingIsNeeded*; e.g. the first *#include* list is about Ethernet libraries. They are quite heavy, so if in the model the Application Designer did not specify any Ethernet support, we do not need to import them. This check is done through the function *checkIfStubIsNeeded*, passing the protocol *TCP*. Same check is done for Serial support.

A similar control is done for pollings. The implementation of pollings is on a different header file and, if the model do not specify pollings, it is useless to include that file.

The Application Logic in the generated code is only related to Handlers and Tasks. If the Application Designer declares handlers or tasks in the model, then their logic has to be implemented in the proper file, respectively *handlersImplementation.h* and *tasksImplementation.h.* If no tasks or handlers are found in the model, these files are not included in the Sketch.
These files are, however, not generated automatically. If the user has specified handlers or tasks in the model, the proper file is included in

the Sketch, but not generated. At compile time this will result in a "file missing" error. I made this choice to remember to the user to implement them with the business logic of the application.

In the Sketch file all the tasks are instantiated, so to be called in the loop function and all the supports are initialized.

If some pollings have been defined in the model they are executed through the procedure *execute_pollings* once for each cycle of the loop. Pollings are executed before tasks.

In the loop function it is implemented the loop of tasks defined in the model, with the proper preconditions. Empty preconditions have to be implemented in the file *tasksImplementations* together with the tasks application logic.

### 6.2.2 BasicLibrariesGenerator

This class generates all the basic libraries previously introduced, each one in a different header file.

We also find the instantiation of all the declared devices, with the right parameters passed to the constructor.

### 6.2.3 HandlerGenerator

We saw in the semantic mapping that each handler is implemented as a class that extends the main class Handler and implements the virtual method *doJob*. Here there is the generation of these classes, one for each declared handler.

Handlers' name have to be unique in the model.

### 6.2.4 TaskGenerator

Here we find three functions:

- *generateTasks*: is used to generate every task class that extend the class Task, like what is done for handlers.
- *generateTaskDeclaration*: this is called by the MainStructure-Generator and returns a String with the instantiation of all the tasks declared in the model.

- *generateLoop*: this is where the loop of tasks is generated, taking care of the right set of preconditions.

### 6.2.5 CommunicationGenerator

This is the access point to the generation of all communication stubs. It in fact executes in order *TCPSupportGenerator* and *SerialSupportGenerator*.

These two classes explores the AST checking if some messages are exchanged using the protocol they implement and generate the proper functions in dedicated header files called *Protocol_stub.h*, where "Protocol" is replaced with the protocol implemented.

In this class we also find the implementation of the previously introduced function *checkIfStubIsNeeded*. What it does is exploring all exchanged messages checking if at least one uses the specified stub.

The function *initCommunication* is used to generate the code to initialize the communication supports like Serial and Ethernet.

The class *TCPSupportGenerator* first of all explores all the InOperation declared. For each InOperation, if it declares a TCP ExplicitSupportData, it adds to an hashmap the address and port specified, identified with the key *Receiver_MessageName*. In this way we have a map of all the destination or receiving addresses. In case of an OutOperation, an address in the map is a destination and can be retreived taking each receiver task of the message and the message name, and check if it is contained in the map. In case of an InOperation, an address in the map is interpreted as a receiving address, so the created ServerSocket will have those parameters.

## 6.3 Tests

To check if the generated code is correct and works as expected several tests have been carried out.
The first part is focused on the concentrated part of the code, that is the one running on a unique platform. The second part checks the code generated to manage the communication with other entities of the System.

Starting from the concentrated part:

- the definition of a Device (Actuator or Sensor) has been tested using different type of devices: temperature sensors, light sensors, LEDs, potentiometer, tilt sensor. All the operations to read or write a value works properly.

- both PinChange and External Interrupts have been tested, on several pins of the board. They react to all the event declared in the meta-model and execute the proper handler. There are still some problems related to the bouncing of the generated signals. The solution introduced is a check on the timing of interrupts handlers, that is, if the handler of a certain interrupt has been executed less than a time limit before the actual execution, it means that this handler is run after a bounce. The established time limit is 10 milliseconds, but sometimes it seems to be not enough. One solution can be the introduction of an external hardware debouncing circuit. It is important to remember that in Interrupt handlers the use of *delays* is forbidden.

