TÉCNICO LISBOA

An Arduino-Based Domotic System for the Internet-of-Things

Paulo Ivanilson Cabral Borges

Thesis to obtain the Master of Science Degree in

Telecommunications and Informatics Engineering

Supervisor: Prof. Doutor Paulo Jorge Fernandes Carreira

Examination Committee

Chairperson:	Prof. Doutor Paulo Jorge Pires Ferreira
Supervisor:	Prof. Doutor Paulo Jorge Fernandes Carreira
Member of the Committee:	Prof. Doutor Renato Jorges Caleira Nunes

November 2015

Resumo

Apesar dos grandes avanços na área de Automação de Casas e Edifícios, a adesão aos sistemas de automação, para aplicações de tamanho reduzido como apartamentos e pequenos escritórios, ainda não é notável. Os fabricantes continuam a desenvolver sistemas de automação focados em aplicações de grande porte e, os sistemas disponíveis aplicáveis à pequenas instalações ainda não correspondem às expectativas dos utilizadores, devido à baixa funcionalidade, preço, complexidade e desempenho. Numa tentativa para abordar as questões acima mencionadas, este trabalho propõe o desenho, o desenvolvimento e a avaliação de um sistema de automação modular simples baseado em electrónica aberta e componentes de baixo preço, destinado a pequenas aplicações. Sistemas como este contribuem para uma maior adesão e uma maior visibilidade para a área de Automação de Casas, e este sistema servirá de base para a criação e o desenvolvimento de novos sistemas de automação com menores custos e maior nível de funcionalidades. Foi implementado um protótipo usando electrónica aberta, e uma API está disponível para facilitar o desenvolvimento de aplicações para este sistema. Implementou-se também um software que demonstra a capacidade do sistema em termos de endereçamento, descoberta de módulos e funcionalidades.

Palavras-chave: Automação, Automação de Casas, Internet-of-Things, Electrónica Aberta

Abstract

Despite major advances in the Home and Building Automation field, the adoption of automation systems in small applications such as apartments and small offices is not yet visible. Manufacturers keep developing Building Automation Systems focused on large applications and the ones available for small applications are still not corresponding to user expectations, due to low functionality, price, complexity and performance issues. In an attempt to address the above mentioned issues, this work proposes to design and evaluate a simple modular automation system based on open-source hardware and cheap electronics, and focused on small applications. Systems like this will contribute for a higher adoption and visibility to Home Automation, and this system will serve as basis for the creation and development of new automation systems with lower costs and rich in functionalities. A prototype is implemented using open-source hardware, and an API is made available to make the development of applications for this system possible. It is also implemented a software which demonstrates the capabilities of the system in terms of addressing, modules discovery and functionalities.

Keywords: Automation, Home Automation, Internet-of-Things, Open-Source

Contents

List of Tables

List of Figures

Acronyms

BA Building Automation

BAS Building Automation System

DM Device Module

ETS Engineering Tool Software

HA Home Automation

HAS Home Automation System

HBAS Home and Building Automation System

I2C Inter-Integrated Circuit

IFTTT If This Then That

IoT Internet-of-Things

PID Proportional, Integral and Derivative

PWM Pulse Width Modulation

SCL Serial Clock Line

SDA Serial Data Line

Chapter 1

Introduction

As computational systems get more powerful, more compact and efficient, many new areas of interest have been opened benefiting quality of live. The two fields that emerged with this evolution are Home and Building Automation. Home Automation (HA) differs from Building Automation (BA) in the fact that HA uses automation focused more on comfort while BA is more focused in economic benefits, the efficient management of a large building environment. Despite being discussed since the 1980s HA e.g., is still a field under development. Recent concepts and technologies, such as the Internet-of-Things (IoT) and the open-source hardware platforms offer new possibilities for a vast variety of systems. IoT is an integrated part of Future Internet and could be defined as a dynamic global network infrastructure with self configuring capabilities based on standard and interoperable communication protocols where physical and virtual entities have identities, physical attributes, and virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network [14]. In the acIoT, these entities are expected to become active participants in business, information and social processes where they are enabled to interact and communicate among themselves and with the environment by exchanging data and information “sensed” about the environment, while reacting autonomously to the “real/physical world” events and influencing it by running processes that trigger actions and create services with or without direct human intervention. There are many different solutions for HA. Currently, many big manufacturers offer complete, functional and reliable automation systems but usually these are very expensive and very difficult to install. Thus, these solutions may not be suitable for small applications, like a small office or bedroom.

Open-source hardware consists in platforms that allow easy and fast prototyping and systems development. The term ”open” that information related with Hardware design in addition to the software that drives the hardware are easily available. Many projects based on this platforms can be easily found.

There are solutions based on the IoT, where each device is capable of connecting to some network, making it possible to create an automation system based on the connected devices. These solutions are suitable for small ad hoc applications specially because the bulk of complexity is factored out to the cloud.

There are also solutions based on open-source hardware, which although being cheaper then the

ones stated above, have limited functionality.

The problems of high complexity, expensiveness and limited functionality are the reason why the existing systems are yet not widely used and why they are not suitable for Home Applications. The optimal solution, would be a system that combines, at least, the following qualities:

- Low cost and reliable
- Simple and easy installation
- Rich in features and highly customizable, in order to fit every kind of environment.

This thesis proposes to investigate the feasibility of developing a system that is based on the concepts of IoT, Open Hardware and HA, that aims at overcoming the main limitations of existing systems. A system of this kind opens a new path in the development of automation systems, more accessible to the public that may be purchased, installed and used by the majority. The main limitation to the development of such system is the complexity of concepts an knowledge regarding the field of Home Automation System (HAS). At the end our proposed system results in a prototype that in conjunction with the concepts that cross the fields of HAS and IoT and benefiting from the costs of open-source hardware, surpasses main limitations of existing systems. This system is also evaluated in terms of reliability and flexibility in order to understand how will it perform in real world scenarios as small offices or homes.

1.1 Problem Statement

The demand for automation systems has been growing fast in the last years mainly motivated by the energy consumption efficiency that such systems can bring, and also by the concerns for better life quality and comfort. This growing is much noticeable in the industrial and enterprise segment than in the home segment. The high costs and complexity associated with this technology are the main limiting factors for its desirability. In the other hand the more accessible alternatives intended for small applications are limited by the level of functionalities that its provided. These are the limitations that by affecting the desirability for acHAS will be the main concern in this work.

