Development of a Domotic Module for DomoBus

António Miguel Falcão Barbosa Bacelar de Sousa

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Renato Jorge Caleira Nunes
Prof. Paulo Rogério Barreiros de Almeida Pereira

Examination Committee

Chairperson: Prof. António Manuel Raminhos Cordeiro Grilo
Supervisors: Prof. Renato Jorge Caleira Nunes
Members of the Committee: Prof. Nuno Filipe Valentim Roma

June 2018
Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.
Acknowledgments

The completion of this thesis determines the end of my life as a full-time student. The achievement of this landmark wouldn’t be possible without the contribution of a set of people to which I owe some words of gratitude.

First of all, I want to transmit my sincere thanks to my supervisor, Prof. Renato Nunes. I feel grateful not only for his patience, availability and dedication that characterized the way I was guided throughout this time, but also for all the advices and shared knowledge, that were essential for the completion of this thesis.

I also would like to address to all my friends some words of appreciation, for their patience and companionship. I’m specially thankful to the ones that, directly or indirectly, kept on motivating me with their words and their example throughout this last year.

Finally, I want to dedicate these last words of gratitude to all the members of my family, for being a source of unconditional support and motivation to overcome the various challenges I faced along the years. I hope that someday I can repay in actions, more than in words, all the blessings I got from them.
Abstract

DomoBus is a home automation system being developed in an academic context. It aims to be a solution for the problems presented by current systems, such as the lack of support for interoperability between different technologies. DomoBus proposes a generic model for a domotic device, which is characterized by a set of properties, and a set of well-defined messages to interact with those properties.

A Control Module (CM), in DomoBus, is a hardware component that may control multiple domotic devices of different types. The main objective of this work is to design and implement a software architecture for CMs that will follow the DomoBus approach and simplify the future development of applications that will interface with different sensors and actuators. The designed software will be executed in a cooperative multitasking scheduling that includes time management and an implementation of the communication protocol. It will also be offered a model for the development of DomoBus applications in which they access the properties through a simple API, as if they were local variables.

In DomoBus, a Gateway Module (GM) is a hardware component that interfaces the CMs with a computer. The software that supports the desired functionalities will also be developed.

The software developed for the CMs and GMs is intended to be implemented in devices like Arduino boards, that present limited hardware resources. For the validation of the presented concepts, a prototype with two CMs and a GM connected to a computer was developed. The obtained results satisfy the established objectives.

Keywords

Home Automation, Domotic Systems, KNX, LonWorks, DomoBus, Domotic Module
Resumo

O sistema DomoBus é um sistema de automação de casas, desenvolvido num contexto académico, que visa solucionar os problemas apresentados pelos sistemas existentes, tais como a falta de interoperabilidade entre tecnologias diferentes. Este sistema propõe um modelo genérico para um dispositivo domótico, caracterizado por um conjunto de propriedades e um conjunto de mensagens para a interacção com essas mesmas propriedades.

Um Control Module (CM), no sistema DomoBus, é um componente hardware que pode controlar múltiplos dispositivos domóticos de tipos diferentes. O objectivo principal deste trabalho é o desenho e implementação de uma arquitectura software para os CMs que siga o método DomoBus e simplificar o futuro desenvolvimento de aplicações que façam a interface com os vários sensores e actuadores. O software desenvolvido irá executar num núcleo operativo multitarefa que inclui gestão do tempo e uma implementação do protocolo de comunicação. Será também oferecido um modelo para o desenvolvimento de aplicações DomoBus em que o acesso às propriedades é feito através de um API simples, como se fossem variáveis locais.

No sistema DomoBus, um Gateway Module (GM) é um componente hardware que faz a interface entre os CMs e um computador. O software que suporta as funcionalidades desejadas será também desenvolvido.

O software desenvolvido para os CMs e GMs será implementado em dispositivos como placas Arduino que apresentam recursos hardware limitados. De forma a validar os conceitos apresentados, foi desenvolvido um protótipo com dois CMs e um GM ligado a um computador. Os resultados obtidos satisfazem os objectivos estabelecidos.

Palavras Chave

Automação de Habitações, Sistemas Domóticos, KNX, LonWorks, DomoBus, Módulo Domótico
1 Introduction ................................................................. 1
  1.1 Motivation ...................................................................... 1
  1.2 Objectives ...................................................................... 2
  1.3 Document Outline .......................................................... 3

2 State of The Art ............................................................... 5
  2.1 Domotic Systems ............................................................ 5
  2.2 KNX ............................................................................. 6
    2.2.1 Introduction ................................................................ 6
    2.2.2 KNX Devices ............................................................. 6
    2.2.3 Programming a KNX System ........................................ 9
  2.3 LonWorks ....................................................................... 11
    2.3.1 Introduction ................................................................ 11
    2.3.2 Neuron Core and Smart Transceivers ............................... 12
    2.3.3 Control Network Protocol ........................................... 13
    2.3.4 Developing LonWorks Applications - Neuron C .................. 14
  2.4 DomoBus ....................................................................... 15
    2.4.1 System Architecture ................................................... 16
    2.4.2 Interaction with Devices .............................................. 18
  2.5 System Comparison .......................................................... 19

3 Proposed Solution ............................................................ 21
  3.1 DomoBus Control Networks ............................................. 21
  3.2 Requirements ................................................................. 22
  3.3 CM Software Architecture ............................................... 24
    3.3.1 Main Components ...................................................... 24
    3.3.2 Tasks and Applications .............................................. 25
    3.3.3 Operating Nucleus ....................................................... 25
  3.4 NET Layer ..................................................................... 27
  3.5 MSG Layer .................................................................... 29
  3.6 PROP Layer .................................................................... 30
  3.7 Applications ................................................................... 31
  3.8 DCN Gateway ................................................................. 32
4 Implementation

4.1 Hardware Selection

4.1.1 ATmega328P

4.1.2 nRF24L01+ Communication Module

4.2 Control Modules Components

4.2.1 NET Task

4.2.2 PROP Layer

4.2.3 MSG Task

4.2.4 Control Applications

4.3 Gateway Implementation

4.3.1 Control-Level Gateway

4.3.2 Supervision-Level Gateway Application

5 Results

5.1 Description of the Prototype

5.2 Configuration Process

5.3 Implementing the Applications

5.3.1 CM 1 - LAMP Application

5.3.2 CM 2 - LIGHT Application

5.4 Test 1 - GET, SET and NOTIFY

5.5 Test 2 - Message Retransmission

5.6 Test 3 - Error Messages

5.7 Test 4 - Array Manipulation

5.8 LED Control with a LDR

5.9 Implementation Problems

6 Conclusions

6.1 Work Performed and Results Achieved

6.2 Future Developments

Bibliography

Appendix A List of Communication Errors

Appendix B Node 1 Configuration File

Appendix C Node 2 Configuration File

Appendix D LAMP Application Code File
List of Figures

2.1 KNX logo .................................................. 6
2.2 KNX Decentralized System Architecture ........................................ 7
2.3 Bus Devices (BD) ................................................................ 8
2.4 KNX TP UART .................................................... 9
2.5 LonWorks Control Network Topology ........................................... 12
2.6 FT 6000 Smart Transceiver .................................................... 12
2.7 FT 6000 Smart Transceiver Chip Architecture ............................... 13
2.8 Sample LonWorks network ..................................................... 14
2.9 Functional Profile for a Constant Light Controller ....................... 15
2.10 DomoBus Logo ..................................................... 16
2.11 DomoBus System Architecture ............................................. 16
2.12 DomoBus GET Message ................................................ 18
2.13 DomoBus SET Message ................................................. 18
2.14 DomoBus NOTIFY Message .......................................... 19
3.1 DomoBus Control Network ............................................. 22
3.2 Control Modules - Available Hardware Devices .............................. 23
3.3 CM Software Architecture ............................................. 24
3.4 EXEC Phase .................................................................. 25
3.5 NET-DCN interface ..................................................... 27
3.6 DomoBus Control Level Packet ...................................... 27
3.7 D_ARRAY Property .................................................. 30
3.8 Gateway Modules Software Architecture .................................. 32
4.1 nRF24L01+ radio Module ............................................ 38
4.2 Basic Structure of the Data Arrays .................................. 42
4.3 Property Data Arrays Example ............................................. 43
4.4 DomoBus Control Level data frame .................................... 43
4.5 DomoBus Control Level Data Frame Address Fields .................. 44
4.6 DomoBus Control Level Data Frame CTR Field ......................... 44
4.7 DomoBus Control Level Data Frame Data Field Property Descriptor 45
4.8 DomoBus Control Level Packet Data Fields ................................ 46
4.9 Event Structure ........................................................ 47
4.10 List of Activated Events - Functioning Example ...................... 49
Abbreviations

ADC  Analog-To-Digital Converter
AP   Application Parameters
API  Application Programming Interface
BCI  Batibus Club International
BCU  Bus Coupling Unit
BD   Bus Devices
BLE  Bluetooth Low Energy
CL   Control Level
CM   Control Modules
CNP  Control Network Protocol
CP   Configuration Properties
CRC  Cyclic Redundancy Check
DCN  DomoBus Control Networks
DDR  Data Direction Register
EEPROM  Electrically-Erasable Programmable Read-Only Memory
EHSA  European Home Systems Association
EIB  European Installation Bus
EIBA  European Installation Bus Association
ETS  Engineering Tool Software
FB   Functional Blocks
FP   Functional Profiles
GM   Gateway Module
HVAC Heat, Ventilation and Air Conditioning
ICSP In-Circuit Serial Programmer
IDE  Integrated Development Environment
ISM  Industrial, Scientific and Medical
ISO OSI  International Standard Organization Open Systems Interconnect
KNX TP  KNX Twisted Pair
KNX PL  KNX Powerline
KNX RF  KNX Radio Frequency
LAE  Lists of Activated Events
MCU  Microcontroller Unit
NV  Network Variables
PEI  Physical External Interface
RAM  Random Access Memory
ROM  Read-Only Memory
SL  Supervision Level
SM  Supervision Modules
SPI  Serial Peripheral Interface
SRAM  Static Random Access Memory
TM  Transmission Module
UART  Universal Asynchronous Receiver/Transmitter
UI  User Interface
USART  Universal Synchronous Asynchronous Receiver/Transmitter
USB  Universal Serial Bus
Chapter 1

Introduction

The human being lives in an era where technology plays, more than ever, an essential part of the daily life. More and more often one gets news about new developments in a wide range of industries and their products that aim to facilitate and improve the quality of life of its users. This evolution is noticeable in a great variety of devices, like smartphones, that turned cellphones from tools for making phone calls or send text messages by SMS into small computers with countless capabilities, or cars, in which electric motors will gradually be their mainstream form of motion power, not to forget the developments being made on automated vehicles, that will make driverless cars a reality in a nearby future.

Surprisingly, one can’t verify this same evolution on houses. It is undeniable that applying technology in our homes would ease everyday life. By using home automation systems, people would be able, for example, to turn on/off the different devices of the house, like lamps, heaters, air conditioning systems or other household applications. This could be done in a remote way by using an application installed on the computer or in the smartphone, increasing the comfort and providing energy savings. This kind of system would provide a significant increase in the quality of life of the population, being especially useful for elderly or handicapped people. Unfortunately, even though there are already some home automation systems in the market, there still isn’t a mainstream system applied to the majority of our homes.

The creation, development and application of home automation systems are processes related to the Domotics field of study. The first kind of this type of systems appeared in France in the 1970’s [1]. These were centralized systems, in which a central control unit was responsible for the management of the whole system. By using various units and applying a hierarchy around them, the first distributed systems appeared. These systems evolved into integrated systems in the beginning of the 1980’s.

1.1 Motivation

Nowadays, one can easily verify that there are plenty of home automation systems available in the market, like KNX and LonWorks, that already provide a substantial amount of functionalities. However, all these systems present the problem of being incompatible with one another, and the manufacturers are not making efforts in order to present solutions to this issue [2]. In addition, it is not easy to install and configure these technologies, along with the fact of being hard to adapt to the user needs and preferences. Besides, technologies like KNX and LonWorks use specific circuits and devices, which are very difficult to adapt to other protocols. Finally, in some of these systems, the software
that is responsible for their management is proprietary. Therefore, it is difficult to get an idea of the software architecture used, with the objective of facilitating the development of similar applications or the adaptation of the existing ones.

In this context, it is easy to understand why one should invest in a system that could present a solution to the enumerated problems and provide the same or better features than the ones offered by existing systems. That is precisely the main goal of DomoBus, an open-source home automation system created in an academic environment for teaching purposes.

In DomoBus, devices are generic entities characterized by a set of properties. A lamp, for example, may be characterized by two properties, the On/Off status and the level of intensity of the light. The interaction with the devices is made through a simple set of messages: GET (to get the value of a device's property), SET (to modify a property's value and, consequently, change the state of the device) and NOTIFY (transmitted automatically by the devices when one of its properties has changed its value).

The DomoBus architecture is divided into two interconnected levels. The first one, called Supervision Level (SL), is responsible for the global management of the system, the high-level interface with the users and control of the system behaviour. The other one, designated Control Level (CL), is constituted by the modules that interface with the sensors and actuators. These modules are interconnected through a network, called DomoBus Control Network (DCN).

1.2 Objectives

This work will focus on the Control Level of the DomoBus system, more specifically on the development of a Control Module (CM) for a DCN. A CM is a hardware component similar to an Arduino board, characterized by its low-cost and open development environment, which is responsible to interface with sensors and actuators. Each CM is able to control multiple domotic devices of different types.

The objective of this work is the development of a modular software architecture for a CM that will support all the communication requirements and the management of devices' properties. The main goal is to offer a simple and generic API for device developers that will ease the integration in DomoBus of new devices. Developers just need to interface with the devices' hardware and map their relevant characteristics into properties, which will be managed through the given API.

The software architecture of a CM will be composed of three main layers: NET, MESSAGE and PROPERTIES. All these layers, in conjunction with the control applications, will be executed in a cooperative multitasking scheduling.

The NET layer is responsible for the transmission and reception of packets, allowing each CM to interact with other CMs and with the applications of the Supervision Level. This layer implements an interface between the CM and the communication module used in the exchange of data. The developed software will support the use of various communication means. One just needs to change the code of the NET layer according to the used communication module.
The MESSAGE layer processes the DomoBus messages, received through the NET layer. These are standard DomoBus messages of one of the following types: GET, SET and NOTIFY. These messages operate over the properties of the domotic devices, which are managed by the PROPERTIES layer.

The PROPERTIES layer will offer a simple API to the developer of a domotic device that allows an easy access and manipulation of its properties. Developers just need to interface with the devices’ hardware and map their relevant characteristics into properties. The given API will allow the automatic update of a property’s value and, through the MESSAGE layer, transmit the corresponding NOTIFY message to the Supervision Level. On the other hand, when the MESSAGE layer receives a SET message and modifies a property’s value, the PROPERTIES layer will trigger automatically an event that signals that the corresponding device’s state must be changed. In case the Control Module receives a GET message, the MESSAGE layer will retrieve the requested property value, without needing any kind of intervention from the control applications.

Given the involved complexities, it will also be developed an application that handles the configuration of a Control Module, according to a set of applications, devices and properties defined by the user. This application will also generate a basic structure for the code of the CM applications, exposing the developed API and simplifying the access to the property values.

With the objective of facilitating the integration of the developed Control Modules in the DomoBus system, a modular software architecture for a Gateway Module (GM) will also be developed. A GM is a hardware component similar to a CM that allows the connection of the Control Modules to the system’s Supervision Level. The software architecture of the gateway will be constituted by two layers: NET and SM_COMM.

The NET layer, as it happens on the Control Modules, is responsible for the exchange of data between the Gateway Module and the Control Modules. The SM_COMM layer, in turn, deals with all the operations related to transferring data between the GM and the SL.

1.3 Document Outline

Chapter 2 analyses how domotic modules are developed in other technologies, namely in KNX and LonWorks, both regarding their hardware and software architectures. Chapter 2 also includes an introduction of the DomoBus system, along with a description of its main functionalities and operation principles. In the end will be presented a comparative study of the three mentioned systems.

Next, in Chapter 3, it is performed an initial study regarding the hardware requirements of the boards that will be used as Control Modules (CM) and Gateway Modules (GM), followed by a description of the proposed software architectures. The proposal includes different software layers, which will be detailed, along with other main functionalities.

In Chapter 4, will be performed a detailed description of the various functionalities implemented in each one of the software layers of the CM’s and GM’s software architectures. This chapter also includes a description of the features of the boards and the communication modules chosen for the
implementation of the developed software architectures.