- Pollings have been defined for all the kinds of event that they can detect, checking each time if the handler was called only when an event happened. From the tests done we can say they works correctly.

- The Loop concept is translated correctly and all the tasks are executed in the same order as they are defined in the model. Also preconditions are translated as expected. Remember that tasks are executed after pollings.

The distributed part have been tested among two Arduino boards and the tests run are:

- exchanging of one or more Dispatch between two Arduino boards.

- exchanging of one or more Requests between two Arduino boards. Of course also the Responses have been checked.

- exchanging of one or more Invitations between two Arduino boards. Of course also the Acks have been checked.

Initial problems were given by the TCP stub and they were related to the management of the connections. It happened that everything was working fine only if the system that runs the server was started first. The problem was solved changing the TCP stub and properly managing all the connections.

The resulting code is very modular and the Application Designer has not to write a big amount of code, even for more complex behaviours. I think that the provided example gives an idea of the amount of code to be written.

All the examples provided in the thesis have been tested too and work as expected.

# Chapter 7

# A Case Study with Multiple Arduino Platforms

In this chapter it will be discussed a distributed use case obtained from an extension of the first domotic example seen before. This example is thought to be a proof of concept of what can be done with the meta-model developed and a tutorial on how to do it.

We will start from the requirements, passing then to the modeling phase and after to the implementation of the business logic of the application.

## 7.1 User Requirements

Suppose the user wants to realize a system based on two Arduino UNO boards, interacting each other. One board is connected mainly to Sensors and sends regularly some messages to the other board whose content is based on the observed values. The second board executes some tasks depending on the messages received.

The first Arduino is called *Domotic1*, while the second is *Domotic2*.

The requirements for the first Arduino can be summarized in the following list:

- When the temperature is over 30°C, send a TCP message to Domotic2 to communicate it is over.

- When the temperature is under 30°C, send a TCP message to Entity2 to communicate it is not over.
- When there is still a good level of light, send a TCP message to Entity2 with the light level.
- When it is getting dark, that is the level of light is under a certain boundary, send a TCP message to Entity2.
- The board should be connected to a potentiometer used to regulate the level of the light emitted by a Lamp (in this case a LED).
- A sensor is connected to the doorknob of the main door, and when it is rotated the board has to send a TCP message to Entity2 to communicate it.
- Domotic1 does not have to send anything, until it receives a signal from Entity2 meaning that a button has been pressed. If the button is pressed again, it has to stop sending messages to Domotic2.

The second Arduino, called *Domotic2*, has the following requirements:

- When the board receives a TCP message, it has to choose from the following situations:
  - if it contains information about the temperature: if it is over 30°C turn ON a red LED, otherwise turn it OFF.
  - if it contains information about the light: if it is getting dark turn ON a green LED, otherwise turn it OFF.
  - if it contains information about the doorknob rotation, blink three times the previous introduced red and green LEDs.
- When a button is pressed notify it to the Domotic1, so it can start or stop its work. In fact the button can either start or stop the system, depending on its actual state.

## 7.2 Brief Requirements Analysis

In this chapter we will not follow all the steps that normally characterize the Analysis and the Design of a Software because, in this phase, what is relevant to show is an example and a tutorial of how to develop a system based on Arduino boards, focusing on modeling and not on the code.

Reading the requirements for Entity1 we deduce that we will of course need a Temperature and a Light sensor. They will be both analogic sensors.

To cope with the rotation of the doorknob we have to use a Tilt sensor.

Domotic1 and Domotic2 communicate through TCP, so an Ethernet shield has to be stacked on the Arduinos we are using.

The messages exchanged with Domotic2 have to be mapped on the messages defined in the meta-model. I decided to use three Dispatches called: *manageTemperature*, *manageLight* and *manageTilt*. The information about each sensor will be mapped on those messages, respectively information about temperature, light and tilt. The content of the messages will not be the value read from the sensor, but a string that expresses the actual situation of the observed environment. In this way the policies about the actual value and boundaries are kept in the Domotic1 side.