1.2 Methodology and Contributions

This thesis, designs and implements a prototype of a modular automation system that demonstrates the feasibility of our ideas. This prototype is tested for reliability and usability in controlled environment. It is intended to open a new path of possibilities in the design of low cost HAS.

1.3 Document Organization

This document starts by describing the concepts of HA and BA (Section 2.1) and then introduces concepts and functionalities that are part of these automation systems (Section 2.2). Concepts like devices, fieldbus, etc. are introduced. Section 3 presents a study of related works, i.e., systems and implementations that are related with this work or that try solve the same problems as this works. At the end the proposed solution to solve the problems stated in Section 1.1 is presented (Section 4.1) followed by its validation method in Section 5.

Chapter 2

Concepts

The field of automation is actually a very studied and well documented subject due to the growing interest and research that exists. Therefore, it is important to understand some specific concepts that are associated with the structure and architecture of these systems.

2.1 Home and Building Automation

Building Automation System (BAS) are distributed systems capable of performing measurement, control and management in a building environment. Home and Building Automation Systems architecture typically follow a layered architecture with three hierarchical levels [16]. The bottom level is known as the **Field Level**, this level consists of Sensors and Actuators (field devices) which are responsible for the interaction with the physical environment. Data is collected by Sensors and transformed into a representation suitable for transmission. In the opposite direction, parameters of the environment are physically controlled by the Actuators in response to commands received from the system. The second level, known as the **Automation Level**, it consists of Controllers that are responsible for controlling the Field Level, comprising tasks like automatic control and data aggregation, as well as establishing logical connections between the Field devices. In some cases, functionality such as scheduling also exists on this layer. Finally, the top level, is the **Management Level**, where all system information and application logic exists. This level, implements more complex behaviour such as remote access and control, as well as system alerts. One such layered architecture is depicted in Fig. 2.1. Typically, Home Automation Systems do not include the Management Level thus, for the purpose of this work the architecture top layer will be abstracted away.

Controllers are responsible for the extraction of values and parameters configuration in field devices. The purpose of Controllers is to orchestrate Field Device's interaction through some predefined logic. This logic is embedded in modules called controller modules. These modules consist of specific hardware or, sometimes, application's software present in a server. Controller modules continually update device's state depending on monitored inputs or commands sent to the controller. Controllers expose different types of logic objects that can be read or written, such as, memory registers, timers and cal-

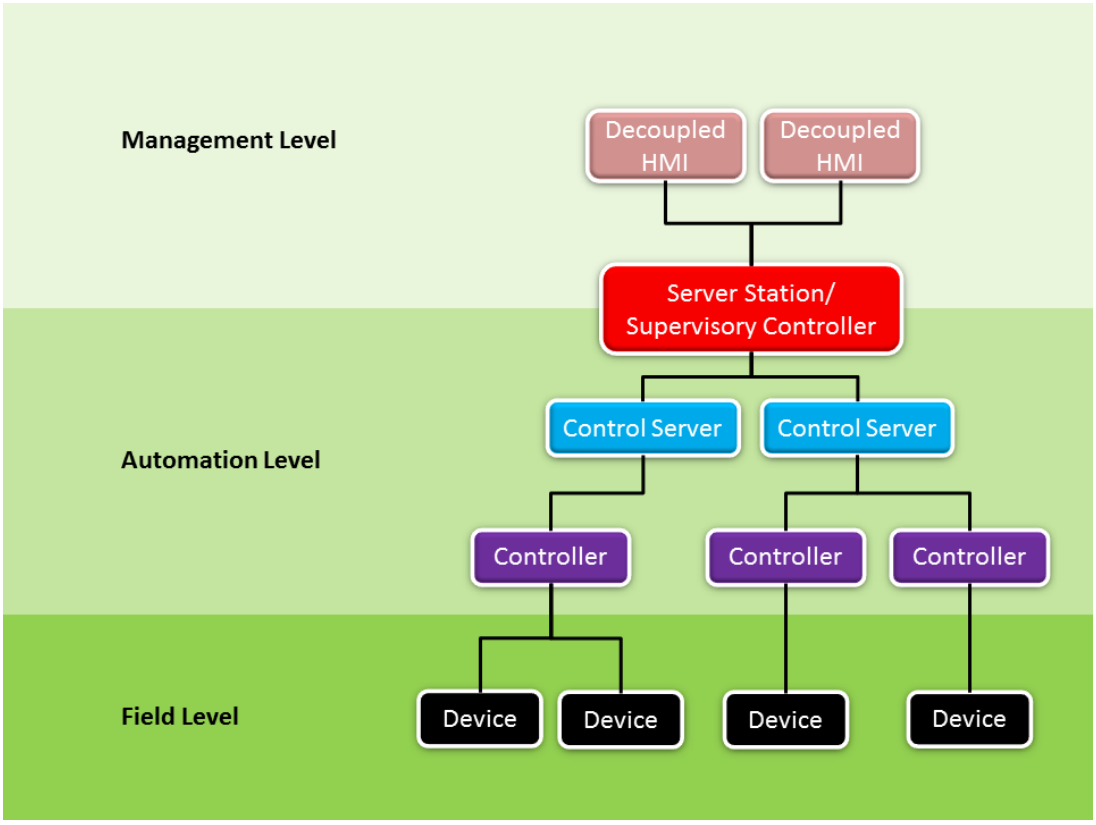


Figure 2.1: BAS Architecture

endars. Depending on the sophistication of the controller, they may be capable of running more than one control program, reading and writing on I/O ports over the fieldbus network or other logic objects. A common type of control functions used are Proportional, Integral and Derivative (PID) [1] controllers. A PID controller consists of a control loop feedback mechanism that calculates the error between some measured process value and the desired value (also known as setpoint), and then actuates in the process's control inputs in an attempt to minimize that error. A software controller module has its logic in the application's layer of a BAS in opposition to a hardware one where the logic runs in a piece of hardware at the field level layer. Software controller modules tend to centralize the system's control functions in its higher level layers. The software controller's logic can take advantage of more complex decision logic and control several devices simultaneously. Unfortunately such high-level and centralized control has disadvantages. Software controllers should not be used to continually control a process, due to the risk of flooding a network with update packets separated by very short periods of time. For example, if a process requires its controller to update its state every 2 millisecond, the network would have to handle at least 500 packets per second directed to that device, and it would get worse if there were more devices like that in the system. Hardware controller modules are best suited for real-time control, because they can take advantage of their proximity to the devices.