In the following, Chapter 5 will show the results of various simulations and tests performed on a prototype that aim to demonstrate and validate the implemented functionalities.

Finally, in Chapter 6 some final remarks will be provided and also will be made some suggestions regarding further developments of the work carried out.
Chapter 2

State of The Art

In this chapter will be analysed the development process and the operation principles of the domotic modules used in the most important home automation systems available on today’s market, namely KNX and LonWorks.

This analysis will start with a brief description of domotic systems in Section 2.1 followed by a study of the domotic modules used in KNX and LonWorks in Sections 2.2 and 2.3 respectively. Next, in Section 2.4 is performed a study of the DomoBus system in which are presented its main operation principles. Finally, in Section 2.5 is carried out a comparison between KNX and LonWorks regarding the hardware and software features of the systems’ domotic modules, and is also shown how the modules to develop for the DomoBus system present solutions to the problems found in modules from other technologies.

2.1 Domotic Systems

Home automation systems are based on networks of sensors and actuators. Sensors gather all kinds of data concerning the state of a house such as luminosity or temperature. With the gathered data, the system performs actions in the installed set of actuators according to what it is defined in its control software. Based on the value of the air temperature measured inside a certain room, for example, a Heat, Ventilation and Air Conditioning (HVAC) system will heat up or cool down the air to match a defined temperature value.

In the assembly process of these kind of networks, there are two main approaches to take into consideration [3] [4]:

- conventional/centralized control systems
- control networks or bus technologies

The first approach consists of the employment of a star topology, in which all the controlled devices are connected to a central distribution board that controls the system. In this method, a small increase in the number of components to control extends the wiring and the size of the distribution boards in an unreasonable way, turning the installation phase into a tedious process. In this kind of systems the controlled devices act as “islands of automation”, with limited communication between them [4]. The installation process of these systems is an expensive process and their posterior expansion is quite difficult.
Concerning control networks, all the controlled devices are connected via the same communication protocol. Given their decentralized structure, a sensor may communicate directly with an actuator without having to exchange information with a central control unit. In this kind of networks, the various devices can go from simple sensors and actuators to complex supervisory control and data acquisition systems, that monitor and control other devices in the network.

These control networks are particularly advantageous if one desires a system that controls a substantial number of devices. In comparison with centralized systems, the installation process of decentralized systems is cheaper and more flexible [3].

Given their simplicity and flexibility, decentralized systems represent the present and the future of home automation systems. As such, it is important to analyse some of the main systems of this kind, like KNX and LonWorks, in order to gain the necessary knowledge on the subject.

### 2.2 KNX

#### 2.2.1 Introduction

KNX is the most popular home automation system in Europe.

Originally, this system was called European Installation Bus (EIB) and it was owned by the European Installation Bus Association (EIBA) [3]. In 1999, EIBA merged with Batibus Club International (BCI) and with the European Home Systems Association (EHSA), resulting in the establishment of the KNX Association, whose logo is depicted in Figure 2.1.

![KNX logo](source.png)

**Figure 2.1:** KNX logo. (Source: [5])

All KNX devices are compatible with the EIB system and vice-versa [3].

In 2011, a study conducted by the agency BSRIA showed that at that time KNX-based solutions exceeded 70% of the total market value [6]. This study also states that the market share of KNX is steadily growing, as more and more often this system is the one that is installed in buildings across the continent.

#### 2.2.2 KNX Devices

KNX is a system based on a decentralized architecture, as depicted below in Figure 2.2.
All the control modules, equipped with a microprocessor, share the task of controlling the overall system [3]. This allows the well-functioning of the system in case one or more modules stop functioning, which is possibly the best advantage of this technology. However, in accordance with the same source, it is possible to have centralized units for very specific applications. Naturally, this means that if one of these central control units stops functioning for some reason, the connected control modules may also fail. In the future, it is desirable that this kind of approach becomes less and less necessary, in order to create the least dependencies possible between devices.

With its decentralized system architecture, it is possible in KNX to connect a big number of devices at the same time. Theoretically, more than 50000 (fifty thousand) different devices can be connected to the same system [3]. This feature demonstrates the great control capacity of the KNX technology, that should become a standard feature for emerging home automation systems.

In this system the different devices can be separated into two separate categories [3]:

• System Devices, such as power supplies or the programming interfaces;

• End Devices or Bus Devices (BD), associated with the different devices controlled by the system.

It is important to notice that each one of these modules was developed for the control of a specific type of device. This means one cannot have a lamp and a light sensor connected to one same BD, for example.

Taking into account the main objectives of this work, it is important to perform an analysis of the structure of the Bus Devices.

**KNX Bus Devices**

As represented in Figure 2.3, the KNX Bus Devices are constituted by two components, a Bus Coupling Unit (BCU) and the application module.
These components may be separated [3]. If so, a 10 or 12 pin Physical External Interface (PEI) is needed to connect them.

Focusing on the BCU, this is a device with many possible designs that, fundamentally, is constituted by two sub-components: a Transmission Module (TM) and a Controller [3].

The TM is responsible for implementing the communication protocol by which the BD exchanges information. In KNX there are four different communication media with their own protocol:

- KNX Twisted Pair (KNX TP), in which the communication between devices is ensured by a twisted pair data cable;
- KNX Powerline (KNX PL), that uses the 230 V power supply network for the transmission of information;
- KNX Radio Frequency (KNX RF), where radio signals are deployed for the exchange of data;
- KNX IP, in which inter-device communication is ensured by Ethernet cables.

The variety of communication media is a very interesting and advantageous feature, that increases the overall flexibility of the system. However, it is also important to point out that the different communication protocols presented can’t co-exist in the same system, as that would bring problems in the transmission of data [3]. If needed, the system may be divided into sub-systems, each one for the different protocols that are used.

The TM incorporated in the BCU depends on the chosen communication medium. Generically, all Transmission Modules are responsible for the interface between the Controller and the KNX data bus. The TMs used in KNX TP and KNX PL include a power supply for the Controller and also send reset pulses to this sub-component of the BCU [3].

If one uses KNX TP as the system communication protocol, for example, the associated TM is called TP UART. The structure of this TM is depicted in Figure 2.4.
As it can be seen, this particular Transmission Module is constituted by two interconnected parts: the Analog Part and the Digital Part.

The Analog Part is directly responsible for the exchange of data between the BCU and the KNX data bus, and also contains the device’s power supply, and the Digital Part implements the UART interface between the TM and the Controller.

All the hardware specifications of the TP-UART are provided by the manufacturer, along with the details regarding its UART interface.

Concerning the Controller, this sub-component is basically an NEC, ATMega or Texas Instruments generic 8-bit microcontroller [3]. These are characterized by three types of memory:

- **RAM** where temporary parameters are stored;
- **EEPROM** that stores the parameters, addresses, and other data downloaded from the PC;
- **ROM** in which the BCU’s system software is stored.

The BCU has a variety of functions. It is responsible for controlling the access of the device to the communication bus, and also manages the actual end device (that sometimes comes with its own microcontroller, that replaces the BCU in this task).

### 2.2.3 Programming a KNX System

There are two possible configuration modes for the programming of a KNX system [3]:

- Easy Mode (E-Mode), in which no software is needed;
• System Mode (S-Mode), in which a computer application designated Engineering Tool Software (ETS) is used to configure the system. This is the most common configuration process.

Taking into consideration that one of the main objectives of this thesis is the development of a software architecture for the control modules of a home automation system, it's important to analyse the ETS as this is the software responsible for the configuration and control of the devices connected to the KNX system.

In order to understand the configuration process of a KNX system on S-Mode, it is important to notice that various manufacturers of KNX products provide a database that includes their specifications and software. This data is essential for the integration of the devices in the KNX system [3]. Naturally, the first stage of the configuration process consists precisely in the download of the products’ data, that is then imported to ETS.

Having the needed data, the actual system configuration may proceed. The first phase is the design of the system, in which the layout of the structure of the building is specified and the location of the devices is defined along with their functions in the system. This process generally goes through the following steps:

1. creation of a project with the downloaded data, which is related to each type of device that will be used;
2. specification of the building structure layout, the devices to be used and their location inside that same building;
3. establishment of the parameters of the KNX products. In this phase, for example, it’s possible to define the functions of the various KNX device buttons;
4. definition of the system functions and of the Group Addresses. As an example, if a room has two windows, each one with blinds that are independent of one another, one can define three Group Addresses:
   • Blind 1 Up/Down;
   • Blind 2 Up/Down;
   • Blinds 1 and 2 Up/Down (allows commanding both blinds at the same time);
5. linkage of the Group Objects to the Group Addresses by establishing virtual connections between the inputs and outputs of the devices. In this phase occurs the definition of the relations between sensors and actuators;
6. next, one can optionally define the trades to which the KNX devices in the installation relate;
7. review, saving, and backing up of the project. The project can be edited and modified later on.

The design phase is followed by the commissioning phase, in which Individual Addresses are assigned to the different devices. This is performed by pressing a special button on the devices called Programming Button. With this action, ETS will assign the next address it has to be issued to the
device in which the button was pressed. There is a need for a special care in this phase, as errors may occur and the time spent in their correction is quite substantial. After this, the system software can be downloaded to the devices.

After the installation, ETS also makes available a set of diagnostic functions that help on the maintenance of the system. With them, it is possible to gather information regarding the state of operation, along with various other functionalities. It is also possible to check the activity of the communication bus and access it via a computer (normally using a KNX USB port or a KNXnet/IP router), which is a very powerful functionality, particularly if one wants to test a certain device.

Concerning other aspects, it is important to note that there are also some special plug-ins to consider as some devices require some additional software. Besides, applications for smartphones and tablets, called ETS Apps, are also available and be purchased online. The amount of available software extensions and applications shows that this system is in constantly evolving, in accordance with the technologies of the present times.

The major flaw of ETS is the fact that this is a proprietary software, which means that one can only get little or no information regarding the development of the system's control modules, like programming methodologies or software libraries.

2.3 LonWorks

2.3.1 Introduction

The LonWorks platform, developed by Echelon Corporation, is a very complex and sophisticated system that is used in various industries and normally installed in office buildings, malls or hospitals. Echelon started the development of this platform in 1988 along with its own communication protocol. Since then, this protocol was standardized and today it is an international standard, the ISO/IEC 14908-1 Control Network Protocol.

The main goal of LonWorks is to turn the creation of building control systems into an easy and cost-effective process, by providing a well-integrated, optimally designed and economical platform for creating smart devices and networks. This way, manufacturers can put more focus on the development of control features and applications for their products, instead of spending time and effort on the implementation and testing of their own protocols and control devices.

The typical LonWorks Control Network topology is represented in Figure 2.5. It is possible to divide this generic network into three different levels of control:

- Supervisory Level, related to the modules responsible for the management of the system;
- Field Level, that comprehends the modules that directly control the various devices;
- I/O Level, related to the sensors and actuators connected to the system.
2.3.2 Neuron Core and Smart Transceivers

In a LonWorks Control Network, the different control devices communicate using a common protocol, the ISO/IEC 14908-1 Control Network Protocol (CNP), that will be briefly described in Section 2.3.3. Each device implements the communication protocol and performs control actions. These control devices may include communication transceivers to allow the coupling of the device with the communication medium.

In order to achieve economical and standardized deployment Echelon designed the Neuron Core, that includes multiple processors, memory and interfaces for communication and I/O [4]. Echelon also developed a complete operative system for the Neuron Core called Neuron Firmware, that includes a complete implementation of the CNP.

The Neuron Core is available as a standalone component called the Neuron Chip. There are a few different types Neuron Chips available in the market, with their own interfaces, speeds and memories.

In order to reduce costs, these cores may be combined with communication transceivers. The resultant devices are called Smart Transceivers.

The FT 6000 Smart Transceiver

As an example, let’s consider the FT 6000 Smart Transceiver [9], depicted in Figure 2.6.
These devices include the following features:

- 4 CPUs, where each one has a specific task assigned. They are responsible for the chip operation, network communication, user application processing and interrupt handling;

- 16 KB ROM, where it is stored a system firmware image responsible for booting a system image from the flash memory;

- 64 KB RAM used to store the user application and data;

- 12 bi-directional I/O pins, with 35 programmable I/O modes that support both 5V and 3.3V operation;

- on-chip clock, with a maximum oscillation frequency of 80 MHz.

- a communication port that provides network access for the chip

The chip architecture for this family of transceivers is represented in Figure 2.7.

![Figure 2.7: FT 6000 Smart Transceiver chip architecture (Source: [9])](image)

It is important to notice that, in opposition to what happens in the KNX system, all the information regarding the hardware architecture of the modules used in LonWorks platform is available, which is a quite advantageous feature for developers of LonWorks applications.

### 2.3.3 Control Network Protocol

The Control Network Protocol (CNP) is the name given to the communications standard for control applications in the LonWorks Platform [4]. Its major features include the efficient and reliable delivery of messages, its compatibility with multiple communications media, low installation and maintenance costs and an authentication protocol that protects the network from commands given by unauthorized users.
Echelon’s implementation of the CNP is called the LonTalk protocol [4]. Taking into account a Neuron Core, as an example, running this protocol requires less than 10 KB of code and less than 1 KB of RAM.

The CNP is a complex 7-layer protocol that follows the International Standard Organization Open Systems Interconnect (ISO OSI) reference model. This ensures that the different communication services occur without unexpected interactions between them. This protocol was developed not only to provide a robust communication solution for the different applications but also to meet the needs of evolving applications in the future. A complete description of this protocol and its services for each one of the seven OSI layers is available from the Echelon Corporation [4].

In CNP, there are many types of communications transceivers available that support both the transmission and reception of data between devices. These support communication over twisted pair, link power, power line, radio-frequency, fiber optic, and infrared media [4].

2.3.4 Developing LonWorks Applications - Neuron C

The LonWorks applications are programmed in a special language called Neuron C [10]. This language is based on the classic ANSI C and it was specially conceived for the LonWorks Neuron Chip and Smart Transceivers. This language includes network communication, I/O and event-handling extensions for the ANSI C.

In this language, the most important kind of variables is called Network Variables (NV). These variables are used by the applications to identify the devices to control. For example, consider the sample network depicted on Figure 2.8.

![Sample LonWorks network](Source: [10])

In this example, the NV `nvoSwitch` and `nviLamp` represent the devices controlled by the respective applications. These variables are connected by a network tool like the LonMaker Integration Tool [10].
Another important kind of variables is the Configuration Properties (CP). These are variables that help in the configuration of the respective device, in which one establishes its characteristics.

To facilitate the application development, the LonWorks platform defined sets of NV and CP called Functional Profiles (FP). The FP are used to describe devices with similar functional behaviour. In Figure 2.9 is depicted a FP for a light controller.

![Figure 2.9: Functional Profile for a Constant Light Controller (Source: [10])](image)

In Figure 2.9 one may easily verify that the Functional Profiles include mandatory and optional CP and NV. It’s important to point out that some FP may not contain a certain type of variables [10]. Programmers may also define their own Functional Profiles when developing their applications.

LonWorks application programmers may define instances of FP called Functional Blocks (FB). When declaring these instances, a programmer may assign additional NV and CP to the ones that were already defined in the FP. In a LonWorks application, one may define a maximum of 254 Functional Blocks, depending on the hardware features [10].

In opposition to what happens with KNX, Echelon makes available a complete reference guide for programmers that want to develop applications in the LonWorks platform [10]. Unfortunately, the software architecture behind the Neuron Firmware is still unknown, which means that it’s almost unfeasible for someone to get a general idea of it in order to develop compatible modules. This way, the development of LonWorks applications is almost impossible to perform for devices other than Neuron Chips.

### 2.4 DomoBus

DomoBus, whose logo is depicted in figure 2.10, is a system created in an academic environment for teaching purposes that includes features like inter-protocol compatibility and ease-of-use by common users and programmers.
This is the system for which the work reported in this thesis was developed. As such, it is important to give an insight into its main features and characteristics, namely its system architecture, devices and communication protocol.

### 2.4.1 System Architecture

Figure 2.11 illustrates the architecture of the DomoBus system.

By analysing the picture above, it is possible to verify that the system may be divided into two different levels:

- the **Supervision Level (SL)**, constituted by the Supervision Modules (SM) is responsible for the management of the system. In this layer, it is possible to implement some sophisticated automatisms concerning the devices controlled by the system;

- the **Control Level (CL)**, constituted by Control-Level Networks, is responsible for the interface between the Supervision Level and the various control points (sensors and actuators). These Control-Level networks may belong to different technologies, like KNX or LonWorks, or may be
related to DomoBus itself. In the latter case, such networks are designated DomoBus Control Networks (DCN). A DCN is constituted by various Control Modules (CM), responsible for the control of the different home devices, and by a Gateway Module (GM) that provides a connection between the Supervision Modules and the Control Modules.