So the content of messages sent will be:

- for the temperature the board will send the string ”over” if the temperature is over 30 degree and ”under” otherwise.
- the light level will be expressed by the string ”sunny” if it is over a defined boundary and ”dark” otherwise.
- the activation of the tilt sensor is expressed with the string ”true” and it will be ”false” otherwise.

To be signaled when the button is pressed on Domotic2, one pin of Domotic2 must be connected to a pin on Domotic1. In this way Domotic2 can emit a signal when the button is pressed.

A brief discussion on analog boundaries is needed because we have to decide which is the bounday value for temperature and light.

Arduino accepts values from 0 to 5V and maps them on integers in the range 0 - 1023. So what should be done is deciding an analog limit in the 0-5V range and then map it in the range 0-1023.

All the sensors, except the tilt one, have to be read periodically, so we will use pollings to read them and generate proper events. Tilt is instead managed with an interrupt; when it goes to 1, it raises an interrupt handled by a defined handler.

Each handler has to manage the events generated by pollings and interrupts and take the right decision on the basis of the actual value read. The potentiometer handler, called when its value changes, only has to change the intensity of a red Led, using analogic output (PWM).

Since there are three messages to be exchanged, I decided to create three tasks on both Domotic1 and Domotic2, respectively to send data and to receive them and act properly.

To avoid a flooding of messages from Domotic1 to Domotic2 I decided to define a task called *wait* that will contain a wait of 500ms.

The behaviour of Domotic2 is organized in four tasks:

- three tasks defined to receive incoming messages about temperature, light and tilt.
- a wait task of 50ms because it is useless to check continuously for new messages because Domotic1 sends a maximum of three messages every 500ms. This also prevents an high usage of batteries.

The button is managed through an interrupt. When it is pressed an interrupt is raised and the handler sends the defined signal to Domotic1. This signal is simply an output of an high digital value that will be read from Domotic1.

## 7.3 Hardware Setup

After a fast analysis, we found what kind of hardware we need and we can now proceed with its setup.

A scheme of the hardware setup is in Figure 7.1. Domotic1 is the top board, while Domotic2 is the bottom board. Both are represented by an Ethernet Shield because they have to exchange TCP messages;