Fieldbus is a digital serial data bus that interconnects and enables communication between field level devices, such as sensors and actuators, and controllers. The choice of the fieldbus physical communication medium depends if whether an intrusive installation is applicable, or a high speed connection is needed, e.g. air, twisted-pair, powerline, etc.. A fieldbus system is usually associated with communi-

cation protocol that is a set of rules that devices follow in order to communicate and interoperate. The fieldbus technology promises to improve quality, reduce costs and boost efficiency by communicating with devices digitally, by reducing the required wiring and installation effort, since analog signal standards require devices to have their individual set of wires. Also, in a fielbus network, is expected that all devices connected to it have some computational power, and because of that they are considered smart devices, that can replace several analog devices at once, helping to drop even more the costs of the system [19]. There are many fieldbus systems on the market and their specifications vary according to requirements of the applications they are used in [3]. There have been created several industrial standards such as BACnet [2], KNX [17], LON [8] and CAN [20]. KNX and LON will be detailed in section **??** as these are the most compatible with the HA concept.

2.1.1 System Centralization

Control Systems can be categorized according to a property called logical or topological centralization.

A **centralized system**, as the name suggests, is a system that has a central device (one or several devices) which controls system's functions. So the system does not need very smart sensors and actuators, but is sensitive to a failure of the central devices. Today, these systems usually use a bus topology (fieldbus), but sometimes they still use direct connections to sensors and actuators.

In **distributed systems** there is no central device. This means that every device needs to have computational power in order to be intelligent enough to make complex decisions. This increases system's reliability as a device's failure does not lead to other device's failure. This increased computational power leads to higher device costs.

2.2 Automation Model Concepts

In what concerns the complex interaction and organization of the elements that are part of automation systems for this work is important to know which function and information is associated with each of these elements.

2.2.1 Devices

A physical device denotes some type of sensor, that converts a physical reality into an electric signal that can be measured, or an actuator, which is any electrically actionable device, piece of equipment or appliance, such as a a window blind or a lamp.

Automation hardware is highly heterogeneous and should be abstracted in a way that enables software applications to be as independent as possible from the specificities of the hardware. A common abstraction is to group devices into two generic groups: sensors and actuators. Some devices fit in both groups due to their sensing and actuating capabilities, but for simplicity they may be perceived as two different virtual devices: one device capable of sensing and another one capable of actuating.

So far as it concerns a BAS, devices have two interfaces, an *electrical interface*, that defines how to connect the device to the rest of the system, and an *application interface* that enables other devices, software application or even humans to interact with the device.

From a software point-of-view, reading from a sensor is equated as reading from a variable that represents a hardware input port, and commanding an actuator is equated with writing a value on a variable representing an output port. So when a device is created we have to specify what are the input and the output ports where the device will be connected.

2.2.2 Device Drivers

A device driver is a software component that controls a particular type of device attached to the system. Each device must have its own device driver, meaning that a system having to support hundreds of different devices, must install hundreds of device drivers.

The great amount of devices available poses a challenge to the creation of a generic device driver that fits every device of the same type, i.e. devices that have compatible interfaces. The interface of one device is said to be compatible with another if all the properties of one interface exist in the other and are all of the same type. For example a device driver for every lamp device where all lamps have the same properties and methods, such as on or off.

In BAS we can distinguish *hardware device drivers* and *software device drivers*. Hardware device drivers usually consist of an hardware module with I/O ports and a micro controller. Those I/O ports are connected to physical devices, which usually possess no intelligence and are directly operated using electric signals, such as a lamp, a thermistor or an air conditioner's fan. The micro controller is responsible for driving those devices and exposing an interface that can be used to address each I/O port for reading and writing purposes. This hardware module can be connected to a computer, a network or to another hardware module [5]. Software device drivers consist of software applications used to expose an interface that enables other applications to interact with devices. These device drivers are used to *(i)* convert electric signals into values stored in variables or objects that can be read by software applications, and in contrast they also *(ii)* convert values into electric signals that will be used to act on devices. The first conversion relates to situations where the applications must read the values from a sensor device, or need to query about the device's status at a certain instant. The second conversion is what enables software applications to actuate on devices, such actuations can consist of modifying device's properties or make a device actuate in its environment of influence.

In sum, physical devices are mapped into I/O ports by hardware device drivers running in a hardware modules. Which in turn can be connected to other hardware device drivers, providing an higher level of abstraction, or to a software device driver that will act as an hardware abstraction layer, serving other software applications with the capability of operating those devices, without concerning with low-level hardware related details such as device communication protocols, number of connections used by a device, baud rates etc. Figure 2.2 illustrates this type of situation, where physical devices are connected to I/O ports of a hardware module (a hardware device driver) capable of providing an interface

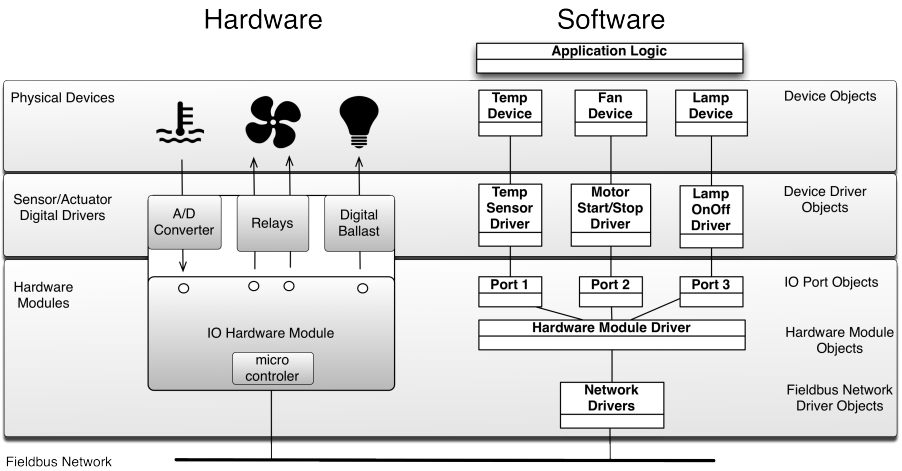


Figure 2.2: Hardware and Software Stack. Physical devices are connected to an I/O module, responsible for exposing them as addressable entities (on the left). The corresponding software representation of the hardware stack, where the I/O module is operated by a software device driver, exposing each port and each device connected to it (on the right).