The Supervision Modules and the Control Modules are the most important components of the entire DomoBus system. Therefore, it is important to analyse these devices in order to gain a better understanding of the working principles of this system.

**DomoBus Supervision Modules**

The Supervision Modules are the devices responsible for the management of the system. These are also the modules through which the users may interact with the system. By interacting with the Control Modules, the SMs can get information regarding the state of the devices to control and perform actions on them according to the programmed automatisms. Many devices may be used as Supervision Modules, such as Raspberry Pi boards, common PC, tablets or smartphones. The SMs may run multiple applications, from which the most significant ones are the following:

- **Supervisor** - application where system supervision actions are programmed. This application can interact directly with User Interface applications and the various Gateway applications;

- **User Interface (UI)** - application by which a user can interact with the system. By using this application, the user can monitor and command any device of the whole system;

- **Gateway** - this type of application is responsible for the communication between the SMs and a specific Control-Level network. A SM may have one or more Gateway applications, depending on the number of CL networks it interfaces with.

It is possible to have as many SMs as one may want [11]. In small systems probably just one SM may be enough but, in bigger systems, it may become necessary to use various SM.

**DomoBus Control Modules**

The Control Modules are hardware devices that interact directly with the home devices that one wishes to control with the DomoBus system.

Hardware-wise, the CM are based on simple and generic low-cost boards similar to Arduino, characterized by their open development environment. The CM microprocessors must include memories for code and data, a communication transceiver and an interface for input and output devices, among other peripherals. The objective is for a CM to be able run different applications and control multiple devices simultaneously. This is a specially important feature in homes where the number of control points is quite substantial. By taking this approach, it's possible to reduce costs and keep a simple system architecture.

Another important aspect to consider is the fact that the DomoBus Control Modules may be installed together in a technical panel, making use of ordinary switches and power sockets [11]. This is
a cheaper and more practical approach than the one applied in some traditional systems, where each device incorporates their own microcontroller, peripherals, power supply and switches.

2.4.2 Interaction with Devices

In DomoBus the devices to control are considered to be generic entities characterized by a set of controllable properties. If one wishes to control a common lamp, for example, two properties may be taken into consideration: its On/Off status and its light intensity.

DomoBus will interact with the different home devices by controlling its properties. This interaction is based on a simple communication protocol, with three main types messages: GET, SET and NOTIFY.

The GET message is sent in order to get a property value of a certain device. This is the message to send if, for example, a Supervision Module wants to know the On/Off status of a lamp, as illustrated in Figure 2.12. The respective reply message sent by the device, with the requested value, is called A_GET(Answer to GET).

![Figure 2.12: DomoBus GET message](image)

By using the SET message, it is possible to change a property value of a certain device. For example, if one wishes to turn on a lamp, as depicted in Figure 2.13, a SET message should be sent with the desired value for the lamp On/Off status property.

![Figure 2.13: DomoBus SET message](image)

The NOTIFY message is sent by the Control Modules to the Supervision Modules when the properties of their associated devices change their value. In order to understand the dynamics of this message, let’s take into consideration the example represented in Figure 2.14 in which a CM controls a lamp that, at a given moment, is switched off. When the CM receives a SET message from an SM that changes the lamp status’ property value (off to on), for example, the actual change in the lamp’s status is reported with a NOTIFY message, that includes the new value of the referred property.
This message is particularly useful in the sensors’ domain. If a temperature sensor, for example, reads a new value for the temperature of the room in which it is placed, a NOTIFY message is sent in order to report the situation.

Generally, all these messages are exchanged between the Supervision Modules and the Control Modules. But sometimes the exchange of messages between Control Modules may be useful. If one has a light sensor that affects the properties of a single lamp, for example, a Supervision Module may not be needed, as the respective Control Modules can be programmed in order to allow the light sensor to directly command the lamp with SET messages in accordance with its measurements.

2.5 System Comparison

This chapter introduced some of the home automation solutions considered to be more relevant, namely KNX and LonWorks, and of DomoBus, which is the system for which the work reported in this thesis was performed. Now that one has a general notion of the working principles of each one of these systems, it is possible to make an evaluation of their features and characteristics, with a special focus on the control modules and their software/hardware architectures. With this assessment, it is also possible to establish a policy to be followed in the DomoBus Control Module’s development.

Hardware

Concerning the control modules’ hardware, it’s possible to verify that each one of the current systems follows its unique approach, considering their accessibility and functionality:

• in KNX and LonWorks the Bus Devices and Neuron Chips/Smart Transceivers were specifically designed for their respective systems;

• in LonWorks the modules’ hardware architecture is completely accessible. In KNX, on the contrary, one can only access the hardware details of some module components, such as the TP-UART, along with specifications regarding the various communication protocols;

• the Neuron Chips/Smart Transceivers are capable of controlling devices from different types at the same time, while in KNX the various modules are related to a specific type of devices. While in LonWorks one may easily have a light sensor and a lamp controlled by the same Smart
Transceiver, this is difficult to perform with KNX control modules, although they are also capable of controlling various devices (of the same type) simultaneously.

Taking these aspects into account, one can easily verify that hardware-wise the LonWorks system follows a more open and flexible policy than KNX. However, in both these systems, one has to use control modules that were specifically designed for them, which are hard to obtain.

In DomoBus, the Control Modules are intended to be based on accessible low-cost generic boards, which is probably the greatest difference between the CM and the modules from the mentioned technologies. As it happens in LonWorks, all the hardware specifications of the used boards must be available. The CM must also be able to control multiple devices of different types simultaneously.

Software

Regarding the software architecture and programming methods of the control modules, it is also clear that each system has its own policy, particularly in respect to its accessibility. In KNX, although the programming process of the system with ETS is available, one cannot get information regarding the actual methods deployed by ETS in this configuration process and the software architecture of the various control applications. In respect to LonWorks, Echelon corporation developed a special language called Neuron C, based on the widely-known classic C language, that includes some specific extensions, for which a complete programming guide is available. However, the software architecture of Neuron Firmware, the devices’ operative system, is unknown.

One more time, it is possible to notice that LonWorks follows an approach characterized by being more open and accessible than the one followed in KNX. However, not all the source code of the software implemented in LonWorks modules is available. Consequently, regarding both technologies, it is difficult to obtain the information needed for the development of similar modules, for example.

To face the problems detected in other technologies, the software implemented in the DomoBus Control Modules will be completely accessible, without any kind of restrictions. This way, people interested in understanding the software details of the CM can get the information they need. Regarding the programming language, and as it happens in LonWorks, it is intended that the development of CM applications to be based on a widely-known language like C. However, one wants the software development to be as generic as possible, so the use of additional libraries and extensions will be avoided at a maximum extent.
Chapter 3

Proposed Solution

In this chapter is performed a description of the hardware and software features of the Control Modules and Gateway Modules developed for a DomoBus Control Network (DCN).

A DCN is described in Section 3.1 in which the objectives of the work developed regarding the components of a DCN are also presented. Next, in Section 3.2 is defined a set of minimal hardware features for the microprocessors to use in Control Modules and Gateway Modules of a DCN. In Section 3.3 is presented the software architecture proposed for a CM. The operation principles of the software’s main layers, NET, MESSAGE and PROPERTIES, and of the control applications, are described in Sections 3.4 3.5 3.6 and 3.7 respectively. Finally, in Section 3.8 is described the software architecture proposed for a Gateway Module and the functioning principles of its layers, NET and SM_COMM.

3.1 DomoBus Control Networks

The DomoBus system may be divided into two levels, namely the Supervision Level (responsible for the management of the system) and the Control Level (where the various devices are controlled). In the Control Level, the various devices are controlled by Control Modules (CM), that may be connected with each other in a DomoBus Control Network (DCN), as represented in Figure 3.1. This network also contains a Gateway Module (GM) that is responsible for the communication between the CMs and the modules of the Supervision Level.

Considering the structure of a DCN network (Figure 3.1) and its purpose, the present work aims to fulfil the following objectives:

- definition and development of a modular and generic open-source software architecture for the DomoBus Control Modules that, meeting the system’s inherent characteristics and features, deals with all the communication details and manages the properties of the controlled devices (sensors/actuators);

- creation of an API that facilitates the development of the applications responsible for the control the devices connected to a CM;

- delineation and implementation of a modular and generic software architecture for the Gateway Modules (GM), that interconnects the two levels of the DomoBus system architecture;

- establishment of the minimal hardware requirements needed for the implementation of the developed software.
3.2 Requirements

Before proceeding with the software development process, it is important to establish the hardware requirements of the modules where this software is going to be implemented. Taking into account the functionalities one wishes to implement in both the GMs and CMs of a DCN, one can already establish a set of minimal hardware requirements for the microprocessors that will support the developed software.

Regarding flash memory, 16 KB memories are expected to be enough for the developed software. Concerning RAM memory, some initial estimations were performed given the number and type of the data structures that will support the desired functionalities. 512 bytes probably would be enough, but to give a safety margin it was decided that the microprocessor should have 1 KB or more of RAM memory. Regarding EEPROM memory, it was estimated that 1 KB is sufficient for the support of the functionalities where this kind of memory is needed.

The microprocessors to use should also include the following components:

- interface peripherals, such as SPI, I²C and UART, to support serial communication and interface with different communication modules;
- digital I/O pins (no less than 16), and some should support PWM generation;
• analog I/O ports, with at least 2 pins (with their respective Analog-To-Digital Converter (ADC)) for the control of analog peripherals;

• 2 timers at least (one to be used by the system and other being available for applications);

It is possible that some of the mentioned components won’t be needed in a certain context, like some of the interface peripherals or the ADC, for example, but their availability increases the overall flexibility of the system without a significant cost impact.

In today’s market, there are already some devices that suit the enumerated requirements, like the Arduino, WEMOS and NodeMcu boards represented in Figure 3.2.

![Arduino UNO board](image1)
![WEMOS D1 Mini board](image2)
![NodeMcu board](image3)

**Figure 3.2:** Control Modules - suitable hardware devices

DomoBus is a system conceived to support also the automation of Super Automated Houses (SAH), which are characterized by their high number of control points and, consequentially, a considerable amount of Control Modules. As such, the modules to use should be based on the most simple and economical microprocessors possible, balancing other factors such as power consumption, accessibility and support.

Taking into account all these aspects, one should choose devices based on 8-bit microprocessors in spite of 16-bit or 32-bit microprocessors. It is possible that 16-bit or 32-bit MCUs would present better features and a better performance. However, 8-bit microprocessors are enough for the implementation of the developed software and, in a general way, are simpler and cheaper than the others. It is also important to notice that the software developed for 8-bit microprocessors can also be implemented in 16-bit or 32-bit microprocessors.

One should also pay attention to the power consumption of the control devices. This is an especially delicate issue considering the Super Automated Homes context, characterized by its substantial quantity of Control Modules.

It’s also recommended that the chosen control devices are easy to acquire (particularly if one needs to swap devices as a consequence of a malfunction, for example) and also that the process of developing software for these devices is simple and well-supported. These features may be found within the Arduino boards scenario, making them strong candidates for being used as prototypes of Control Modules or Gateway Modules in the context of a DCN.
3.3 CM Software Architecture

3.3.1 Main Components

Figure 3.3 presents the proposed software architecture for DomoBus Control Modules.

The software architecture can be divided into 4 separate layers:

- **NET** - this layer is responsible for the packet exchange between different CMs on the DCN;

- **MSG (Message)** - it is responsible for the treatment of received messages and for the assembly of the messages to be sent to the DCN. These messages involve mainly reading, writing and notifying changes in the properties’ values of the devices in a CM;

- **PROP (Properties)** - this layer is responsible the management of the properties related to the various devices controlled by the CM. This layer presents some differences in comparison with the others as it merely consists of data structures and interface functions, isolating the control applications \( (APP_{k}) \) from the communication services;

- **Applications** - the last layer is constituted by the different applications that control the CM’s devices. Among them, it is important to emphasise the application SYS, implemented in all of the system’s Control Modules, whose functions are related to the supervision and management of the Control Module itself, instead of the control of devices.
3.3.2 Tasks and Applications

The different software components of the presented architecture can be classified into two different types: tasks and applications. The difference between these two types relies on the fact that the applications may be the origin or the destination of the messages exchanged within the DCN. This aspect establishes a clear separation between the NET, MSG and PROP components (that act like system tasks) and the control applications, with which is possible to interact from the outside, through the properties’ contents.

3.3.3 Operating Nucleus

To ensure a good modularity of the software and other mechanisms for multitasking, it was decided to incorporate in the software architecture a basic operating nucleus. The operation of the different tasks and control applications can be divided into two distinct phases called INIT and EXEC.

In the INIT phase, that occurs only once, all the tasks and applications execute sequentially a special function called init() that includes initialization actions that setup the Control Module, the connected hardware devices (both the devices to control and the external communication modules) and the data structures for the execution cycle EXEC.

![Figure 3.4: EXEC operation phase](image)

After the INIT phase, the CM software operation enters in the EXEC phase. This operation stage occurs uninterruptedly until the deactivation or the reset of the CM. In this phase, all the tasks and
applications execute another special function called task(), that implements the logic related to their role within the overall software architecture. Given the fact that there is only one single processor available, it was decided that the various task() functions are executed in this processor according to a non-preemptive scheduling in a "round-robin" fashion, as illustrated in Figure 3.4. The execution order by which the various task() functions are represented in this figure is merely representative.

The chosen scheduling is characterized by being non-preemptive and, consequently, without context switches. This means that the task() functions’ execution cannot be interrupted and one must wait until their completion. Like represented in Figure 3.4 the cooperative scheduler assigns the processor to the various task() functions sequentially and periodically. It is intended that any execution "lap" doesn’t last more than 1 ms. Considering that humans can only perceive differences in time on the order of 100 ms, this maximum cycling period constitutes a good choice. In case some applications require a more precise control of time or shorter reaction times, interrupts and a hardware timer can be used. Therefore, in the development of the task() functions, one must pay special attention to avoid occurrence of busy waits or blockages for significant amounts of time (regarding the 1 ms maximum cycling period), as their existence could compromise the well-functioning of the system.

In addition, every task() must control the number of operations executed in each cycle, in order to use only a fair share of the available processing power. For example, if the CM receives a message the NET_task() function will just handle the respective message reception process. In the following, the scheduler will call the MSG_task() function, that will solely perform the message analysis, and so on. This way, one guarantees that all the task() functions get their turn to execute in a timely way. Given these restrictions, the various tasks are typically implemented as state machines, as it will be described later.

In addition to the already mentioned tasks and applications, it is important to mention two more tasks: TIME and EEP.

The TIME task provides timing functions to the other tasks and applications, offering a multiplicity of (software) timers with different periods (1 ms, 10 ms, 100 ms, ...).

The EEP task, in turn, features functions for the management of the Control Module’s EEPROM memory, which is a global resource, but can be used by different tasks. In particular, the EEP task manages writing in the EEPROM, operation that can take a long time to execute.

The init() and task() functions constitute the basic structure of every software task and application. Therefore, it’s essential that every software task and application (including TIME and EEP) includes their own implementation of init() and task().

Finally, it’s important to notice that the PROP component has a different structure, given the fact that it is mainly composed by data structures and corresponding manipulation functions, which will be called mainly by the MSG layer and the applications, running in their processor time slots. However, it may include a PROP_init() function.
3.4 NET Layer

In the CM’s architecture, the NET layer corresponds to the software module directly responsible for the exchange of DomoBus Control Level (CL) data packets within the DomoBus Control Network (DCN).

The DCN can be implemented using different media, such as twisted pair cables, RF or even the house’s power supply network. Depending on the type of communication media, different transceivers and communication modules can be used. The NET software module will be responsible to interface with the communication hardware used, as illustrated in Figure 3.5.

![Figure 3.5: Interface between the NET layer and the DCN](image)

DomoBus Control Packets

Within a DCN, the packets exchanged have the structure illustrated in Figure 3.6.