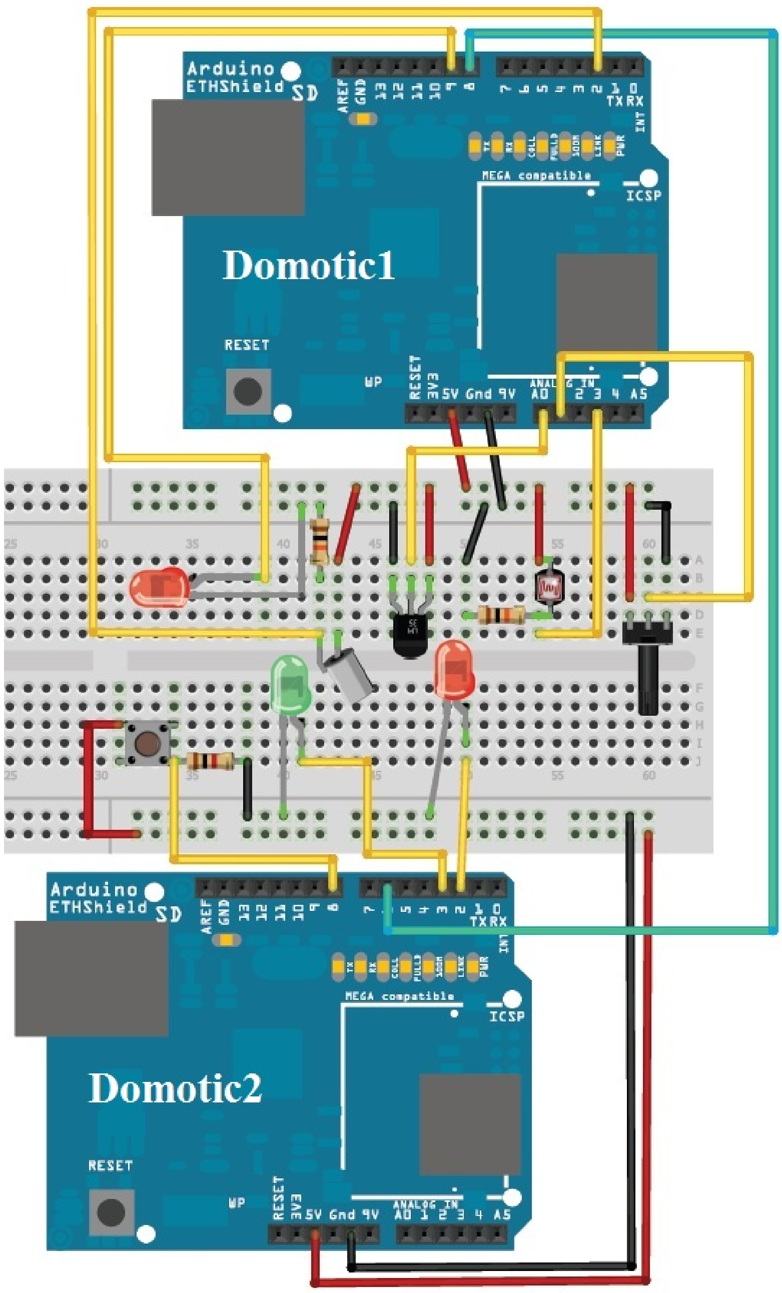


Figure 7.1: Hardware Setup

of course there should be an Ethernet cable that connects the two shields. To maintain the picture clear I have not drawn it, but it is required .

The semantic of the cables' color is: black for the ground connection, red for the connection to 5V and yellow for the cables that goes from devices to pins. The green cable delivers the Start signal from Domotic2 to Domotic1.

Let's start from the description of Domotic1. From the right to the left we find:

- the potentiometer: it is a 10KOhm potentiometer and it is connected to pin A1.
- the photoresistance: it has a 10KOhm resistance connected in series to get the correct value of the potential. It is connected to pin A3.
- the analog temperature sensor: it is connected to the pin A0
- the tilt sensor is connected to digital pin 2 because it is not analogical. Actually the tilt sensor used in my tests is a double angle sensor with three pins and is connected to a resistor to have a rising edge when it is rotated of an angle greater than 45°. In the tool to realize the hardware picture, the tilt sensor used is not available.
- the last component on the left is a red LED. It is connected to pin 9 (that supports PWM) and is the LED whose intensity is set by the potentiometer.

Domotic2 is only connected to two LEDs and a button. A red LED is connected to pin 2, while a green LED is on pin 3. The button is in series with a resistance to get a rising edge when it is pressed and is on pin 8.

## 7.4 Domotic1 Model

Now that we have analyzed the problem and setup the hardware we can define the Model of the system, using the defined DSL.

Starting from Domotic1 we can first declare the devices of the system:

```
Sketch Domotic1 Hardware: Arduino UNO
Sensor Thermistor pin "A0" analog ;
Sensor PhotoResistance pin "A3" analog ;
Sensor Tilt pin "2" ;
Sensor Potentiometer pin "A1" analog;
Sensor ButtonPressed pin "8";
Actuator potentiometerLed pin "9";
```

The name of the sketch is ”Domotic1” and runs on an Arduino UNO board. The hardware is composed by the previous defined devices, connected to the pin highlited in the hardware setup. The analogic sensors are defined as *analog*. The sensor *ButtonPressed* actually is not a physical device; it represents an input pin used to read the signal emitted by Domotic2 when the button is pressed.

We can now pass to the definition of the Handlers, Pollings and Interrupts.

```
Handler tiltHandler ;
Handler gettingDark ;
Handler sunIsShining ;
Handler tempOver30 ;
Handler tempUnder30 ;
Handler potentiometerHandler;
Handler serveStartStop;

Interrupt TiltInterrupt kind External on Tilt
  event kind RISING handled by tiltHandler;
Poll PhotoResistance if BETWEEN 550 & 1023 call sunIsShining;
Poll PhotoResistance if BETWEEN 0 & 549 call gettingDark;
Poll Thermistor if BETWEEN 164 & 1023 call tempOver30;
Poll Thermistor if BETWEEN 0 & 163 call tempUnder30;
Poll Potentiometer if CHANGE call potentiometerHandler;
Poll ButtonPressed if CHANGE call serveStartStop;
```

First a set of handler has been defined, one for each event to manage, interrupt included (*tiltHandler*). Then it is defined when each handler has to be called.