that abstracts these connections. Such interface is then used to connect the hardware module (and thus connecting those physical devices) to a network. On the other hand, software applications require several abstraction layers provided by software device drivers, in order to operate those devices. The network device driver will implement the network's protocol stack, abstracting upper layers from communication specificities and exposing the hardware module's presence in the network. This is followed by the hardware module's device driver that will operate the module directly, providing a simpler interface to the upper layer, exposing the module's ports and mechanisms simplifying the tasks of reading and writing to those ports. Each port will have a device driver associated that will expose the device connected to that port and the corresponding operation mechanisms. Finally, high-level abstractions of these devices are created by defining objects that represent those devices and can be manipulated by software applications.

2.2.3 Device Groups

A device group is a logical identification of a set of devices with compatible interfaces. Interfaces are said to be compatible when they have, at least, one concept in common. For example consider two interfaces, one for controlling a lamp and other for controlling blinds. If they both have the commands ON and OFF, they are said to be compatible with each other, even though they will act differently to the same command—the ON command on lamp will lit it, but on the blinds that same command may open them.

Groups of devices gather devices that are supposed to be commanded together with a given purpose. The group will thus behave as a virtual device whose properties can be assigned. Assigning a property of a group will perform the corresponding assignment on each device that is installed in the group. A group can be ”All lamps in corridors” that are to be commanded together in some situation. Groups

are closely related with the concept of zoning (see Section 2.2.6), since zones are just logical groups, separating devices from different sections of the building.

Device groups can be defined by hardware or by software. A hardware device group is a group created by the use of a hardware device module that abstracts several devices into just one device. For example an air conditioner capable of measuring temperature and ventilate the air can be an abstraction of two independent devices, one being a thermistor and the other a fan. Another way to conceive a hardware device group is by using network logic. Several devices can belong to the same sub-network, thus forming a group. In contrast, software device groups are managed by software applications' data structures such as lists, where each list represents a device group, containing devices belonging to that group. Software device groups have a major advantage over hardware device groups, they can be dynamically arranged in order to add or remove members without requiring any effort modifying the system's structure whereas hardware device groups would require hardware modifications or network restructuring.

2.2.4 Device Commands

Commands are operations that can be executed on devices that, if successful, cause devices to change their internal state or to actuate in their environment of influence. For example, setting the intensity of a dimmable lamp is the result of a very simple command ”Dime to level”.

Certain commands may have parameters, specifying what operation should be done, and also attributes, specifying how it should be done. Dimming the lamp to a given level requires a value for the intensity level parameter to be specified. This command may also have an attribute specifying the fade time. A command definition also specifies the device interface it applies to. The command ”Open” may apply to all devices' interfaces that define a method for opening.

The same commands can be sent to groups of devices. The association of a command with a device is called an action. Actions can have labels. An action sends a command to its associated devices when executed.

2.2.5 Scheduling

BAS sometimes need to execute certain tasks at predefined schedules. BAS software usually provide a scheduling service. A scheduling service is used to relate tasks execution with some moment in time. Schedules are uniquely identified by the combination of date, time and the actions they perform. They can be defined according to fine-grained units of time such as hours, minutes or even seconds. As time passes, scheduled events are triggered and tasks are executed.

Scheduled events can be *one-time events* or *repeatable events*. A scheduling service should also enable different calendars for specific purposes. For example, a calendar that manages actions related to illumination control, in order to reduce energy consumption at night.

This service must also allow registering *user defined schedules* and *delegated schedules*. A delegated scheduled is defined by an algorithm or by an external source that captures local holiday informa-

tion or other events.

Schedules are often embedded in hardware controllers, but in larger facilities they are often configured in the main server that runs the BAS software [3].

2.2.6 Zoning

Building automation is intrinsically related to the idea of controlling spaces, which are areas that are monitored or controlled by the BAS.

Spaces are divided into several zones. A zone can be a floor, a room (or just part of it) or an area within another zone. Zones may be related according to *containment* and *adjacency*, providing information about the building (for example using zones to identify and distinguish halls from rooms in a building). Furthermore, zones are characterized by meta-data such as the zone name, space usage profile and location within the parent zone as well as its boundary polygon that defines the zone's shape and borders.

Dividing space by zones helps commissioning the BAS and eases the facility's users to navigate and recognize the controlled area in a user interface software. Zone information is also useful for querying about the spatial context of devices, this is important for determining where devices are installed and their influence zones.

Chapter 3

State of the Art

This section details common HA standards with respect to their models and characteristics, in particular, how they conceptualize their working domains, as well as other systems that are gaining attention due to their appealing features to the Home Automation Field. Low-level details regarding physical and network specificities such as devices architectures, protocol data units, properties' names, etc. will be abstracted. Instead we will focus on how the information models address fundamental BAS concepts such as grouping, notifying, scheduling, commanding, etc., as well as how installation is performed, the average cost and the complexity of each system.