![Figure 3.6: DomoBus Control Level packet (Source: [12])](image)

The first byte, called TLen, contains the number of bytes of the packet. The fields CDevDest and CDevOrig are address fields that identify the node, application and device the packet was sent/received to/from. The SNum field contains the sequence number of the packet. The CTR field specifies the message type and the operation to perform:

- **Message Types:**
  - Command - message specifies an action to be performed at the destination node;
  - Answers/ACK - message is an answer or acknowledgement of a previous Command;
• Operations:
  
  GET - obtain a property’s value;
  
  SET - modify a property's value;
  
  NOTIFY - report a new property's value;

  The Data field contains information regarding the type of the property (8_BIT, 16_BIT and D_ARRAY property), its ID and value. Finally, the last byte is a check byte used to ensure the integrity of the message in the communication process.

**NET-MSG Interface**

The CM includes one single communication buffer to store a message received or a message to be sent. It was decided to not use more buffers to support the communication process given that the developed software will be implemented in devices with very limited hardware features. The addition of more buffers would reduce the available memory, not to mention that the management of that set of buffers would be a more complex process that wouldn’t be rewarding because, as will be explained in the following, the chosen solution offers a good functional level.

This buffer is managed by the NET and MSG layers, with recourse to some flags that give information regarding the state of the buffer. This communication process was developed to minimize the impact of the communication process in the software execution and taking into account the fact that, physically, the node cannot receive and transmit messages simultaneously. The buffer control is based on the following rules, that depend on its contents:

• if the buffer contains an Answer/ACK message, it should be sent as soon as possible, to conclude a message exchange process. In case a new message arrives in the meantime, it will be discarded, as there is no place to store it;

• the MSG layer only writes a Command message in the buffer if it is empty. That message is for sending;

• if the node receives a message and the buffer contains a Command message to be sent, the received message will override the one on the buffer. This way, it is given priority to answer the received message and the Command that was on the buffer will be generated again and sent afterwards.

Most of the time, the node is waiting for a message to be received, switching out of this state to transmit Commands or Answers. The messages to transmit are generated in a short period of time, without causing any kind of impact in the node.
3.5 MSG Layer

The MSG layer corresponds to the CMs’ software section that constitutes an interface between the messages exchanged with the DCN and the properties associated with the devices controlled by the Control Modules.

In this layer, there are two main functionalities to be taken into consideration.

The first one consists in the treatment of the received messages and the assembly of the respective replies. This is a process that begins with integrity tests to validate the received data. More specifically, it is verified whether the node, application, property and device addressed in a received message are valid or not. If an error is found, a reply is sent back to the origin node, with an appropriate error code. On the contrary, if no issues are found in a received message, the next phase of the message treatment may proceed. This phase depends on the type of message received:

- if the received message is an Answer to a previously sent Command, this will be registered in the property management data structures. If by any chance the received reply isn’t associated with any of the controlled properties, the message is simply discarded;

- in case the message is a Command, an action in the addressed property/device pair is performed according to the received message’s type. As described in Chapter[2], if a GET message is received, a reply (called A_GET) is sent with the desired value. In case the received message was a SET, the property value is updated, and a reply is sent back signalling the success of the operation. The reception and treatment of NOTIFY Commands aren’t supported by the Control Modules.

The second functionality of the MSG layer consists on the assembly of the NOTIFY messages that are sent when a property value has changed. Although this is the typical procedure, it is also possible for a CM to interact directly with other CM, sending SET messages and, therefore, being able to actuate in their devices.

Let’s imagine one has a lamp whose light intensity is regulated by a pressure button. The button’s CM may immediately send a SET message to the lamp’s CM, instead of transmitting it to the Supervision Module, avoiding a potential delay that wouldn’t be convenient.

To simplify the generation of messages, a skeleton of each one of them is stored in the EEPROM. It was decided that the number of message skeletons stored in the EEPROM is equal to the total number of properties controlled by the Control Module, given the hardware limitations related to this memory. Each property is associated with one of the message skeletons stored in the EEPROM.

This method has one disadvantage: the properties that will cause the CM to send SET messages to other CMs can’t send NOTIFY messages to the Supervision Module. This way, the SM can only obtain their value by sending GET messages.
3.6 PROP Layer

As aforementioned, one of the key aspects of the DomoBus system is the fact that the controlled devices are treated as generic entities characterized by a set of properties. The devices’ properties are defined taking into account the characteristics of the devices to control. A lamp, for example, may be characterized by two properties, its activation status and the intensity of the emitted light. In case one wants to control a simple switch, one property correspondent to its commutation state is enough to characterize the device.

The PROP (Properties) layer corresponds to the software section responsible for the management of the properties of the various devices controlled by the Control Modules. In comparison with the other software modules, the PROP layer presents some substantial differences as it is simply constituted by a set of data structures and functions to interface those structures with the MSG layer and the control applications.

There are four types of properties handled by DomoBus:

- **8_BIT**: the most typical type. An 8_BIT property value is composed by a single byte of data;
- **16_BIT**: property value is composed by two bytes of data;
- **D_ARRAY**: this type’s name stands for “DomoBus_Array”. It consists of a byte array, and it is generally used when the property’s absolute value can’t be represented in the other forms. The first byte of a D_ARRAY property indicates the number of data bytes of the property value, as illustrated in Figure 3.7.

![Figure 3.7: D_ARRAY property](image)

All the properties of a CM are stored in specific arrays (one for each type of property). These arrays are accessed by the MSG layer and by the control applications, using an API.

To manage the access to the properties, each one is associated with a pair of control flags designated **events**, triggered every time a new value is assigned to a certain property. It is important to notice that there are two types of events to be considered:

1. the first kind of events, called **RX-events**, signals that a property was changed because the MSG layer processed a SET Command. These events will be used to inform the control applications which properties have changed and need to be processed;
2. the second type of events, designated **TX-events**, is associated with the changes in the properties’ value imposed by the control applications. As aforementioned, one of the tasks of the MSG
layer is precisely the communication of this kind of changes in the properties to the SM or to other CMs in the DCN. Therefore, these events are useful for this functionality as they point out to the MSG layer the properties that were modified. The messages to send as a consequence of the activation of TX-events, as referred above, are stored in the CM’s EEPROM. Again, only one of the stored messages is associated with the TX-event and vice-versa.

In a global way, the events are a useful feature as they prevent the performance penalty of a complete scan of the property data arrays for new values by both the MSG layer and the applications.

### 3.7 Applications

The applications (APPS) are the software modules that are actually responsible for the interface between the various sensors and actuators connected to the Control Module and their respective properties. In case the application reads sensors, it should update the correspondent properties with the new read values. On the other hand, concerning actuators, the applications act on the devices accordingly with the values on the corresponding properties.

One of the main objectives of this work is offering an API that eases the development of these applications by its programmers. With this API, the programmers don’t have to worry about the details that regard both the integration of the Control Module within the DCN and the communication with the other modules of the network. Evidently, as this is an open-source solution, all these specifications are completely accessible if one wants to gain insight into them or even to modify them to suit new needs, as long as the well-functioning of the CM and its integration within the DomoBus system aren’t compromised.

With this API, a programmer just needs to be concerned with the actual control of the various devices along with the update of the corresponding properties.

### SYS and Application Parameters

The SYS application is implemented in all the Control Modules of the DomoBus system. It is an important application, as it is responsible for the management and supervision of the Control Module itself.

The only device controlled by this application is a LED which is inherent to the CM, characterized by one property which is its number of blinks. This property may be used as a debug tool that gives information regarding the state of the CM at a certain moment in time.

In addition to the referred property, the SYS application controls a special type of properties called **Application Parameters (AP)**. The AP can be seen as the input specifications of the remaining control applications. As an example, let’s assume one has an application to control lamps called LAMP, and that this application has the capability to manage two types of lamps. By setting an appropriate AP, one may indicate to the LAMP application the types associated to the controlled
lamps, taking into account that the application may control more than one device simultaneously. This feature increases the overall flexibility of the system.

The other functionalities of SYS aren’t related to any property or device but are essential for the well-functioning of the Control Module. These include the assignment of a communication address to the node (this feature may also be considered to be a property) and the storage of the messages related to the TX-events in the EEPROM, in conjunction with the EEP module.

### 3.8 DCN Gateway

As depicted in Figure 3.1, the Gateway Modules (GM) are integral parts of the DomoBus Control Networks. These modules provide a connection between the Supervision Modules of the Supervision Level (SL) and the various Control Modules of a DCN. Consequently, in order to implement a fully operational DCN, there was a need to develop a software architecture for the DCN Gateway Modules.

The Gateway Modules’ hardware requirements are similar to the CM’s, with a few differences. A GM doesn’t control any kind of external peripherals except the hardware communication module used in the exchange of packets with the remaining nodes of the DCN.

The software architecture developed for the GMs is illustrated in Figure 3.8 where one can see that is composed by two main modules: NET and SM_COMM.

![Figure 3.8: Gateway Modules Software Architecture](image)

The NET layer has exactly the same function of the namesake layer in the Control Module’s software architecture: the exchange of packets between the GM and the DCN. Likewise, it’s code also has to be adapted to the communication features of the DCN, as this layer implements an interface between the GM and the used communication hardware module used.

The SM_COMM layer, in turn, is responsible for the exchange of data between the GM and the SM's Gateway application. Its code also needs to be adapted to the protocol used for the communic-
ation with the SM.

The SM_COMM layer and the NET layer share two communication buffers where the various messages exchanged in the Gateway Module are stored. One buffer is assigned for the storage of the messages received from the DCN that need to be sent to the SM, and the other one is used to save the messages received from the SM that need to be sent to the DCN. The contents of the buffers are controlled via some flags that give information regarding their state. It is important to point out that there is no established priority relationship between the messages that were received from the SM and the ones that came from the DCN.
Chapter 4

Implementation

In this chapter will be described all the details regarding the implementation of a prototype for both the Control Modules (CM) and Gateway Module (GM) of a DomoBus Control Network (DCN) in Arduino boards, as proposed in Chapter 3.

First of all, in Section 4.1, is performed a description of the microprocessor (ATmega328P) and of the communication modules (nRF24L01+ radio modules) chosen for this particular implementation. In Section 4.2, are described in detail the processes implemented in each one of the main software layers (NET, MESSAGE and PROPERTIES) of the Control Modules of a DCN, along with some considerations regarding the development of applications for these modules. Finally, in Section 4.3, is performed a description of the implementation details regarding the Gateway Modules’ software and of a Gateway application developed for a Supervision Module of the DomoBus system.

4.1 Hardware Selection

Hardware-wise, Arduino boards not only present the necessary components but also simplify several aspects of the prototype implementation. In fact, this platform provides a free open-source software development environment, the Arduino IDE that eases the adaptation to its inherent programming policy by individuals with practically no knowledge regarding software development and also by users with considerable programming skills. The Arduino platform and provide, with no additional cost, the tools and libraries which are necessary for the programming of the microprocessor incorporated in the boards and, by extension, of its components (avoiding some restraints imposed by the IDE).

The hardware device to use must be easy to acquire and/or to replace in case of malfunction. Nowadays, the Arduino platform is clearly one step ahead in comparison with the others, given its well established worldwide distribution network across the five continents.

Taking into account all the boards available in the Arduino platform, it was decided to implement the prototypes in Arduino UNO boards, for being the most simple boards of all the product line.

Now that the devices that will constitute the nodes of the DCN were chosen, one has to decide how the communication within the network will occur. In this particular implementation, it was decided that the communication between devices should be performed in a wireless fashion. In today’s market there are many modules that could support the desired type of communication.

One could use Bluetooth Low Energy (BLE) modules given their low energy consumption. However, these are difficult to use and there is a risk that these could not support the type of network to
Modules that provide Wi-Fi communication between devices, like WEMOS, can also be discarded. They present high energy consumption and the used protocol is too complex for the transmission of small data packets as intended.

nRF24L01+ radio modules, in turn are a good option. These are cheap and small radio modules which are fully compatible with Arduino boards, are simple to use and present low energy consumption rates. These modules also provide a set of helpful communication features like automatic packet validation and retransmission. Taking all these factors into account, nRF24L01+ radio modules were the modules chosen to support communication within a DCN. These radio modules will be described in Section 4.1.2.

4.1.1 ATmega328P

The Atmel ATmega328P microprocessor is the 8-bit MCU used in the Arduino platform boards that are based on 8-bit microprocessors, namely Arduino UNO, Nano and Mini.

From the ATmega328P’s datasheet, it’s possible to get the main features of this MCU and, by extension, of the Arduino boards used in the development of DomoBus Control Modules [13]:

- **Memory**
  
  32 KB of flash program memory, with 10000 write/erase cycles;

  2 KB **SRAM**;

  1 KB of EEPROM, with 100000 write/erase cycles;

- **Communication Interfaces**
  
  Two Master/Slave **SPI** Serial Interface;

  programmable Serial **USART**;

  Byte-oriented 2-wire Serial Interface, compatible with Philips **I²C**

- **I/O**
  
  23 programmable I/O lines;

  6 PWM channels;

- **Timers/Counters**
  
  Two 8-bit timers;

  One 16-bit timer;

These boards are especially useful for the implementation of a prototype as they provide a physical interface between the various sensors and actuators and the pins of the ATmega328P microprocessor. They’re also equipped with a power jack, a 16 MHz crystal oscillator and a USB connection port (based on an USB-to-serial converter programmed in an ATmega16U2 **MCU**). This USB connection is particularly useful as one can use it for programming the MCU, for communication purposes, or
even as a simple power supply. The Arduino UNO board also contains an ICSP header that enables the programming of the ATmega328P through external devices called programmers.

**Software Development**

The development of software for the ATmega328P, using the Arduino boards, may be performed in two manners.

The most basic and easy method consists of using the Arduino IDE to write and compile the code and to load the respective program into the device. The programming language is a variant of the C language, in which the program is based on two main functions called `setup()`, which is used for initialization purposes, and `loop()`, which is an infinite loop that is the main execution cycle. The available libraries include functions that provide an easy and intuitive access to the microprocessor components (serial port, EEPROM, SPI, etc.), and the incorporated Serial Monitor allows an easy debug. However, this method has its own limitations. It isn’t possible to divide the code of the program into different files or to include code from other files (unless one writes a function library), which means that programming a complex set of sensors and actuators may result in an unreadable program with thousands of code lines. Besides, the libraries provided by the IDE include code for all the Arduino boards and mask several implementation details. This way, one may load the flash memory with unnecessary data, related to other boards, and doesn’t even have a full control of the code that is actually loaded into the device.

The second method consists of programming the ATmega328P microprocessor directly, in pure C language. The code is compiled using the AVR-GCC compiler and its loaded to the board using a special program called `avrdude`. Programming in this manner is a more complicated process. Besides the lack of debug tools (except for the incorporated LED), adapting to the provided libraries and extensions imply studying the microprocessor control registers and the functions of each one of their bits. Besides, the compilation and loading processes are based on a complicated set of commands with many control flags. However difficult this method may be, there are some advantages on choosing this programming method. In opposition to what happens if one uses the IDE, it’s possible to divide the code into several code and header files, giving modularity to the program, the same way it happens in all C programs. In addition, one has a full control of the code that is loaded into the flash memory, avoiding the inclusion of unnecessary data related to other boards. Besides, one has a full control of the microprocessor’s control registers.

In the development of the modular software architectures for the Control Modules and Gateway Modules of a DCN, the second programming method was naturally chosen, independently of its disadvantages, given the modularity and control it brings to the developed software.

### 4.1.2 nRF24L01+ Communication Module

The nRF24L01+ radio modules, produced by Nordic Semiconductor and represented in Figure 4.1, were chosen to support the desired type of communication for being simple, small and cheap radio transceivers which are completely compatible with the Arduino UNO boards.
According to [15], the nRF24L01+ radio modules’ main features are the following:

- **RF Data Transmission Rates** - 1 Mbps or 2 Mbps;
- **Operation Frequencies** - the modules operate in the 2.4 GHz [ISM] band. This corresponds to a range of frequencies that goes from 2.4 GHz to 2.525 GHz;
- **Non-Overlapping RF Channels** - 126 channels when operating at 1Mbps and 63 when operating at 2Mbps;
- **Transmission Power** - the output power goes from -18 dBm to 0 dBm (the power verified in Bluetooth standard radios);
- **Host Interface** - 4-pin hardware SPI, that supports a maximum data transmission rate of 8 Mbps. Three separate TX and RX FIFOs, with a capacity of 32 bytes, and 5V tolerant inputs are also included;
- **Payload Maximum Length** - 32 bytes.

These radio modules are connected to the CM microprocessor through the SPI protocol. The control of the nRF24L01+ modules is based on a set of commands that allow not only the transfer of the data to the DCN but also the handling of the modules’ control registers. This manipulation allows the users to set the operation mode and the communication parameters such as the communication addresses, the data transfer speed, the data integrity check mode and the RF operation channel, among others.