The *tiltHandler* is called when the Tilt sensor generates a rising edge, raising an external interrupt.
Other sensors are managed with polling. Photoresistance and Thermistor generate an event when they are between two thresholds, while the Potentiometer generates an event when it changes its value. The Button sensor is checked periodically and generates an event when its value changes.

Let's now define the tasks executed and their order.

```
Task tempSender;
Task lightSender;
Task tiltSender;
Task tempManager external;
Task lightManager external;
Task tiltManager external;
Task wait;
execution cycle:
  exec task tempSender
  exec task lightSender
  exec task tiltSender
  exec task wait;
```

The tasks *tempSender*, *lightSender* and *tiltSender* have been put in the model to send the messages about the current situation of the three sensors. The task *wait* is used to suspend the control flow for 500ms.

Other tasks are defined as *external* because they will be executed by Domotic2.

We can now pass to the definition of the communication parameters and of the exchanged messages:

```
define System{
  Communication Parameters:
  Serial baudrate=9600,
  Ethernet mac="90,A2,DA,00,4E,22" ip="192.168.0.3";

  Dispatch manageTemperature;
  Dispatch manageLight;
  Dispatch manageTilt;

  // OutOperations
  tempSender forward manageTemperature to tempManager;
  lightSender forward manageLight to lightManager;
  tiltSender forward manageTilt to tiltManager;

  // InOperations
  tempManager serve manageTemperature
    support= TCP [host="192.168.0.2" port=3000];
  lightManager serve manageLight
    support= TCP [host="192.168.0.2" port=3001];
  tiltManager serve manageTilt
    support= TCP [host="192.168.0.2" port=3002];

}
```

Domotic1 provides two communication supports: a Serial with baudrate 9600, used for debugging purpose, and an Ethernet with IP address *192.168.0.3* and its proper MAC.

We also find the definition of the messages exchanged. With no surprise we find the same messages introduced in the analysis.

The communication will be with tasks executed by Domotic2 and declared as external. Communication is divided into Out and In operations and follows the same logic introduced in the analysis.

The specification of Domotic1 is completed. When the file is saved all the code is generated and the Application Designer has only to implement the Handlers, in the header file *handlersImplementation.h*, and the Tasks, in the header file *tasksImplementation.h.*

The file *handlersImplementation.h* implementation is:

```
#ifndef handlers_h
#define handlers_h

#include "tasksImplementation.h"

void gettingDark::doJob(){
  lightMessage="dark";
}

void sunIsShining::doJob(){
  lightMessage="sunny";
}

void tempOver30::doJob(){
  tempMessage="over";
}

void tempUnder30::doJob(){
  tempMessage="under";
}

void tiltHandler::doJob(){
  tiltOn=true;
}

void potentiometerHandler::doJob(){
  int val=Potentiometer.getValue();
  val=map(val,0,1023,0,255);
  potentiometerLed.writeAnalogVal(val);
}

void serveStartStop::doJob(){
  started=ButtonPressed.getValue();
}
#endif
```

For each defined handler the Application Designer has to implement the *doJob* function. The file *tasksImplementation* is imported because it defines some state variables set by the handlers. For example the *gettingDark* handler sets the content of a string called *lightMessage* that will be the content of the message sent from *lightSender* to *lightManager*.

The potentiometer handler checks the value of the potentiometer and set it to the red LED. Of course the introduction of a wait task in the control flow will make the potentiometer LED less reactive to the changes of the potentiometer.