3.1 Wired Systems

3.1.1 KNX

KNX, formerly known as European Installation Bus, is a building control communication system that connects sensor, actuator and controller devices. KNX system is an example of a fully decentralized complex Home and Building Automation System (HBAS). It is a very flexible system that can be used in many applications, from small houses to large enterprise buildings. KNX distributes control across several devices through a data structure called *functional block*. Each functional block is associated with one and only one device. Although a device must implement at least one functional block, it may have multiple functional blocks. A functional block consists of a group of data points and a behavioural specification about the device (like a "binary push button" functional block representing the functionality of an on/off switch)[1]. The KNX specification already defines some standard functional blocks and data points. A data point can be an input or an output data point (representing devices' inputs and outputs), or an application parameter (such as an internal variable storing the minimum light intensity a device that controls a lamp can provide). Data points also have data types associated [24]. Devices communicate by setting each other's data points (which are accessible via network). Data points can be bound (logically connected) to each other by sharing the same link identifier. When a data point value changes, the new value is propagated to all data points bounded to it. This system is able to provide interoperability if

[1] BACnet and KNX, `http://www.emate.com.vn/newsprint.aspx?id=427`

every data points' type is defined in a common standard known by all devices in the same network. The KNX specification defines three distinct ways of binding data points: *free binding*, *structured binding* and *tagged binding*. In free binding data points having the same type can be linked freely. Structured binding follows a certain pattern defined by the information model for linking data points based on devices' functional blocks. For example a push-button must have its output data points linked to the input data points of the device it will control. Finally tagged binding, also defined in the application model, proposes an address, in which part of the address contains some information in order to highlight some data points' aspects (like the group they belong), this can be used to select data points by zone, where part of the data points' address represents their location, thus tagged binding can be used to group data points or devices [17].

The KNX standard uses proprietary tools for commissioning like Engineering Tool Software (ETS)[2], that provides the user with a higher level of abstraction concerning the system's configuration complexity. KNX has some limitations like the lack of extensibility of its information model. Semantic relations like inheritance cannot be modelled using KNX. This makes KNX model not suitable for modelling with more generic environments than just device-to-device communications. Besides the great flexibility and complexity of the system, in terms of functionalities, it suffers from the same problem as most of the standardized systems, the installation and components costs are relatively high when thinking of domestic applications. Even though having a special tool for configuration, the configuration process is not trivial for a common user.

3.1.2 LonWorks

LonWorks is a networking platform developed by Echelon Corporation. Communications in this network are made using the fieldbus protocol LonTalk [9]. The LonWorks information model consists of a collection of objects such as nodes, network variables and functional profiles [11].

A LonWorks network device, having an unique address and identification, is called a *node*. Devices may implement multiple functional profiles. Functional profiles describe in detail the application layer interface of a device, i.e. its expected functionality, this includes its configuration properties, network variables and possible relations with other devices. Network variables are variables exposed by a device to other devices in the network used to exchange information. Every network variable has a data type associated that defines units, scaling and structure of the data it contains. Variables can bound to other devices' variables, but only if they share the same data type. Network variables usually follow well-defined rules defined in LonWorks specification (the Standard Network Variables Types – SNVT), guaranteeing interoperability between devices [12]. Network variables enable devices to make their own decisions based on their input variables' information, rather than forcing the devices to obey specific command requests. By requesting commands instead of sharing variables, we would be required to learn all of the existing devices' commands which would differ from device to device, forcing us to know their names, their consequences and the correct call sequence of a given list of commands. Network

[2]ETS4, `http://www.knx.org/knx-tools/ets4/description/`

variables abstract devices from other devices' internal details making it easy for manufacturers to design new interoperable devices as long as they implement functional profiles [10].

An example of functional profiles in action can be a device acting like a switch, implementing the Switch functional profile [7] that exposes an output network variable used to turn on or off other devices, for example, a lamp device. Then, that lamp device implements a Lamp Actuator functional profile [6], exposing an input network variable used to receive input values controlling the lamp's status (on or off). By linking these variables, the switch device will be able to command the lamp device.

Functional profiles can also be a drawback since there is no way that the specification can foresee every need. Therefore many manufacturers implement their own proprietary extensions, thus destroying interoperability. An example of a lack of functionality is the absence of network variables that define a calendar or a schedule. Although LonWorks can support scheduling and data logging through the use of *programmable devices* as an alternative to define such concepts [10].

Zoning in LonWorks is supported through the creation of logical virtual networks within the physical network's structure this is achieved by creating domains or by creating subnets. Domains are used to separate large, independent groups of devices in a network, for example, separating lighting systems from HVAC systems. Subnets are logical groups of devices within a domain. For example, a subnet can be a group of all devices room. Domains and subnets provide LonWorks with the capabilities of zoning, i.e., grouping devices by their zones. Inside subnets nodes can be addressed individually or by broadcasting. Another way to associate devices in a way that's independent of domain-subnet-node addressing is by creating a group. Groups are collection of devices, where each member can communicate with others through the group identification. Groups can be also used for broadcasting messages. A limitation of groups, domains and subnets is that there's a limit number of each one that can be created. For example, subnets can only have 127 devices and domains can only have 255 subnets.

3.1.3 X10

X10 is a popular protocol for HAS that first appeared in the 1990s. This system primarily uses the power line wiring as communication bus for signalling and control, between the many devices, it is capable of connecting a maximum number of 256 devices in the same environment. Usually X10 devices plug into power outlets where other household appliances are plugged and the system is mostly used for remote controlling, depending on the user's direct inputs, using simple controllers. For zoning functionality and configuration purpose the address space are divided in two: one address for house address, and the other for the device address. This way, controllers are able to control one device independently, using the combination o house and device addresses, or control a group of devices using the house address. Configuration can be very simple, the user will only have to manually select a letter from A to P, corresponding to the house address, and a number from 1 to 16, corresponding to the device address, and commands are pre-defined. X10 controllers range from extremely simple to very sophisticated controllers. In order to build more complex systems sophisticated controllers that can be software

programmed are used, these have the capability for performing complex tasks such as scheduled actuation, software-based group control and event driven actuation. This system's popularity is due to its simplicity and ease of use and installation, it has some reliability issues mainly because of the power line interferences and it's inability to deal with lost commands but it performs well in what concerns to user expectations. Depending on the desired system's functionalities and relative complexity, costs can range from low to medium, relatively to other HAS.

3.2 Wireless Systems

3.2.1 EnOcean

EnOcean is a system developed by the EnOcean Alliance and its main focus is to provide customers with a self-powered wireless BAS. Its devices rely on energy harvesting technologies such as micro-energy motion converters, thermo converters, that explore small mechanical variations, heat dissipation energy, etc. to feed ultra low power electronics that are part of sensors, actuators and controllers, capable of communicating wirelessly, in order to provide a battery independent wireless automation system[13]. EnOcean devices are able to communicate through radio signals that can be transmitted over a distance of up to 300 meters outdoor and up to 30 meters indoor.