The communication between radio modules is supported by an embedded baseband protocol engine called Enhanced Shockburst, that includes features such as Auto-Acknowledgement and Auto-Retransmission of data packets. The respective data frames’ payload has a maximum length of 32 bytes. As a consequence, using these modules to support the inter-device communication in a DCN implies that this is the greatest number of bytes that the exchanged messages may have. This value is sufficient given the functionalities one wishes to implement in Control Modules and Gateway Modules of a DCN.

Finally, it’s essential to point out that the nRF24L01+ radio modules feature five different operation modes:

- **Power Down** - the device is disabled;
• **Standby-I** - low-energy consumption mode in which the device is enabled, but no communication operation is taking place;

• **Standby-II** - operation mode in which the module is set to transmit packets but has an empty TX FIFO;

• **RX** - radio is able to receive messages;

• **TX** - mode in which the radio enters to transmit a packet.

The way the operation modes are used in the CMs’ and GM’s software will be described below.

### 4.2 Control Modules Components

This section presents the details of the implementation of the software architecture for the Domobus Control Modules, presented in Chapter 3, in an ATmega328P microprocessor incorporated in an Arduino UNO board. As one may recall, this architecture is constituted by three components, called NET, MSG and PROP and by the different applications that directly control the home devices and their properties. All these tasks and applications implement two main functions: *init()*, for their initialization, and *task()*, that implements the logic related to their role in the overall architecture. All the *task()* functions are executed sequentially and periodically in a cooperative multitasking scheduling as explained in Chapter 3.

#### 4.2.1 NET Task

The NET task represents the software component that exchanges data with other nodes of the DCN, using a chosen communication module.

**nRF24L01+ Initialization**

The first operation performed in the NET task is the initialization and configuration of the nRF24L01+ radio module, preparing it for the communication operations.

In order to proceed with the configuration actions of the radio modules, it’s necessary to perform a previous activation and configuration of the ATmega328P SPI. By manipulating the bits of the respective control registers, the SCK clock rate was set to 4 MHz, which implies a data transmission rate of 4 Mbps. At this pace, it is expected that the exchange of data between the radio modules and the microprocessor won’t last more than 100 µs, including the associated processing time. A maximum frequency of 8 MHz could be chosen. However, setting this transmission speed could easily result in the occurrence of noise in the data exchange process, due to the fact that the radios aren’t soldered to the microprocessor, and the connection wires have a significant length.

After the conclusion of these initial configurations, in which the radio is set to operate in Standby-I mode, one may advance to the configuration of the various features of the radio modules:
• **Data-Integrity Check protocol:** 16-bit CRC. The nRF24l01+ modules also make available an 8-bit CRC, that would be sufficient, but the 16-bit CRC is more reliable;

• **Packet Retransmission:** the radio modules include an implemented automatic Acknowledge system. If an Acknowledge isn’t received, the packets are retransmitted every 250 µs. A maximum of 4 retransmissions may occur;

• **Data Transmission Rate:** 1 Mbps;

• **Transmission Power:** -12 dBm - At this power, the radio modules are able to communicate with one another at a distance of around 10 meters in a straight line with no obstacles, which is enough for the prototypes’ implementation. To operate with a greater transmission power, it is necessary to weld a capacitor to the radio module given some limitations of the Arduino’s 3,3V pin related to its electric current capability;

• **Radio Operation Mode:** RX - by setting this operation mode, the radio modules are ready for the reception of data at the end of the configuration process;

Finally, after these configuration operations, the communication addresses are set. In [15] one may find that the Address Field of the Enhanced Shockburst data frame is 3 to 5 bytes long. In this particular implementation, it was decided that the 3-byte configuration would be sufficient for the support of the communications within a DCN. Opting for the other configurations would result in the transmission of unnecessary bytes.

Given the selected configuration, it was established that the addresses’ least significant byte would represent a node number within the DCN and that the second least significant byte would contain the DCN identification number. The most significant byte isn’t used, but it may become necessary in future developments of DCNs that use these communication modules. The node address and the DCN identification number are defined in constants declared in one of the software’s header files.

**NET-MSG Interface**

The NET and MSG layers share a communication buffer in which the received messages and the messages to transmit are stored. Given the limitations on the byte size of the exchanged data packets imposed by the nRF24L01+ radio modules, this buffer consists of an array of 32 bytes.

The control of the communication buffer by the NET and MSG tasks is performed with recourse to a set of control bits, grouped in one single byte, that are activated according to the buffer’s state and to the type of data contained inside it. There is a total of six control flags to be taken into consideration:

• **IDLE** - the buffer is free to be used for the reception or transmission of data;

• **BUSY** - the buffer is being used to store received data or to assemble a new message to be sent;

• **TX** - the buffer contains a Command message to be transmitted;
• **RX** - the buffer contains a received message;

• **REPLY** - the buffer contains an Answer/ACK message.

With the settlement of these buffer states, and with the assignment of a degree of priority to each one of them, it's possible to establish a set of rules for the access to the buffer regarding the transmission and reception of data.

Whenever the NET and MSG tasks try to access the buffer, they verify whether the buffer is available or not. This availability depends on the activated control flag and also on the type of data that it is intended to load into the buffer:

• when the buffer is in the IDLE state, any kind of message may be loaded into the buffer;

• if the BUSY flag is activated, the buffer is being loaded with data. The NET and MSG tasks must wait until the loading process gets finished in order to check whether the placement of new data is possible or not. As the buffer is accessed by one task at a time, this situation never occurs in this particular implementation;

• if the TX flag is activated, the buffer may be overwritten in order to store messages received in the NET layer or Answer/ACK messages;

• when the RX flag is triggered, only an Answer/ACK message may still be loaded into the buffer.

• if the REPLY flag is activated, no other messages may be placed into the buffer. This way, sending Answer/ACK messages may be considered the biggest priority of the Control Modules communication process.

These rules were established taking into account that the reception of new data in the NET layer has a priority over the transmission of messages, with the exception of Answer/ACK messages.

The messages that couldn’t be loaded into the buffer because of the established rules are automatically discarded. This is what occurs when, for example, when one tries to load a received message when the REPLY flag is activated.

After loading a message into the buffer, the flag correspondent to the type of message contained in the buffer is activated (TX, RX or REPLY). The buffer returns to the IDLE state when the communication task related to the activated flag gets finished.

**Transmission and Reception of Messages**

In the NET task occur all the actual data exchange operations between the DCN and the CM, through the communication module. Upon execution, only one message reception or transmission is performed, given the cooperative multitasking scheduling used in the CM’s software execution. The communication operation to perform depends on the activated control flag of the communication buffer shared between the NET and MSG tasks, taking into account the rules mentioned above.

Regarding the transmission of data, the first step taken is the execution of the Checksum algorithm, storing the result in the frame’s last byte. Next, the transmission address is set to match the address
of the node one wants to send the message to. Afterwards, the radio module is set to operate in TX mode, the data frame is transferred to the radio’s TX FIFO and the transmission starts. At the end of the transmission, the radio module is set to operate once again in RX mode.

Concerning the reception of data, one checks periodically if the radio modules have received any message, taking into account that these modules include a FIFO that can store a maximum of 3 received messages. If positive, a received packet is loaded into the buffer. Afterwards, some integrity tests are performed in the data frame. If an anomaly is detected, the message is discarded. Otherwise, the message may be treated in the MSG task.

### 4.2.2 PROP Layer

The PROP layer is significantly different from all the others, mainly because it hasn’t its own implementation of the init() and task() functions. This layer contains all the data structures used for the storage of the properties that characterize the devices controlled by the Control Module. The control and supervision of the various data structures are performed by the MSG task in conjunction with the different control applications.

The property data, stored in arrays, is arranged according to the structure depicted in Figure 4.2. Three data arrays were allocated given the three possible property types: 8_BIT, 16_BIT and D_ARRAY.

![Figure 4.2: Basic structure of the data arrays](image)

Each array may be divided into sections related to the various control applications that, in turn, are associated to a certain type of controllable device. Each one of these sections may be split into sub-sections taking into account the different properties associated with the respective control application. These sub-sections contain the actual data. The array elements in these sub-sections depend on the number of devices controlled by each application.

In order to get a better understanding of the applied reasoning, let’s assume one has a CM that controls four lamps and two switches, for example. Each lamp is characterized by 2 properties: its activation status (8_BIT) and its emitted light intensity (16_BIT). On the other hand, each switch
features only one property, related to its commutation status (8_BIT). This means that this Control Module controls a total of six 8_BIT properties and two 16_BIT properties. The layout of the resultant property arrays is represented in Figure 4.3.

It is important to point out that the array that contains the values of the D_ARRAY properties, although it presents the same basic structure, is slightly different from the others. This occurs given that each element of this array is itself an array of bytes. As such, the access to the property values and their modification occurs in a different manner, as one has to take into account the dimension of each property.

### 4.2.3 MSG Task

The MSG task corresponds to the software section in which occurs a direct interaction between the DomoBus messages exchanged in the NET task and the properties of the controlled devices contained in the PROP layer's data structures. In this interaction there are four main functionalities to be considered, namely the treatment of the received data packets, the assembly of Answers/ACK messages to the received Commands, the loading of the Commands to be transmitted, and the management of the events mechanism that is used to that a modification occurred in a property's value.

In order to get a better understanding of the operations performed in this task, it is essential to perform a prior detailed analysis of the DomoBus Control Level Data Frames' structure.

#### DomoBus Control Level Data Frames

The DomoBus Control Level Data Frame, represented in Figure 4.4, is constituted by seven different fields.

The value of the first one, called TLen, is the total size of the data frame. This value depends on the number of bytes of the data payload. In this particular implementation, the maximum value admitted
for the TLen field is 32, given that this is the maximum number of bytes that can be transmitted through the nRF24L01+ radio modules.

The next four bytes are the destination (CDevDest) and origin (CDevOrig) address fields. Both fields share the same structure, depicted in Figure 4.5. The first byte, called NAddr, is used to identify a node inside the network. The second byte is divided into two separate sub-fields. The first one, designated NApp, is a three-bit number that identifies an application inside the node. The other sub-field, called ADev, is a five-bit number that identifies a device controlled by the identified application.

![Figure 4.5: Data frame address fields (Source: [12])](image)

The next field, SNum, is the message sequence number. This number is used to verify whether the Answers/ACKs received are related to a previously sent Command or not. A sequence number is assigned to each property. In the CM initialization process, all the sequence numbers are set to zero. When a CM sends a Command, the sequence number assigned to the property whose value has changed (the Command messages sent by CMs are related to changes in property values) is incremented. This value is then placed in the SNum field of the packet. The Answer/ACK received as a response to the sent command must have the same sequence number. Otherwise, the message is discarded.

The CTR field is constituted by many sub-fields, as depicted in Figure 4.6. The most important one is the 3-bit OpCode sub-field, by which the message type is defined among the three possible types already described in Chapter 2: 000 is for GET messages, 001 for SET messages and 010 for NOTIFY messages. The five remaining bit combinations are reserved, allowing the creation of new message types if needed.

![Figure 4.6: Data frame CTR field (Source: [12])](image)

The CTR field also contains two flags, Error and ACK, that provide information regarding the message status:

- the ACK flag indicates whether a certain message is an Answer/ACK message (A=1) or a Command message (A=0). Every time a node receives a Command message, an Answer/ACK must be sent back to the node that has sent the original message;
- the Error flag is an indicator of the occurrence of any kind of problems in the treatment of a received message. Naturally, it only makes sense to activate this flag in Answer/ACK messages,
along with the ACK flag. Each error can be identified by its unique error code, that is inserted in the message’s last data byte. A complete list of errors and their respective opcodes can be found in Appendix A.

In respect to the data field, there are two sub-fields to be taken into consideration: the Property Descriptors and the actual data bytes.

As already mentioned, in DomoBus the devices to control are treated as generic entities characterized by a set of properties, manipulated by the messages described above. As such, within the DomoBus messages’ context, it’s essential to provide information concerning the properties involved in the exchange of data. This can is done by using special bytes called property descriptors, that have the structure depicted in Figure 4.7. By analysing this picture, it is possible to identify three separate sections:

![Property descriptor](Source: [12])

- Property Type - 2-bit number that identifies the property value type, among the four possible categories: 00 for 8-bit integers, 01 for 16-bit integers, 10 for 32-bit integers and 11 for byte arrays;
- I (Invalid flag) - an error may occur when reading a sensor, causing the correspondent property value to be invalid when updated. This flag is activated in these situations; this flag is activated if an error occurred while reading a sensor and property's value isn't valid;
- Property ID - this is the property identifier. It’s a 5-bit value, which means that a device may have a maximum of 32 different controllable properties of each type. This value is used along with NApp and ADev, from the CDevDest field, in order to find the property to control inside the property data structures of the Control Module.

The number of Property Descriptors in the message depends on the message type, as represented in Figure 4.8. In the case of GET and SET messages, two descriptors are needed: one for the property that originated the message and another one for its target property. In case of NOTIFY messages, one only needs the descriptor of the property whose value has changed.
The remainder of the message's data field is filled with the actual data bytes to send/receive. The number of bytes of this field is dependant on the property type. Naturally, one only needs one byte for an 8-bit integer property, for example. In case the data corresponds to a byte array property, the first data byte indicates the size of the array.

Finally, the last field of the entire message is the data integrity byte, used to ensure the integrity of the message along the communication channel. Various algorithms may be used, from CRC to Checksum. In this particular implementation, a Checksum protocol is used given that the nRF24L01+ radio modules perform an 8-bit or 16-bit CRC integrity check in the inter-device communication process.

Treatment of Received Messages

One of the main functionalities of the MSG task is the treatment of the DomOBus Control Level Data Frames received in the NET layer.

The first step of this process consists in the computation of the index of the property to which the received message is destined. This computation is based on the destination values contained in the received packet, namely the application number, device number, property type and property ID. One needs to get the destination property's type not only because in the PROP layer there are data structures for each one of the property types, but also because the index computation process for the D.ARRAY properties is different in comparison with what happens with the other property types.

The next step depends on whether the received message is a Command or an Answer/ACK, a fact indicated by the flag ACK in the CTR field of the data frame.

In case the received message is an Answer/ACK, its treatment process gets finished after disabling the TX-event that generated the respective Command.

If the received message is a request, the treatment process proceeds with the execution of the action indicated by the Opcode of the CTR field of the data frame. If one receives a GET Command, the destination property's value is retrieved from the respective data structure. On the contrary, in case one receives a SET Command, the value contained in the frame's data field is loaded into the property values' data array, and the respective RX-event is triggered. After these operations, the respective Answer/ACK message is built in the buffer and treatment process gets terminated.
**Answer/ACK Messages**

Answer/ACK messages are sent by the CM as a response to the received Commands or in case an error occurs in the treatment process.

Independently of the cause, the assembly of this kind of messages is a process that goes through the same basic steps. Taking into consideration the original received message, the CDevDest and CDevOrig fields are swapped. Naturally, the ACK flag in the CTR field is activated. If the reply is mounted as a response to successfully treated GET and SET Commands, the process may enter in its terminal phase. In the particular case of the GET Commands, the solicited data has to be previously put into the Answer’s Data field.

If the Answer has to be sent because an error occurred in the treatment of a received message, some additional steps occur. The ERROR flag of the CTR field is activated, and the respective error code is placed in the last byte of the Data field that, in turn, gets overwritten. A complete list of errors may be found in Appendix A.

**Events**

The events are the control bytes that are used to signal that modifications occurred in properties’ values. There are two types of events to take into consideration, namely the TX-events, that signal the alterations caused by the control applications, and the RX-events, which indicate the changes caused by received SET Commands. A TX-event and a RX-event are assigned to each property. In this particular implementation, the two types of events were implemented in one single byte, as depicted in Figure 4.9.

Regarding the RX-events, these are implemented in the event’s second most significant bit. One must activate this bit when activating the RX-event.

Concerning the TX-events, there are more aspects to take into consideration given that, upon activation, the CM must send a message to another node of the respective DCN to point out the modification that occurred in the associated property. The bits assigned to the TX-events are used not only to indicate that this kind of alteration occurred but also to give information regarding the number of retransmissions of the message to send, if needed. The event’s most significant bit is a control flag called SEND_NOW, activated as soon as the modification in the property value occurs, that remains active until the message gets transmitted. The least significant bits are used to count the message retransmissions, that must occur if the CM doesn’t receive an Answer/ACK to a previously
sent command in a given time period. This time interval is set in an auxiliary array, in which the retransmission timeouts are managed.