The file *tasksImplementation.h* instead becomes:

```
#ifndef taskimpl_h
#define taskimpl_h

boolean tiltOn=false;
String tempMessage;
String lightMessage;

boolean started=false;

void tempSender::doJob(){
  if(!started)return;
  forward_tempSender_manageTemperature_
    tempManager_TCP(tempMessage);
}

void lightSender::doJob(){
  if(!started)return;
  forward_lightSender_manageLight_
    lightManager_TCP(lightMessage);
}

void tiltSender::doJob(){
  if(!started)return;
  if(tiltOn)
    forward_tiltSender_manageTilt_tiltManager_TCP("true");
  else
    forward_tiltSender_manageTilt_tiltManager_TCP("false");
  tiltOn=false;
}

void wait::doJob(){
  delay(500);
}
#endif
```

Here the Application Designer has to implement the method *doJob* of each task.

The only thing done by the three sending tasks is sending the string set by the handlers if the *started* variable is set to true, meaning that the system has been started by the button. The *started* variable determines indeed if messages are sent from Domotic1 to Domotic2 or not.

## 7.5 Domotic2 Model

The definition of Domotic2 begins from the devices too:

```
Sketch Domotic2 Hardware: Arduino UNO
Actuator redLed pin "2";
Actuator greenLed pin "3";
Sensor Button pin "8" ;
Actuator ButtonPressed pin "6";
```

The name of the sketch is "Domotic2" and runs on Arduino UNO.
It is composed by two LEDs, one on pin 2 and the other on pin 3, and from a button, on pin 8.

The actuator defined as *ButtonPressed* actually is not a physical device. It is used as an output pin set when the button is pressed to send the signal to Domotic1.

Next thing to define are Handlers and Interrupts:

```
Handler buttonPressed;
Interrupt buttonPressed kind PinChange
  on Button event kind RISING
  handled by buttonPressed;
```

The only handler needed is for the interrupt generated when the button is pressed. The interrupt kind now is PinChange, because the button is connected to pin 8 and External interrupts are only supported on pin 2 and 3.

Let's now see the tasks definition:

```
Task tempSender external;
Task lightSender external;
Task tiltSender external;
Task tempManager;
Task lightManager;
Task tiltManager;
Task wait;
execution cycle:
  exec task tempManager
  exec task lightManager
  exec task tiltManager
  exec task wait;
```

Here we find the same tasks declared for Domotic1, but with different external qualifiers.

The definition of the communication supports and messages exchanged is:

```
define System{
  Communication Parameters:
    Ethernet mac="90,A2,DA,0D,1E,97" ip="192.168.0.2",
    Serial baudrate=9600;

  Dispatch manageTemperature;
  Dispatch manageLight;
  Dispatch manageTilt;

  tempSender forward manageTemperature to tempManager;
  lightSender forward manageLight to lightManager;
  tiltSender forward manageTilt to tiltManager;

  tempManager serve manageTemperature
    support= TCP [host="192.168.0.2" port=3000];
  lightManager serve manageLight
    support= TCP [host="192.168.0.2" port=3001];
  tiltManager serve manageTilt
    support= TCP [host="192.168.0.2" port=3002];
}
```

It defines two communication supports: a Serial with baudrate 9600, mainly used for debugging, and an Ethernet with IP *192.168.0.2* and the proper MAC.

We notice that the definition of the messages exchanged and of the Operations is the same found in Domotic1.

As for Domotic1, the Application Designer has to implement the files *handlersImplementation.h* and *tasksImplementation.h*.

The first is implemented as follows:

```
#ifndef handlersImplementation_h
#define handlersImplementation_h

boolean started=false;

void buttonPressed::doJob(){
  if(started){
    ButtonPressed.writeDigitalVal(0);
    started=false;
  }else{
    ButtonPressed.writeDigitalVal(1);
    started=true;
  }
}

#endif
```

A boolean variable called *started* is used to save the current state of the system. If the button is pressed and started is false it means that the system has to be started and a signal with value 1 is emitted from the actuator ButtonPressed. Otherwise, if started is true, it means that the system has to be stopped and value 0 is emitted.