Recently the EnOcean wireless standard was approved by the International Organization for Standardization (ISO) and is now known as the 14543-3-10 standard. Since EnOcean's standard only covers the Open Systems Interconnection (OSI) layers 1-3 which are the physical, data link and networking layers, this standard doesn't specify an information model at the application level, having just the basic functionality of a network protocol, such as broadcasting and basic address grouping.

3.2.2 ZigBee

ZigBee is a standard maintained and published by de ZigBee Alliance. Among the many applications of ZigBee-based systems is BA. ZigBee systems bring advantages like low-power consumption, wireless communication thus reducing wiring costs and installation intrusiveness. The ZigBee specification defines a mesh network with low-power operation, self-healing and self-configuring capabilities, and the possibility to use with energy harvesting devices, assuring high degrees of flexibility and mobility. ZigBee is built on top of the IEEE 802.15.4 standard [15]. ZigBee provides routing and multi-hop functions to the packet-based radio protocol, adding the network layer as well as security and an application framework. ZigBee can operate in three frequency bands. It operates in the 2.4 GHz band world-wide. The 868 MHz band only operates in Europe and the 915 MHz band is only for North and South America. The ZigBee specification is open and it defines an architecture where devices can share, discover and offer their services. The information model is mainly defined across three layers that manage the application layer's domain instances (such as device objects, endpoints and bindings between those endpoints), these are the Application Support Sublayer, ZigBee Device Object and Application Framework. The Application Support Sublayer is responsible for binding endpoints (where an endpoint corresponds to an address-

able unit on a device), message forwarding between bound devices and group address definition and management. A group address is based on the concept of clusters, where each service is identified by a device profile where it belongs and a Cluster ID containing the service's identification, an application can then read from and write to service clusters (that may result in the interaction with multiple endpoints) instead of addressing each endpoint individually. The ZigBee Device Object is responsible for overall device management, specifically it is responsible for defining the operating mode of the device (if it coordinates ZigBee communications or acts as an end device), device discovery and determination of which application services the device provides (whereas each application service corresponds to an endpoint of a device), and for handling binding requests from other devices or coordinators. Finally the Application Framework is where each device's applications lie. An application object can be a light bulb, a light switch, an LED, an I/O line, etc. Each ZigBee device can have up to 240 applications, thus 240 endpoints are reserved for this purpose in each device. The ZigBee information model doesn't specify anything in relation to event notification, alarms and task scheduling at the application level, but such features can be supported by some application objects on certain devices, for example a fire alarm may notify a cluster that smoke was detected

3.2.3 WeMo

WeMo is a product line manufactured by Belkin that combined provide enough functionalities to build simple Home Automation Systems. Besides providing such functionalities, WeMo[3] products are focused on home appliances remote control using a smartphone. The product line is made of two actuators, a light switch and a power outlet, and a sensor, a motion sensor. All WeMo devices make use of the home's wi-fi network to communicate, saying this, the number of devices is limited to how many the network router can accept, with the clear disadvantage that the interaction with the WeMo system may interfere with the local network performance. Given the computational power and complexity that each device has, prices for the system components are relatively high. Belkin provides a WeMo mobile application software that is used to easily configure and control the whole system. Controlling the system is very simple, based on toggle buttons provided by the application interface, so user is capable of turning on or off devices connected to each of the actuators. The WeMo system does not provide any king of zoning capabilities, so each component has to be controlled individually. Automation scenarios can be set up using what Belkin calls *Rules*[4], these rules are no more than logical connections between the actuators and the sensors, or timer in the case of schedules. These *Rules* are limited to three types: toggle (on or off), turn on then off, and turn on or off based on the sensor state. The first two provides the scheduling functionality, e.g., the user can choose to turn on the light at a specific time, or to turn on the light during a specific time period and turn off when it ends; the last one, as the name suggests, provides automated control based on motion sensing, e.g., turn the lights on when user gets inside the room and passes through the sensor sensing space and off when no motion is sensed. WeMo does not provide alarm and notification functionalities. Besides the low functionality issue of this system, it benefits from great

[3]WeMo Docs, `http://www.belkin.com/us/support-product?rnId=01t80000002yNiUAAU`
[4]WeMo review, `http://www.wired.com/reviews/2013/03/belkin-wemo/`

extensibility through the integration with a recently created web service called If This Then That (IFTTT). IFTTT[5] enables WeMo users to configure advanced automation tasks with the possibility of integrating with web channels that somehow complement the lack of functionalities such as notification and zoning, and introduce new possibilities in terms of social networks interaction for example.

3.2.4 LIFX

LIFX is a recently developed innovative system made by a kind of light bulb that can be used for remote control and automation purposes [6]. LIFX is made of individual wi-fi enabled LED bulbs, that together form a highly customizable and energy efficient illumination system. Each bulb is made of two communication interfaces, a wlan interface and a 802.15.4 wireless mesh interface. The LIFX bulbs can operate in two modes, the gateway mode and the node mode. The difference between the two modes is that a LIFX bulb in gateway mode is used to provide communication with many LIFX bulbs in node mode and to provide an interface between the LIFX system and the local wireless router, allowing devices (e.g. smartphones) on the network to remotely access the LIFX system. For the wireless router communication, the wlan interface is used, and for communicating between LIFX bulbs, the 802.15 interface is used. The purpose of these two operation modes is mainly to explore the energy efficiency and signal range that mesh networks provide, as well as to surpass the limitation for the number of devices that can be connected to the wireless router. The installation process is software based. LIFX labs provides a software application for smartphones that is used to control and configure the system. Users can perform simple control tasks such as toggle LIFX lamps on and off, change color and dimming. For more complex tasks, the systems supports zoning and scheduling functionalities, so groups can be created using the mobile application and/or schedule one of the mentioned simple control tasks to a determined time instant or time period. LIFX is still evolving and provides an API and integration with other devices and web services, such as IFTTT, in order to bring great extensibility. There are solutions similar to LIFX, the most popular are Phillips Hue Phillips HUE, `https://www.meethue.com/` and Ilumi ilumi, `http://ilumi.co/`.

3.3 Discussion

In order to compare the above described systems, a typical office automation scenario is considered. This scenario also corresponds to the typical requirements.