It is important to point out that the implemented retransmission mechanism is used to support end-to-end communication. This means that one may use multiple links in the communication process, and some retransmissions may occur in each link, that won’t interfere with the TX-events and the retransmission timeouts of the node that sent the message. This way, the value value chosen for the retransmission timeouts must be adapted to this kind of circumstances.

A TX-event and a RX-event associated to a certain property can’t be activated simultaneously. It is important to point out that, given that the reception of Command messages has priority over their transmission, the activation of an RX-event may result in the deactivation of the associated TX-event (in case this was previously activated). The opposite situation never occurs:

• regarding sensors, their properties are considered to be outputs of the respective control applications, which means that SET messages have no effect on their properties’ values and, consequently, the associated RX-events never get activated;

• concerning actuators, the properties are considered to be inputs of the control applications, whose values are changed by received SET requests. When reading a new property’s value, the application deactivates the respective RX-event, treats it, and activates the respective TX-event at the end of the process;

In order to store and arrange the indexes of the activated events, some auxiliary lists, called Lists of Activated Events (LAE), were implemented in a single byte array. These lists are used in the management of all the events, regardless the property type. As such, the size of the array is equal to the total number of properties controlled by the CM. This approach was chosen in order to avoid the performance of a full scan of the events’ arrays every time one wants to check for activated events and to guarantee that these are treated according to the order by which they were activated. There are three types of LAE to take into consideration:

• Send_Now - contains all the TX-events with an activated SEND_NOW flag and also the ones associated to messages to retransmit;

• Retransmission - contains the remaining TX-events, associated with messages that are awaiting to be retransmitted;

• RX - contains all the activated RX-events. There is one RX LAE per control application.

Each element of the list contains the index of another element, which is correspondent to the event that was activated immediately after. As an example, if cell number 2’s value is equal to “1”, this means that event number 1 was triggered immediately after the activation of event number 2. The elements considered to be the first (Head) and the last (Tail) of the list are stored in a pair of auxiliary bytes. The hexadecimal value 0xFF is assigned to any element that isn’t in use or to the elements considered to be the last entries of the respective lists.
To understand this reasoning, let's consider, as an example, that a certain CM controls a total of 8 properties, regardless the property type. In addition let's assume that, initially, all the events are disabled but, then, the TX-events 5, 2 and 3 are activated sequentially and placed in the Send_Now list. In Figure 4.10 is represented what occurs in the array as a consequence of these activations.

![Figure 4.10: List of Activated Events - Functioning Example](image)

Next, let's consider that the MSG layer is able to treat the TX-events 5 and 2, by sending the respective Commands. Regarding the LAE, the events placed in the Send_Now list are shifted sequentially to the Retransmission list. Afterwards, one receives an Answer that causes the deactivation of the TX-event number 2. Next, a SET message related to the TX-event number 5 is also received, disabling the TX-event and activating the respective RX-event (that will cause the event to be placed in its RX list). In figure 4.11 one may find what occurs in the array of the LAE as the operations mentioned above occur.
Retransmission Timeouts

The Command messages sent by the CM as a consequence of the activation of TX-events are retransmitted if the respective Answer/ACK isn’t received within a certain time interval. This time intervals, regardless the property type, are defined upon the activation of the event in a special array allocated for the effect. As such, its size is equal to the number of properties controlled by the CM. The contents of this array are managed by the MSG task in conjunction with the TIME task.

In the timeouts array, time is measured in a relative manner. Each cell contains the number of time intervals that occur after the timeout associated with the event that was activated immediately before reaches zero. An auxiliary variable contains the sum of all the timeouts associated with activated events. If one wishes to set a retransmission timeout of 25 time units, and if the sum variable is equal...
to 20, the value to insert in the timeouts array is 5.

To get a better understanding of the principles applied, let's consider the example depicted in Figure 4.12. In this example, it was assumed that a retransmission timeout of 25 units of time is assigned to every TX-event upon entering the Retransmission list. Figure 4.12 represents what occurs in the time array when the TX-events 5 and 2 placed, sequentially, in the Retransmission list, assuming some time has elapsed between the events' activation.

The supervision of the retransmission timeouts occurs every time the MSG task is executed. In this process, as illustrated in Figure 4.12, the time elapsed between two consecutive executions of the MSG task is subtracted from the timeout value associated with the head of the Retransmission list and from the "sum" variable. If the value to subtract is greater or equal to the head's timeout value, the respective event is removed from the Retransmission list and placed again in the Send_Now list.

**TX-Events Supervision**

The control of the activated TX-events is one of the main functionalities of the MSG task. This is a process composed of two different phases, namely the management of the retransmission timeouts, described above, and the assembly of a Command message related to the event which is the Head of the Send_Now list.

The assembly of the messages associated with the head of the Send_Now list may only be performed if the buffer is empty. Otherwise, this process stops for now but it will be repeated the next time this task runs.

After the selection of the TX-event, the skeleton of the associated message is retrieved from the EEPROM and loaded into the buffer. If the message is already prepared for its transmission, the assembly process terminates. If not, the necessary operations are performed (normally the filling of
the Data field of the packet with the property's current value). Upon the completion of this process, the TX buffer control flag is activated.

4.2.4 Control Applications

The various sensors and actuators are directly managed by control applications. Given the DomoBus system context, this control is based on the devices’ properties, that must be defined by the user according to the devices’ characteristics. The interaction with the devices’ properties depends on the type of application:

- **sensor applications** - in this type of applications, the properties are seen as output values. Throughout the execution cycle, these applications perform modifications in the associated property values according to what is read from the sensors. These alterations may cause the activation of the respective TX-events;

- **actuator applications** - concerning actuators, the properties are seen as input values, changed by received SET messages. As such, upon execution, these applications must verify whether the properties they control were changed or not. If they were, the applications perform actions in the respective actuators according to this new values. Finally, they send a message to the node that sent the original request to confirm that the new property values were treated.

An API was developed to allow a simple and intuitive interaction between the applications and the property values. This way, the application programmers don’t have to be concerned about the implementation details of the CM's software, along with the communication specifications, only having to understand the basic operating principles of the DomoBus system.

Control Module Configuration

All the data structures, pointers and constants related to the properties and the events, along with some other configuration elements, are declared in a separate header file. The contents of this file are dependent on the control applications one wishes to implement in the CM and the number of devices and properties controlled by each one of them.

The information needed for the generation of this header file is provided by the user/programmer of the Control Module, through a configuration text file. In this file, the user writes in each line all the information related to each control application. The different pieces of information are separated by tabs, organized into 6 different columns:

- Column 0 - application name;
- Column 1 - number of devices controlled by the application;
- Column 2 - device type. A capital "S" indicates sensor applications and a capital "A" indicates actuator applications;
• Column 3 - number of 8_BIT properties;
• Column 4 - number of 16_BIT properties;
• Column 5 - number of D_ARRAY properties;
• Column 6 - properties' designations, separated by space characters. In case of D_ARRAY properties, their dimension is indicated by adding to the name a colon followed by the dimension value (property name :<array dimension>).

In Figure 4.13 one may find an example of one of these configuration files. The lines that start with exclamation marks are used for comments, and the dot indicates the end of the file.

```
<table>
<thead>
<tr>
<th>Control Module (CH) 01</th>
</tr>
</thead>
<tbody>
<tr>
<td>app type dev prop8 prop16 prop_s prop_names(max_useful_len)</td>
</tr>
<tr>
<td>SYS A 1 1 0 0 pulses</td>
</tr>
<tr>
<td>LWM A 2 1 0 0 state</td>
</tr>
<tr>
<td>SWF A 2 1 0 0 state</td>
</tr>
<tr>
<td>TST A 2 2 1 3 state intensity power code:14 id:4 rgb:8</td>
</tr>
</tbody>
</table>
```

**Figure 4.13:** Example of an input configuration file

A special program was developed to generate the header file with the CM configurations from the input text file. In addition, this program also generates in a separate folder the code and header files with a skeleton of the control applications. The created code files include some useful information, such as some constants, function macros, variable masks, and also basic versions of the init() and task() functions, to ease the application development process. The software's main file, with the cooperative multitask scheduler and a compilation batch file, that includes all the commands needed for the compilation and loading of the software to the board, are also generated automatically.

**API**

The developed API consists of a set of macros that allow the interaction with the property values using the names assigned to the applications and properties by the user in the configuration input file, like the one represented in Figure 4.13. There are four function types to take into consideration: `set_and_send()`, `send()`, `get_new()` and `process()`.

The `set_and_send()` function is used to perform modifications in the property values. Its arguments are the number of the affected device and the value to write in the data structures. If this value is different than the one contained in the data structures, the CM will also send a message that indicates the occurrence of this modification. In a general way, this function is used in applications for sensors.

The `get_new()` function, in turn, is used to get the details (type, ID and device number) of the actuators’ properties that got their value changed by SET messages. This function is only used in actuator applications and is implemented in a separate file for convenience, as programmers should not perform modifications in this function. Calling this function is the first operation performed by the task() function of the respective application. With the obtained information, a programmer may access the property arrays in order to retrieve the desired value, taking into account the following format:
It is also possible to modify property values in this fashion. However, that isn’t recommended as the respective TX-events won’t get activated. In Table 4.1 one may find some examples of this kind of access to the property data structures.

<table>
<thead>
<tr>
<th>Code</th>
<th>Application</th>
<th>Property</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAMP_intensity[1]</td>
<td>LAMP</td>
<td>intensity</td>
<td>1</td>
</tr>
<tr>
<td>SWT_state[0]</td>
<td>SWT</td>
<td>state</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1: Access to the Property Values - Examples

The process() function is used to treat a property value associated with the application, device and property ID obtained with the get_new() function. This function receives the obtained device number as an argument. The process() function is essential for actuator applications to process property values modified by SET messages. This function may be used in sensor applications, albeit they aren’t needed in this context, and therefore is treated as a normal function.

Finally, the send() function is used to confirm that the property values changed by SET messages were treated by the applications that control the respective actuators. Its argument is the number of the affected device.

The process() and send() functions related to the modified property are called in the get_new() function. This way, a programmer doesn’t have to call extra process() and send() functions for the treatment of new property values.

In the declaration of any of the aforementioned functions, the application, function and property names are arranged in the following manner:

\[
< \text{application} > \_< \text{function} > \_< \text{property} > \_ (\text{arguments})
\]

In the get_new() function, the field where one inserts the property name is discarded as it’s unnecessary.

Let’s assume, as an example, that one is developing an application called “APP” that controls a property designated “temperature”. In Table 4.2 one may find how to use the mentioned functions in the application development, regardless the device type.

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_and_send()</td>
<td>Device, Value</td>
<td>APP_set_and_send_temperature(dev, value)</td>
</tr>
<tr>
<td>process()</td>
<td>Device</td>
<td>APP_process_temperature(dev)</td>
</tr>
<tr>
<td>send()</td>
<td>Device</td>
<td>APP_send_temperature(dev)</td>
</tr>
<tr>
<td>get_new()</td>
<td>(None)</td>
<td>APP_get_temperature()</td>
</tr>
</tbody>
</table>

Table 4.2: Implementation Examples of the API Functions
4.3 Gateway Implementation

The Gateway application of the Supervision Modules and the Gateway Modules are the components of the DomoBus system that support the interaction between the Supervision Level and the Control Level.

The GMs’ software was implemented in a ATmega328P microprocessor on an Arduino UNO board. Regarding the Gateway application, this was implemented in a PC. These two devices are connected by an USB cable, that supports a serial communication channel between them.

4.3.1 Control-Level Gateway

The Control-Level Gateway Module’s software is composed by two layers, NET and SM_COMM, that are executed sequentially in a non-preemptive multitasking without context switches, the same way it happens in the Control Modules. The communication operations that occur in each one of these tasks are supported by two communication buffers, one for the storage of the messages received from the Supervision Module (SL-buffer) and one to store the messages received from the DCN (CL-buffer).

The control of the two communication buffers, performed by both software tasks, is based on a set of control flags implemented in two separate bytes, one for each buffer.

The following states are admitted for each one of these buffers:

- **IDLE** - the buffer is empty or contains no relevant data;
- **BUSY** - the buffer is being loaded with data;
- **FULL** - the buffer contains a message that has to be transmitted;

Regarding the NET task, this is where occurs the exchange of data between the GM and the CMs of the DCN, through the nRF24L01+ radio modules. This task’s operating principles are practically the same as the NET task of the Control Modules’ software. The main difference relies on the fact that upon execution two operations are performed sequentially, namely the transmission of a message contained in the SL-buffer and the storage of a message received by the radio modules in the CL-buffer.

**SM_COMM Task**

In the SM_COMM task occur all the data exchange operations between the GM and the SM to whom it’s connected. In this task, two operations are performed sequentially, namely the storage of a message received from the SM in the SL-buffer and the transmission of a message contained in the CL-buffer to the SM.

As aforementioned, the GM is connected to the PC used as a Supervision Module via USB. In the Arduino UNO board used as a GM, this connection is supported by an USB-to-Serial converter implemented in an ATmega16U2 microprocessor, connected to the ATmega328P via USART.
The SM_COMM task handles the initialization and configuration of the ATmega328P USART, which is used to communicate with the SM. In this process, in which the USART is set to operate in asynchronous mode, the most important factor to define is the baud rate, which is the rate at which the data is transmitted/received.

In both the microprocessor and the USB-to-Serial converter, the available baud rate values depend on the frequency of the system clock. In this particular case, this signal is generated by a 16 MHz crystal oscillator that is also incorporated in the Arduino UNO board. At this frequency, one may apply a baud rate value from 2400 bps to 1 Mbps in both of them [13] [16]. Given these values, one must take into account the fact that the baud rate generators can’t always perform an exact division of the oscillator frequency in order to obtain the desired baud rate. By analysing the datasheets, it’s possible to verify that this situation won’t occur if one sets a baud rate of 250 kbps, 500 kbps and 1 Mbps. It is desired that the transmission and reception of data occur as fast as possible. Therefore, 1 Mbps was the value chosen for the baud rate. At this baud rate, the transmission of a 32-byte message, given that each byte is composed of 8 bits, will last 256 $\mu$s , for example. This means that one can exchange, approximately, a total of 3900 messages per second, which is an appropriate value.

4.3.2 Supervision-Level Gateway Application

Regarding the Supervision Level of the DomoBus system, it was developed a prototype of a Gateway application, implemented in a Windows PC, through which it’s possible to test the software implemented in the Gateway Modules and Control Modules of a DomoBus Control Network. This application handles all the aspects related to the communication between the computer and the GM and also provides a simple user interface (UI) that allows the interaction with the devices controlled by the Control Modules through the computer's command prompt. With this UI, a user may easily interact with the devices controlled by the Control Modules of a DCN without having to know the implementation details of the DomoBus system, such as the structure of the Control Level data frames. The developed UI includes the following features:

- shows all the information contained in the messages received from the GM;
- assembles data frames based on information provided by the user. In the following, the assembled data frames and sends them to the GM, that redirects them to the desired CM.

The Gateway application configures the virtual serial COM port through which the computer communicates with the GM. It assigns values to the various parameters (baud rate, number of stop bits, number of parity bits), ensuring that they are equal to the ones defined in the UART of the GM.

After this configuration process, the PC is able to communicate with the Gateway Module. This communication process is handled by two concurrent threads that share the communication port. One thread handles the reception of data and assembles the replies to the received requests, and the other is responsible for the assembly and transmission of the data frame defined by the user. The access to the communication port is regulated by a mutex.
Chapter 5

Results

This chapter describes the prototype developed to illustrate a DomoBus Control Network (DCN) and details the tests performed to the various functionalities of the software developed for the Control Modules (CM) and Gateway Modules (GM).

The developed prototype is described in Section 5.1. Section 5.2 presents the details regarding the configuration of the Control Modules through the analysis of the configuration files used in this process. Next, in Section 5.3 are presented how were developed the applications implemented in the Control Modules. In sections 5.4, 5.5, 5.6, 5.7 and 5.8 are exposed the results obtained from the various tests that were performed on the prototype. Finally, in Section 5.9 are described the problems found during the performance of these tests.

5.1 Description of the Prototype

This prototype includes a PC as a Supervision Module, which runs the developed Gateway application, and 3 Arduino UNO boards, as illustrated in Figure 5.1. One of the Arduino boards was used as a Gateway Module and the two remaining boards were used as Control Modules. Each one of the Arduino boards is connected to a nRF24L01+ radio module.