The tasks are implemented as:

```
#include "handlersImplementation.h"

String mess;

void tempManager::doJob(){
  mess=serve_tempManager_manageTemperature_TCP();
  if(getMessageContent(mess)=="over")
    redLed.writeDigitalVal(1);
  else if(getMessageContent(mess)=="under")
    redLed.writeDigitalVal(0);
}

void lightManager::doJob(){
  mess=serve_lightManager_manageLight_TCP();
  if(getMessageContent(mess)=="sunny")
    greenLed.writeDigitalVal(0);
  else if(getMessageContent(mess)=="dark")
    greenLed.writeDigitalVal(1);
}

void tiltManager::doJob(){
  mess=serve_tiltManager_manageTilt_TCP();
  if(getMessageContent(mess)=="true"){
    int i;
    for(i=0;i<3;i++){
      greenLed.writeDigitalVal(0);
      redLed.writeDigitalVal(0);
      delay(100);
      greenLed.writeDigitalVal(1);
      redLed.writeDigitalVal(1);
      delay(100);
    }
  }

}

void wait::doJob(){
  delay(50);
}
```

Each task read a message from the TCP buffer and depending on its content, executes a particular action. All the actions are on LEDs. The wait task introduces a 50ms delay.

# Chapter 8

# Conclusions And Future Development

It is time to summarize the work done and the goals achieved, as well as the possible future development.

## 8.1 Conclusions

The aim of the thesis was the definition of a Meta-Model and the development of a Software Factory for Arduino platforms.

The process is gone through the definition of the Meta-Model, its formalization with a DSL and the implementation of all the code generators.

The meta-model have been directly mapped on a DSL using the framework Xtext. It has been designed with a bottom-up approach, starting from a deep study of the platform (features and constraints), so to identify the main concepts that characterize the domain.

After the definition of the DSL, the next step has been the implementation of the automatic code generators. All the generators have been fully implemented and tested, especially for the part related to the structure and behaviour of a single Arduino.
The distributed part has been added subsequently and it constitutes a good basis for future development.

To explain better all the work done and how to use it, the thesis contains some examples that can also be used as tutorials.

At the end of the work I can say that all the targets decided at the beginning of the thesis have been achieved, proving that a Model Driven approach can bring to a speed up of application development processes and to the production of good quality systems. In the examples provided it is possible to have an idea of the quality of the software produced and of how the work of the Application Designer becomes simpler and faster, compared to the classic Software Engineering approach.

## 8.2 Future Development

Even for future development we can divide it into the concentrated and distributed parts.

For the first part some features can be added:

- the introduction of special purpose sensors and actuators: the defined concepts of Sensors and Actuators are general and refers to devices connected only to a single pin. A device can also be connected to more than one pin (e.g LCDs) and this feature can be brought at the meta-model level.
- introduction of the concept of data exchanged or shared between the components of the system.

For the distributed part the work done can be extended with:

- support to synchronous message exchanging.
- introduction of proper policies related to the management of the buffers: e.g. what should the sender do if the receiver buffer is full?
- introduction of broadcast messages, like Events or Signals.
- introduction of other communication protocols.
- improvements of what has been done in this thesis.

# Bibliography

[1] OMG - Object Management Group: *Meta Object Facility (MOF) Core Specification*

[2] Sami Beydeda, Matthias Book, Volker Gruhn : *Model-Driven Software Development*, *Springer-Verlag Berlin Heidelberg, 2005*

[3] Antonio Natali, Ambra Molesini : *Costruire Sistemi Software: dai Modelli al Codice. Seconda Edizione. 2009*,

[4] Eclipse Foundation: *Ecore Tools*, http://wiki.eclipse.org/Ecore_Tools

[5] Denis Brighi: *Progettazione e sviluppo di un DSL ad agenti per la piattaforma Arduino, 2012*

[6] Karsten Thomas (Itemis): *Language Workbench Competition 2011 Xtext Submission, 2011*

[7] *XText 2.1 Documentation, October 31, 2011*

[8] Arduino Official Website: *http://www.arduino.cc/*

[9] ATMEL: *ATmega48A/PA/88A/PA/168A/PA/328/P Summary*

[10] Wiring Official Website: *http://wiring.org.co/*

[11] Antonio Natali: *Software system specifications in Contact, 2011*