3.3.1 Scenario

Consider an office room with a typical configuration, a door, two windows, one for the interior and the other for the exterior, a motor-operated venetian blind, two work tables, a dimmable desk lamp and simple appliances like a radio or coffee pot. The office rooms control systems have some requirements

[5] IFTTT, `https://ifttt.com/wtf`
[6] LIFX support, `http://support.lifx.co`

which each of them may be classified based on its implementation complexity, these requirements, the corresponding classification and functionalities needed are described as follows, in ascending order in terms of complexity:

R1 At night, when the occupant arrives the blinds are to be closed automatically to guarantee the privacy of the occupant. (Complexity: 1; Functionalities: scheduling or sensing, actuation)

R2 When the occupant leaves the room, the luminaries, the desk lamp are other simple appliances are to be automatically turned off. (Complexity: 1; Functionalities: sensing, actuation, zoning)

R3 The windows can be open manually. When they are open, the HVAC system should be stopped at once. (Complexity: 2; Functionalities: sensing, actuation)

R4 The person responsible for the offices should receive a notification if it is occupied after hours. (Complexity: 2; Functionalities: sensing, notification)

R5 The person responsible for the offices should be able to control the devices in the offices. (Complexity: 2; Functionalities: remote access, actuation)

R6 Whenever the occupant moves around the room the luminaries should increase their intensity and then fallback to the appropriate intensity as he sits to work. (Complexity: 3; Functionalities: sensing, actuation)

R7 When the occupant arrives the blinds are to be opened completely to maximize daylight. (Complexity: 1; Functionalities: scheduling, sensing, actuation)

R8 The intensity of the luminaries should be adjusted depending on which table the occupant is sitting. If the occupant moves to the table near the door the luminary near the door should be at a higher intensity level than the one nearer to the window. (Complexity: 3; Functionalities: sensing, actuation)

R9 The luminaries are to be dimmed according to the desired interior luminance performing automatic daylight control based on the exterior luminosity. (Complexity: 4; Functionalities: sensing, actuation)

R10 When the occupant starts working the blind should be adjusted automatically to prevent excessive glare (Complexity: 4; Functionalities: sensing, actuation)

R11 When in daylight control, luminaries should work in progressive dimming. Each luminary may display a distinct dimming level: the one nearer the window requires less intensity than the one nearer to the door to guarantee the same luminance level. This effect is best observed at dawn or at sunset. (Complexity: 4; Functionalities: sensing, actuation)

For a BAS to fulfil all the requirements presented above it would need the following components:

- 1 Motion sensor for simple occupation estimation (R1,R2,R4,R6 and R7)

- 1 Blinds motor actuator (R1,R7 and R10)
- 1 Exterior light sensor (R9,R10 and R11)
- 2 Magnetic contact switches (door and window sensors) (R3)
- 2 Simple relay actuators (for simple appliances) (R2)
- 4 Dimmable lamps (R9 and R11)
- 2 Light sensor for the tables (R8)
- 2 Pressure sensors for the seats (R8)

3.3.2 Comparison

Several systems that try to solve the problems stated in Section 1.1 where described. Table 3.1 summarizes some of the technologies' features that may be relevant to this project. This table is based on the technologies' specifications and in extra functionalities offered by third party systems.

Functionalities	LIFX	WeMo	X10	ZigBee-based	EnOcean	LonWorks	KNX
Zoning	Yes	Yes	Yes	Yes	No	Yes	Yes
Notifications	No	No	Yes	Yes	Yes	Yes	Yes
Scheduling	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Scenario's Coverage	9,09%	27,27%	45,45%	100,00%	45,45%	100,00%	100,00%
Complexity	Low	Low	Medium	High	High	High	High
Installation Dificulty	Easy	Easy	Easy	Difficult	Difficult	Difficult	Difficult
Relative Cost	Medium	Medium	Low	Medium	Medium	High	High

Table 3.1: Summary of the studied technologies

Requirements	**LIFX**	**WeMo**	**X10**	**ZigBee-based**	**EnOcean**	**LonWorks**	**KNX**
R1	○	○	●	●	●	●	●
R2	○	●	●	●	●	●	●
R3	○	○	●	●	●	●	●
R4	○	●	○	●	○	●	●
R5	●	●	○	●	○	●	●
R6	○	○	●	●	○	●	●
R7	○	○	●	●	●	●	●
R8	○	○	○	●	●	●	●
R9	○	○	○	●	○	●	●
R10	○	○	○	●	○	●	●
R11	○	○	○	●	○	●	●

Table 3.2: Requirements fulfilment mapping to each system. ●=Requirement fulfilled ○=Requirement not fulfilled

Chapter 4

Implementation

Aiming at designing, implementing and evaluating an automation solution that is low cost, rich in functionalities for the Internet-Of-Things, our proposal is based on open hardware technologies, in this case, using the Arduino platform. The goal was to develop an easy to install system, inexpensive, rich in functionalities and capable of integration with other systems, to be used for automating offices and small houses.

4.1 Solution Overview

Arduino is an open hardware platform which has been frequently used in order to enable fast prototyping[1]. The Arduino-based controller will orchestrate all the communication between the field devices, and same as being the system's gateway. At the Field Level, specially designed DMs will be developed using atiny micro-controllers [2] and other application dependent components, as the field devices for actuation and sensing capabilities. The field bus protocol will be implemented on top of the ubiquitous Inter-Integrated Circuit (I2C) bus protocol, the applicability to this kind of systems will be evaluated.

4.1.1 Fieldbus (Physical Layer)

I2C is a simple, bidirectional 2-wire communication bus, developed by Philips Semiconductors (now NXP Semiconductors) in 1982, aiming at efficient inter integrated circuit control [22]. All I2C-bus compatible devices incorporate an on-chip interface which allows them to communicate directly with each other via the I2C-bus. Nearly all micro-controller units are I2C compatible devices thus simplifying the field bus implementation of the proposed system. In particular, the AVR micro-controller used in the Arduino platform.