Figure 5.1: Prototype’s modules and devices

Both CMs execute the same SYS application that controls a blinking LED based on the value of
a property called "pulses". The number of blinks of the LED is equal to the value of the mentioned property. There was no need to implement more functionalities in the SYS application as the ones that were implemented already illustrate its operation principles.

The first CM, designated "CM 1", controls a green LED (LED_0) and a red LED (LED_1). Both LEDs are characterized by a single property designated "state" that controls the state of the LED. When one assigns a value greater or equal to 1 to this property, the respective LED will be switched on. Otherwise, when the property value is equal to zero, the respective LED will be turned off.

Regarding the other CM, designated "CM 2", this module controls a single LDR connected to the module's analog I/O port. This device is characterized by two properties. The first one, called
"lux", corresponds directly to the value of light intensity measured by the LDR. The other property, called "darkness", is used to characterize measured luminosity. The value of this property is equal to 1 in a dark environment and equal to zero on an illuminated environment. Naturally, the value of the "darkness" property is directly dependent on the value of the "lux" property and of what the programmer defines to be an illuminated or a dark environment.

In Figure 5.2 are represented two schemes that illustrate the connections between the Control Modules and the devices controlled by each one of them.

5.2 Configuration Process

Taking into account the devices and properties to be controlled by each one of the Control Modules, two configuration files were generated, as illustrated in Figure 5.3. This files follow the structure described in Section 4.2.4. In Appendixes B and C one has the generated header files with the configurations for CM 1 and CM 2, respectively.

59
5.3 Implementing the Applications

One of the main objectives of this work is to simplify the development of applications for Control Modules. As such, it is important to demonstrate how the applications implemented in this prototype were developed. Two applications will be analysed: the LAMP application implemented in CM 1 and the LIGHT application of CM 2.

5.3.1 CM 1 - LAMP Application

The first step of the application development was the definition of the digital I/O port (PORTD) and the pins to which the devices are connected: LED_0 is connected to pin 6 and LED_1 to pin 5. One had also to define the respective Data Direction Register (DDR), that in this case is DDRD, where one defines whether a certain pin is used as an input or as an output.

```c
#define LAMP_PORT PORTD
#define LAMP_DDR DDRD
#define LAMP_0_PIN_MASK 0x20 /* PIN5 : 0010 0000 */
#define LAMP_1_PIN_MASK 0x40 /* PIN6 : 0100 0000 */
```

Next, the LAMP_process_state() function was implemented. In this function one defines the actions to perform in the devices taking into account the respective property values. In this prototype both LEDs are “active-low”, which means they are switched on when their property value is equal to zero and are switched off otherwise.

```c
void LAMP_process_state(uint8_t device){
    if(LAMP_state[device]){ // LED on
        LAMP_PORT &= ~(LAMP_0_PIN_MASK << device);
    } else { // LED off
        LAMP_PORT |= LAMP_0_PIN_MASK << device;
    }
}
```

In the LAMP_init() function, where all the initialization tasks are performed, pins 5 and 6 were defined as output signals and the logical value “1” was assigned to them, switching off the LEDs.

```c
void LAMP_init(void){
    LAMP_DDR |= (LAMP_0_PIN_MASK | LAMP_1_PIN_MASK);
    LAMP_PORT |= (LAMP_0_PIN_MASK | LAMP_1_PIN_MASK);
}
```

Finally, in the LAMP_task() function, one only needs to call the LAMP_get_new() function to check if the values of the “state” property of the LEDs were modified by SET messages. In case they
were, this function will call LAMP_process_state() function to apply the respective change in the target device. The LAMP_send_state() function will be called immediately after to send a NOTIFY message to the module that sent the SET message, in order to confirm the reception and treatment of the new property value.

```c
void LAMP_task(void){
    LAMP_get_new();
}
```

Appendix D presents the code file of the LAMP application. In this appendix, the text displayed in bold font corresponds to the code lines inserted to program the LEDs. The remaining code lines and comment lines were generated automatically by the configuration application.

### 5.3.2 CM 2 - LIGHT Application

The LIGHT application is more complex than the application analysed in the previous section, given the fact one controls two properties with it. As such, in this section will be analysed a simplified version of this application, where one only controls the "lux" property, correspondent to the luminosity read by the LDR. This way, one has a simple example of how to develop applications for sensors.

The first step of the application development was the definition of a constant related the analog I/O pin connected to the LDR (pin 0). It were also defined two global variables, one used to measure the time elapsed between readings and the other one to store the luminosity value read from the sensor. The timer variable is incremented every 100 ms.

```c
#define INPUT_CHANNEL 0X00
uint8_t LIGHT_timer = 0;
static uint16_t read_lux = 0;
```

In the LIGHT_init() function, the ADC and the global variables are initialized.

```c
void LIGHT_init(void){
    ADMUX = (1<<REFS0);
    ADCSRA = ((1<<ADEN) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0));
    LIGHT_timer = 0;
    read_lux = 0;
}
```

Finally, in the LIGHT_task() function, one obtains the luminosity value read by the sensor. For test purposes, it was decided these readings occur with a time interval of 1 second between them. If the read value is different from the property value, the function LIGHT_set_and_send_lux() is called to update the property value and to send the respective NOTIFY message.
void LIGHT_task(void) {
    if(LIGHT_timer >= 10) {
        LIGHT_timer = 0;
        analogRead(INPUT_CHANNEL);
        if(read_lux != LIGHT_lux[0]) {
            LIGHT_set_and_send_lux(device, read_lux);
        }
    }
}

5.4 Test 1 - GET, SET and NOTIFY

This test was performed on the “state” property of CM 1’s LED_0, controlled by the LAMP application. At the beginning of the test, the LED was switched off, which means the property value was equal to zero. Next, the following messages were sent from the PC Gateway application:

1. a GET message was sent to obtain the value of the target property. As expected, the CM sent back an Answer that contained the requested property value, which was equal to zero;

2. next, a SET message was sent to turn on the LED, by setting a value greater or equal to 1 in the data field. The CM sent back an ACK message that confirmed the reception of the SET message, followed by a NOTIFY message (that is used to inform the Supervision Module that the property value has changed). LED_0 was successfully turned on;

3. GET - one last GET message was sent to confirm the new property value.

This same process was repeated for LED_1, which means the two LEDs were turned on by the end of the test, as represented in figure 5.4.

Figure 5.4: Test 1 - by the end of the test, both LEDs were turned on
In this test it was possible to verify that one may interact successfully with values of properties controlled by the CM using GET, SET and NOTIFY messages. This test has also showed that one may control multiple devices with a single CM.

5.5 Test 2 - Message Retransmission

This test demonstrates the well-functioning of the messages' retransmission process. To perform this test, the PC Gateway application was configured to stop sending the Answer/ACK messages to the received messages.

This test used the properties of the LDR controlled by CM 2. By performing some alterations in the lighting conditions of the room where the LDR was placed, the values of both properties changed. These alterations caused the transmission of the respective NOTIFY messages. These messages were retransmitted 7 times each 1.25 seconds given that the PC didn't send the respective replies. The value of the number of retransmissions and of the timeouts were chosen only for test purposes.

In conclusion, Test 2 showed the well-functioning of the message retransmission functionalities, such as the TX-events and their respective lists, along with the retransmission timeouts. It has also shown that the correct operation of 16_BIT properties.

5.6 Test 3 - Error Messages

This test was performed to verify whether the CMs were able to detect the errors in the received messages, such as invalid application numbers or property indexes, for example.
Message received with an error code equal to 2

Message received with an error code equal to 4

Figure 5.5: Test 3 - received error messages

In Figure 5.5 are represented the messages received when a message is sent with a wrong device number in the packet’s CDevDest field (error 2) and also the message received when one tries to interact with a non-existent 8-BIT property (error 4).

Similar tests were performed for the remaining error types. The respective error messages were successfully received.

This test demonstrated that the Control Modules can detect errors in the received messages, sending back a reply with the correct error code, ensuring a robust interaction.

5.7 Test 4 - Array Manipulation

None of the devices used in these tests had D_ARRAY properties. However, it is important to show that all the functionalities implemented in Control Modules can also be used for the control of this kind of properties. As such, it was implemented an additional application in CM 2, called TST,
that controls three D ARRAY properties (two with a maximum dimension of 4 bytes and one with a maximum dimension of 8 bytes). Although this application and its property weren’t related to any device, it was decided that the byte dimension of these properties would be assigned to the “pulses” property of the SYS application. This way, one could visualize an alteration in the number of blinks of the LED controlled by the SYS application as a consequence of a change in the value of the D ARRAY property controlled by the TST application.

Various SET messages were sent to modify the values of the mentioned properties. As expected, the respective ACK messages were received along with the NOTIFY messages which certified that the new property values were processed by the application. The expected change in the number of blinks of the LED controlled by the SYS application could also be verified. Finally, a GET message was sent to confirm that the new property values were set.

With this test, it was possible to confirm that the software developed for the Control Modules deals correctly with D ARRAY properties.

5.8 LED Control with a LDR

Taking into account the devices controlled by CM1 and CM2, it was implemented a small automatism in the PC Gateway application. With this automatism, one may control LED 0’s “state” property from the “darkness” property that characterizes the used LDR. More precisely, if the “darkness” property of the LDR is equal to 1, LED 0’s “state” property value will be set to 1, turning the LED on. On the contrary, the LED will be switched off. This process is illustrated below in Figure 5.6 in which are depicted two screen-shots of a small video in which the implemented automatism was recorded.
In this test was demonstrated in a simple manner that the properties of the various devices controlled by Control Modules may be used in the development of automatisms in the Supervision Level.

5.9 Implementation Problems

In a general way, one may perform a positive evaluation of the obtained results. The various tests performed showed the well-functioning of the implemented functionalities.

However, it was possible to notice that the used Gateway Module presents some problems regarding the communication with the PC, especially when one wishes to communicate at a high transmission rate (500 kbps - 1 Mbps). Consequently, sometimes it wasn’t possible to send or receive messages to/from the Control Modules. After several attempts, it wasn’t possible to either isolate or correct this error. As a safety measure, and given some time restraints, it was decided to use a 9600 bps baud rate in the communication between the PC and the GM. At this baud rate, the GM presents a more stable behaviour. However, some occasional errors still occur.
Chapter 6

Conclusions

6.1 Work Performed and Results Achieved

This thesis presented the work performed regarding the development of Control Modules (CM) and Gateway Modules (GM) for the DomoBus home automation system.

First of all, it was performed a study regarding the development of domotic modules in other systems, namely KNX and LonWorks, two of the main existent technologies on the market. It was possible to verify that, in these systems, domotic modules are expensive devices that use specific hardware associated with the technology, making them incompatible with each other and not offering modular solutions easily modifiable or adaptable to different devices. There is also very little information available regarding the hardware and software structure of these modules, especially in case of the KNX system.

Then, the DomoBus system, which is being developed in an academic context, was introduced. In the DomoBus approach, domotic devices (sensors and actuators) are treated as generic entities characterized by a set of properties. One may interact with the property values through a set of control messages: GET (obtain property value), SET (modify property value) and NOTIFY (report a change in a property value). With this simple model, it is possible to treat home devices with the most diverse characteristics according to the same principles.

The main objective of this work was the development of the Control Modules’ software. These components are directly responsible for the control of home devices according to the values of their properties. The CMs are able to control multiple devices of different types simultaneously, which is an especially useful feature for houses with a high number of devices to control.

Hardware-wise, the Control Modules are based on low-cost generic boards, similar to Arduino boards. To keep the price as low as possible it was decided to use boards based on 8-bit micro-processors, which have some hardware limitations. To facilitate the implementation of a prototype, Arduino UNO boards, the simplest of the Arduino product line, were chosen.

The developed software is based on a simple and modular software architecture, composed of three main components: NET, MSG (MESSAGE) and PROP (PROPERTIES). These components, along with the applications responsible for the control of domotic devices, are executed sequentially in a non-preemptive operative nucleus in a "round-robin" fashion. This simple solution was chosen for being well-supported by the used hardware, for allowing the development of the software modules independently and for facilitating the conjunction of the different software modules.

The NET component is responsible for the transmission and reception of packets, allowing each
CM to interact with other CM and with the applications in the Supervision Level. This component can fully implement the required communication protocol or it may just interface with specific communication modules, such as Wireless transceivers. In this particular implementation were used nRF24L01+ radio modules to support the communication between the modules of a DomoBus Control Network (DCN). These modules were used taking into account their low-cost, small size, low energy consumption and for their fully implemented communication protocol.

The MSG software component processes DomoBus messages received through the NET component and mounts the messages to be sent to other nodes of a DCN. This component includes a fully implemented message retransmission mechanism. The received messages operate over the properties of the domotic devices, managed in the PROP component.

The PROP component is composed of a set of data structures used to store and manipulate the property values, accessed by the MSG component and by the control applications.

One of the main objectives of this work was to facilitate the development of applications for the Control Modules. As such, it was created an auxiliary program for the configuration of the CMs, based on a set input data inserted by the user, such as the application names, number of devices, number of properties of each type (8_BIT, 16_BIT and D_ARRAY), and the properties’ names. This program generates a header file where the PROP component’s data structures are declared. In addition, this configuration program also creates a set of code files which include the basic skeleton of the applications to develop. It is also offered an API through which programmers may access and modify property values using a set of macros that use the application and property names inserted in the configuration process. With the defined architecture and the developed software, programmers don’t have to be concerned about the module’s communication details. They just have to deal with the control of the home devices based on their properties’ values, which are accessed and/or updated through the provided API.

It also was developed the software of the Gateway Modules of a DCN, that interface the Supervision Modules (SM) of the Supervision Level, the Control Modules of a DCN. The developed software is based on a modular software architecture composed of two components: NET, where occurs the exchange of packets with other nodes of a DCN, and SM_COMM (Supervision Modules Communication), responsible for the communication between the GM and the Supervision Modules.

Finally, it was also developed a prototype to test the functionality of the developed modules. This prototype included a Gateway Module and two Control Modules, that controlled their own set of devices. The results obtained of the various tests validated the proper functioning of the implemented functionalities and the software architecture.

6.2 Future Developments

There is a set of functionalities to be developed in the future to increase the degree of functionality of the system and its components.

Regarding the Control Modules, it wasn’t possible to implement the exchange of messages directly
between CMs in a DCN. This process would be based on messages whose basic structure would be stored in the module’s EEPROM. Unfortunately, the bootloader installed in the module’s MCU doesn’t support the storage of data into the EEPROM. As such, it must be developed an alternative process for the insertion of this data into the EEPROM. Additionally, it would be rewarding to use other devices as communication modules, instead of using only nRF24L01+ radio modules, to demonstrate the modularity of the developed software architecture and, in particular, of the NET module.

Concerning the CM’s configuration software, it was verified that the option that was taken regarding the classification of the controlled devices in pure “sensors” and “actuators” may bring some restrictions. Sensors may contain controllable properties, like their activation state, and actuators may be characterized by properties that shouldn’t be modified by SET messages. In this context, the improvements to implement have to avoid to a maximum extent the introduction of complexities in the fulfilment of the configuration file.

Regarding the Gateway Modules, efforts should be deployed to correct the problems found in the USB connection between the PC and the Gateway Module, ensuring a reliable exchange of data between these two components.

Finally, and regarding the system itself, it would be rewarding to implement modifications in the communication protocol to support the control of various properties with one single message. This alteration would increase the efficiency of the communication process.
Bibliography


## Appendix A

### List of Communication Errors

<table>
<thead>
<tr>
<th>Error Opcode</th>
<th>Designation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ERROR_NAPP</td>
<td>The received message was destined to a non-existent application</td>
</tr>
<tr>
<td>2</td>
<td>ERROR_ADEV</td>
<td>The received message was destined to a non-existent device controlled by the destination application</td>
</tr>
<tr>
<td>3</td>
<td>ERROR_OPCODE</td>
<td>The received message contained an invalid opcode</td>
</tr>
<tr>
<td>4</td>
<td>ERROR_ID_8_BIT</td>
<td>The received message was destined to a non-existent 8-BIT property of the target device</td>
</tr>
<tr>
<td>5</td>
<td>ERROR_ID_16_BIT</td>
<td>The received message was destined to a non-existent 16-BIT property of the target device</td>
</tr>
<tr>
<td>6</td>
<td>ERROR_ID_DARRAY</td>
<td>The received message was destined to a non-existent D_ARRAY property of the target device</td>
</tr>
<tr>
<td>7</td>
<td>ERROR_DATATYPE</td>
<td>The target property's type is invalid</td>
</tr>
<tr>
<td>8</td>
<td>ERROR_ACK_NTFY</td>
<td>The CM received a ACK message with a NOTIFY opcode</td>
</tr>
<tr>
<td>9</td>
<td>ERROR_ARRAY_DIMENSION</td>
<td>The received D_ARRAY dimension exceeded the established maximum length</td>
</tr>
<tr>
<td>10</td>
<td>ERROR_SENSOR_SET</td>
<td>The CM received a SET message for the control of a sensor property</td>
</tr>
</tbody>
</table>

*Table A.1: List of Communication Errors*
Appendix B

Node 1 Configuration File

/* DEFINES */

#define NODE_MAX_APPS 2
#define SYS_APP_NUM 0
#define SYS_DEVICE_TYPE 1
#define SYS_NUM_DEVICES 1
#define SYS_NUM_PROP8 1
#define SYS_NUM_PROP16 0
#define SYS_NUM_PROP_A 0
#define LAMP_APP_NUM 1
#define LAMP_DEVICE_TYPE 1
#define LAMP_NUM_DEVICES 2
#define LAMP_NUM_PROP8 1
#define LAMP_NUM_PROP16 0
#define LAMP_NUM_PROP_A 0
#define CM_TOTAL_NUM_DEVICES 3
#define CM_TOTAL_NUM_PROP8 3
#define CM_TOTAL_NUM_PROP16 0
#define CM_TOTAL_NUM_PROP_A 0
#define CM_TOTAL_LEN_PROP_A 0
#define SYS_PROP8_0 0
#define LAMP_PROP8_0 1
#define CM_LIST_EMPTY 255

/* DATA STRUCTURES AND VARIABLES */
#if defined(_NODE_MSG_)

// application devices array
uint8_t CM_num_devices[NODE_MAX_APPS] = {SYS_NUM_DEVICES, LAMP_NUM_DEVICES};

// application device type
uint8_t CM_device_type[NODE_MAX_APPS] = {SYS_DEVICE_TYPE, LAMP_DEVICE_TYPE};

// property arrays
uint8_t CM_prop8[CM_TOTAL_NUM_PROP8];
uint8_t CM_num_prop8[NODE_MAX_APPS] = {SYS_NUM_PROP8, LAMP_NUM_PROP8};
uint8_t CM_prop8_offset[NODE_MAX_APPS] = {SYS_PROP8_0, LAMP_PROP8_0};
uint8_t CM_ctr8[CM_TOTAL_NUM_PROP8];
uint8_t CM_seqn8[CM_TOTAL_NUM_PROP8];

uint16_t *CM_prop16 = NULL;
uint8_t *CM_num_prop_16 = NULL;
uint8_t *CM_prop_16_offset = NULL;
uint8_t *CM_ctr16 = NULL;
uint8_t *CM_seqn16 = NULL;

uint8_t *CM_prop_a = NULL;
uint8_t *CM_num_prop_a = NULL;
uint8_t *CM_prop_a_idx = NULL;
uint8_t *CM_prop_a_offset = NULL;
uint8_t *CM_prop_a_dim = NULL;
uint8_t *CM_ctr_a = NULL;
uint8_t *CM_seqn_a = NULL;

// LAE
uint8_t CM_lidx[CM_TOTAL_NUM_PROP8 + CM_TOTAL_NUM_PROP16 + CM_TOTAL_NUM_PROP_A];
uint8_t CM_tout[CM_TOTAL_NUM_PROP8 + CM_TOTAL_NUM_PROP16 + CM_TOTAL_NUM_PROP_A];

// heads of lists
uint8_t CM_send_now_head = CM_LIST_EMPTY;
uint8_t CM_retrans_head = CM_LIST_EMPTY;
uint8_t CM_rx_head[NODE_MAX_APPS] = {CM_LIST_EMPTY, CM_LIST_EMPTY};
#define SYS_rx_head CM_rx_head[SYS_APP_NUM]
#define LAMP_rx_head CM_rx_head[LAMP_APP_NUM]

// tails of lists
uint8_t CM_send_now_tail = CM_LIST_EMPTY;
uint8_t CM_retrans_tail = CM_LIST_EMPTY;
uint8_t CM_rx_tail[NODE_MAX_APPS] = {CM_LIST_EMPTY, CM_LIST_EMPTY};
#define SYS_rx_tail CM_rx_tail[SYS_APP_NUM]
#define LAMP_rx_tail CM_rx_tail[LAMP_APP_NUM]

// property name variables
uint8_t *SYS_pulses = CM_prop8 + SYS_PROP8_0;
uint8_t *LAMP_state = CM_prop8 + LAMP_PROP8_0;

#else

// application devices array
extern uint8_t CM_num_devices[NODE_MAX_APPS];

// application device type
extern uint8_t CM_device_type[NODE_MAX_APPS];

extern uint8_t CM_prop8[CM_TOTAL_NUM_PROP8];
extern uint8_t CM_num_prop_8[CM_TOTAL_NUM_PROP8];
extern uint8_t CM_prop_8_offset[CM_TOTAL_NUM_PROP8];
extern uint8_t CM_ctr8[CM_TOTAL_NUM_PROP8];
extern uint8_t CM_seqn8[CM_TOTAL_NUM_PROP8];

extern uint16_t *CM_prop16;
extern uint8_t *CM_num_prop_16;
extern uint8_t *CM_prop_16_offset;
extern uint8_t *CM_ctr16;
extern uint8_t *CM_seqn16;

extern uint8_t *CM_prop_a;
extern uint8_t *CM_num_prop_a;
extern uint8_t *CM_prop_a_offset;
extern uint8_t *CM_prop_a_idx;
extern uint8_t *CM_prop_a_dim;
extern uint8_t *CM_ctr_a;
extern uint8_t *CM_seqn_a;

// LAE
extern uint8_t CM_lidx[CM_TOTAL_NUM_PROP8 + CM_TOTAL_NUM_PROP16 + CM_TOTAL_NUM_PROP_A];
extern uint8_t CM_tout[CM_TOTAL_NUM_PROP8 + CM_TOTAL_NUM_PROP16 + CM_TOTAL_NUM_PROP_A];

// heads of lists
extern uint8_t CM_send_now_head;
extern uint8_t CM_retrans_head;
extern uint8_t CM_rx_head[NODE_MAX_APPS];
define SYS_rx_head CM_rx_head[SYS_APP_NUM]
define LAMP_rx_head CM_rx_head[LAMP_APP_NUM]

// tails of lists
extern uint8_t CM_send_now_tail;
extern uint8_t CM_retrans_tail;
extern uint8_t CM_rx_tail[NODE_MAX_APPS];
define SYS_rx_tail CM_rx_tail[SYS_APP_NUM]
define LAMP_rx_tail CM_rx_tail[LAMP_APP_NUM]

// property name variables
extern uint8_t *SYS_pulses;
extern uint8_t *LAMP_state;

#endif

// MACROS
#define SYS_send_pulses(dev) CM_send_prop8(dev+SYS_PROP8_0);
#define SYS_set_and_send_pulses(dev, value) CM_set_and_send_prop8(dev+SYS_PROP8_0, value);

#define LAMP_send_state(dev) CM_send_prop8(dev+LAMP_PROP8_0);
#define LAMP_set_and_send_state(dev, value) CM_set_and_send_prop8(dev+LAMP_PROP8_0, value);

B-3
Appendix C

Node 2 Configuration File

/* DEFINES */
#define NODE_MAX_APPS 2
#define SYS_APP_NUM 0
#define SYS_DEVICE_TYPE 1
#define SYS_NUM_DEVICES 1
#define SYS_NUM_PROP8 1
#define SYS_NUM_PROP16 0
#define SYS_NUM_PROP_A 0
#define LIGHT_APP_NUM 1
#define LIGHT_DEVICE_TYPE 0
#define LIGHT_NUM_DEVICES 1
#define LIGHT_NUM_PROP8 1
#define LIGHT_NUM_PROP16 1
#define LIGHT_NUM_PROP_A 0
#define CM_TOTAL_NUM_DEVICES 2
#define CM_TOTAL_NUM_PROP8 2
#define CM_TOTAL_NUM_PROP16 1
#define CM_TOTAL_NUM_PROP_A 0
#define CM_TOTAL_LEN_PROP_A 0
#define SYS_PROP8_0 0
#define LIGHT_PROP8_0 1
#define LIGHT_PROP16_0 0
#define CM_LIST_EMPTY 255

/* DATA STRUCTURES AND VARIABLES */
#if defined ( _NODE_MSG_ )

// application devices array
uint8_t CM_num_devices[NODE_MAX_APPS] = {SYS_NUM_DEVICES, LIGHT_NUM_DEVICES};

// application device type
uint8_t CM_device_type[NODE_MAX_APPS] = {SYS_DEVICE_TYPE, LIGHT_DEVICE_TYPE};

// property arrays
uint8_t CM_prop8[CM_TOTAL_NUM_PROP8];
uint8_t CM_num_prop8[NODE_MAX_APPS] = {SYS_NUM_PROP8, LIGHT_NUM_PROP8};
uint8_t CM_prop8_offset[NODE_MAX_APPS] = {SYS_PROP8_0, LIGHT_PROP8_0};
uint8_t CM_ctr8[CM_TOTAL_NUM_PROP8];
uint8_t CM_seqn8[CM_TOTAL_NUM_PROP8];

uint16_t CM_prop16[CM_TOTAL_NUM_PROP16];
uint8_t CM_num_prop16[NODE_MAX_APPS] = {SYS_NUM_PROP16, LIGHT_NUM_PROP16};
uint8_t CM_prop16_offset[NODE_MAX_APPS] = {0, LIGHT_PROP16_0};
uint8_t CM_ctr16[CM_TOTAL_NUM_PROP16];
uint8_t CM_seqn16[CM_TOTAL_NUM_PROP16];

uint8_t *CM_prop_a = NULL;
uint8_t *CM_num_prop_a = NULL;
uint8_t *CM_prop_a_idx = NULL;
uint8_t *CM_prop_a_offset = NULL;
uint8_t *CM_prop_a_dim = NULL;
uint8_t *CM_ctr_a = NULL;
uint8_t *CM_seqn_a = NULL;

// LAE
uint8_t CM_lidx[CM_TOTAL_NUM_PROP8 + CM_TOTAL_NUM_PROP16 + CM_TOTAL_NUM_PROP_A];
uint8_t CM_tout[CM_TOTAL_NUM_PROP8 + CM_TOTAL_NUM_PROP16 + CM_TOTAL_NUM_PROP_A];

// heads of lists
uint8_t CM_send_now_head = CM_LIST_EMPTY;
uint8_t CM_retrans_head = CM_LIST_EMPTY;
uint8_t CM_rx_head[NODE_MAX_APPS] = {CM_LIST_EMPTY, CM_LIST_EMPTY};
#define SYS_rx_head CM_rx_head[SYS_APP_NUM]
#define LIGHT_rx_head CM_rx_head[LIGHT_APP_NUM]

// tails of lists
uint8_t CM_send_now_tail = CM_LIST_EMPTY;
uint8_t CM_retrans_tail = CM_LIST_EMPTY;
uint8_t CM_rx_tail[NODE_MAX_APPS] = {CM_LIST_EMPTY, CM_LIST_EMPTY};
#define SYS_rx_tail CM_rx_tail[SYS_APP_NUM]
#define LIGHT_rx_tail CM_rx_tail[LIGHT_APP_NUM]

// property name variables
uint8_t *SYS_pulses = CM_prop8 + SYS_PROP8_0;

uint8_t *LIGHT_darkness = CM_prop8 + LIGHT_PROP8_0;
uint16_t *LIGHT_lux = CM_prop16 + LIGHT_PROP16_0;

#else
// application devices array
extern uint8_t CM_num_devices[NODE_MAX_APPS];

// application device type
extern uint8_t CM_device_type[NODE_MAX_APPS];

extern uint8_t CM_prop8[CM_TOTAL_NUM_PROP8];
extern uint8_t CM_num_prop8[NODE_MAX_APPS];
extern uint8_t CM_prop8_offset[CM_TOTAL_NUM_PROP8];
extern uint8_t CM_ctr8[CM_TOTAL_NUM_PROP8];
extern uint8_t CM_seqn8[CM_TOTAL_NUM_PROP8];

extern uint16_t CM_prop16[CM_TOTAL_NUM_PROP16];
extern uint8_t CM_num_prop_16[NODE_MAX_APPS];
extern uint8_t CM_prop_16_offset[CM_TOTAL_NUM_PROP16];
extern uint8_t CM_ctr16[CM_TOTAL_NUM_PROP16];
extern uint8_t CM_seqn16[CM_TOTAL_NUM_PROP16];

extern uint8_t *CM_prop_a;
extern uint8_t *CM_num_prop_a;
extern uint8_t *CM_prop_a_offset;
extern uint8_t *CM_prop_a_idx;
extern uint8_t *CM_prop_a_dim;
extern uint8_t *CM_ctr_a;
extern uint8_t *CM_seqn_a;

// LAE
extern uint8_t CM_lidx[CM_TOTAL_NUM_PROP8 + CM_TOTAL_NUM_PROP16 +
CM_TOTAL_NUMPROP_A];
extern uint8_t CM_tout[CM_TOTAL_NUM_PROP8 + CM_TOTAL_NUM_PROP16 +
CM_TOTAL_NUMPROP_A];

//heads of lists
extern uint8_t CM_send_now_head;
extern uint8_t CM_retrans_head;
extern uint8_t CM_rx_head[NODE_MAX_APPS];
#define SYS_rx_head CM_rx_head[SYS_APP_NUM]
#define LIGHT_rx_head CM_rx_head[LIGHT_APP_NUM]

//tails of lists
extern uint8_t CM_send_now_tail;
extern uint8_t CM_retrans_tail;
extern uint8_t CM_rx_tail[NODE_MAX_APPS];
#define SYS_rx_tail CM_rx_tail[SYS_APP_NUM]
#define LIGHT_rx_tail CM_rx_tail[LIGHT_APP_NUM]

//property name variables
extern uint8_t *SYS_pulses;
extern uint16_t *LIGHT_lux;

#endif

/*MACROS*/
#define SYS_send_pulses(dev) CM_send_prop8(dev+SYS_PROP8_0);
#define SYS_set_and_send_pulses(dev, value) CM_set_and_send_prop8(dev+
SYS_PROP8_0, value);

#define LIGHT_send_darkness(dev) CM_send_prop8(dev+LIGHT_PROP8_0);
#define LIGHT_set_and_send_darkness(dev, value) CM_set_and_send_prop8(dev+
LIGHT_PROP8_0, value);
#define LIGHT_send_lux(dev) CM_send_prop16(dev+LIGHT_PROP16_0);
#define LIGHT_set_and_send_lux(dev, value) CM_set_and_send_prop16(dev+
LIGHT_PROP16_0, value);
Appendix D

LAMP Application Code File

```c
#include <stdint.h>
#include <avr/io.h>

#include "node.h"
#include "LAMP.h"
#include "LAMP_msg.h"

/* NUMBER OF DEVICES */
//LAMP_NUM_DEVICES 2

/* PROPERTY VARIABLES */
//LAMP_state[<device_number>];

#define LAMP_PORT PORTD
#define LAMP_DDR DDRD
#define LAMP_0_PIN_MASK 0x20 /* PIN5 : 0010 0000 */
#define LAMP_1_PIN_MASK 0x40 /* PIN6 : 0100 0000 */

/*****************************************/
LAMP_process_state ( )
*
* Arguments:
*   - device: number of the device to control
* * Return value: void
* * Effect: function for the treatment of the property <state>
****************************************/
void LAMP_process_state ( uint8_t device ){

    if( LAMP_state[device] ){
        LAMP_PORT &= ~(LAMP_0_PIN_MASK<<device);
    } else{
        LAMP_PORT |= LAMP_0_PIN_MASK<<device;
    }
}
```

/*****************************************************************************/
/*
 * LAMP_init ( )
 * Arguments: void
 * Return value: void
 * Effect: function where the initialization tasks are performed
******************************************************************************/
void LAMP_init ( void )
{
    LAMP_DDR |= (LAMP_0_PIN_MASK | LAMP_1_PIN_MASK);
    LAMP_PORT |= (LAMP_0_PIN_MASK | LAMP_1_PIN_MASK);
    return ;
}

*****************************************************************************/
/*
 * LAMP_task ( )
 * Arguments: void
 * Return value: void
 * Effect: main execution cycle of the application
******************************************************************************/
void LAMP_task ( void )
{
    // DO NOT REMOVE THIS LINE – This function calls the correct process() function
    LAMP_get_new ( ) ;
    return ;
}