The I2C bus communication protocol is widely-used for many applications that require low cost and simple but reliable implementation. The main benefits from using the I2C bus and compatible devices can be organized as follows:

[1] Arduino's website, `http://www.arduino.cc/`
[2] ATiny's page, `http://www.atmel.com/products/microcontrollers/avr/tinyavr.aspx`

- Multiple devices communication with only two bus lines (two wires). A Serial Data Line (SDA) and a Serial Clock Line (SCL).
- Each device connected to the bus is software addressable by a unique address. Thus a simple master/slave relationships exist at all times.
- It is a multi-master bus including collision detection and arbitration (CSMA/CD).
- Flexible data transmission rates. Up to 100 kbit/s in the Standard-mode, up to 400 kbit/s in the Fast-mode, up to 1 Mbit/s in Fast-mode Plus, or up to 3.4 Mbit/s in the High-speed mode.
- No need to design bus interfaces because the I2C-bus interface is already integrated on-chip;
- Integrated addressing and data-transfer protocol allowing systems to be completely software-defined.

Despite the attractive features of I2C, the bus protocol has limitations, that can be circumvented through the use of additional components [21]. These limitations are the following:

Addressing - Since only 7-bits (or 10-bits) are available for device addressing, devices on the same bus sometimes share the same address. Some devices are capable of configuring the last few bits of the address, but this still imposes a limitation of devices on the same bus.

Reliability - The shared nature of the I2C bus may result in entire system failure if a single device fails to work properly.

Scalability - The electrical limitations of the I2C bus typically limit the number of devices on a bus to around a dozen devices.

Regarding the architecture communication sub-system, our solution will have the Arduino-based controller as the master, controlling the state of all devices. DMs will work as slaves, responding to the controller's commands. This topology (single master - multiple slaves) according to I2C specifications is depicted in Figure 4.1. To make the wiring and connection simpler, modular connectors of the type 6P4C (6 plugged, 4 connected) will be used.

4.1.2 Controller

An Arduino will be used as Controller, to control the communication and system's parameters. The Arduino will run a firmware that will perform real time control and adjust of the DM parameters, according to a control logic defined in the server's software, e.g. Finite State Machines (FSM) [18].

The Controller and all the DMs will be provided with a network interface module to connect these to the I2C bus.

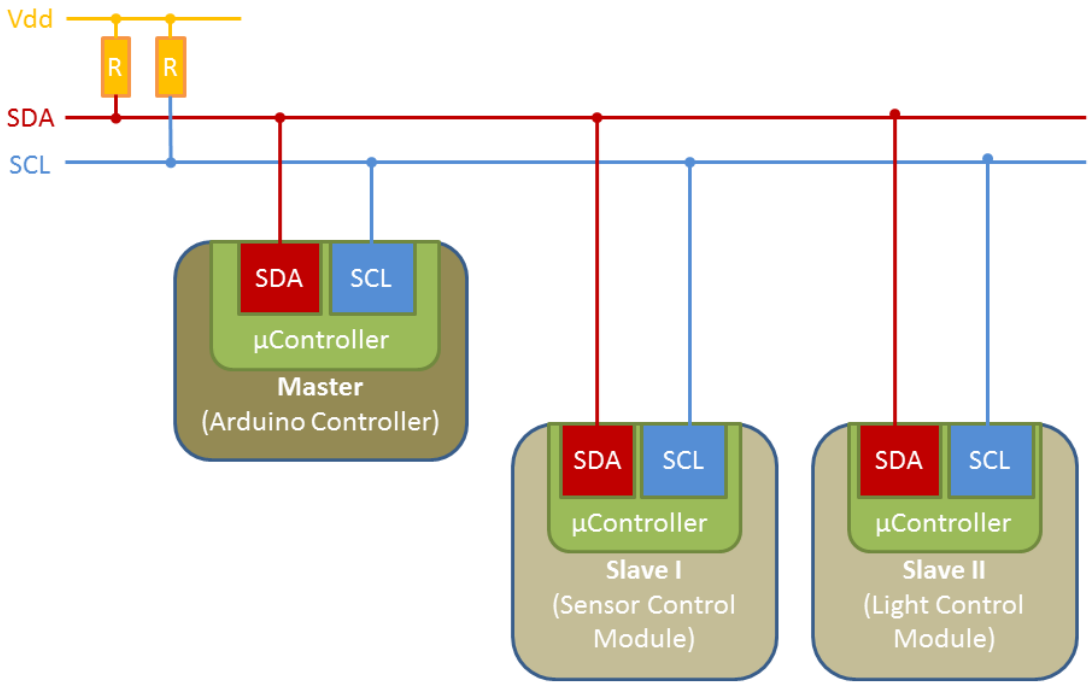


Figure 4.1: Illustration of I2C connections and single-master/multiple-slaves topology implementing home automation modules. An Arduino controller acts as a master and the DM as slaves.

4.1.3 Device Modules

The DMs are responsible for interfacing the system with the environment. These DMs consist of two sub-modules: the network interface module and the hardware device driver module to control sensors and actuators. The proposed system's components architecture is depicted in Figure 4.2. The software running on these modules consists of two blocks: one for the communication protocol and the other based on the device functionalities, e.g. light control actuator.

4.1.4 Functionalities

The level of functionalities is mainly limited by the effort that is taken when developing the software that will run on the server. This solution is capable of having a sophisticated behaviour when the software running on the server comprises functionalities that meet this behaviour. Some of the functionalities can be implemented as described below:

Device Commands: As the DMs will be individually accessible by the Controller in an Master/Slave protocol, the latter will naturally issue defined **commands** to the DMs.

Scheduling: a device-state/address pair is associated to a time instant or time interval triggered by a real-time clock.

Grouping: Through the use of software it will be able to build DM **groups** mapping interface types to the compatible existing devices addresses.

Zoning: For **zoning** functionalities two possibilities will be studied. The first possibility is to build zones

using multiple Controllers (Multi-Master/Multi-Slave), where each Controller aggregates DMs in a determined zone, and a Master Controller responsible of controlling all the controllers. The other second is similar to the one stated for grouping functionality, where a table is built to map zone addresses to devices addresses; this possibility is naturally less scalable.

More complex relations like the ones stated in Section 3.3.1 will be provided through the use of a dedicated Computer Server connected to the Controller through the Internet running more complex control algorithms performing readings and issuing commands to the system.

4.1.5 Envisioned Devices

To assure the maximum coverage of the major HA tasks, a minimum number of DM devices must be developed according to the defined architecture. To validate our architecture, the following modules were developed: , , , , and .

- A **relay DM**, for simple power toggle operations
- A **dimmer DM** for dimming functionalities
- An **occupancy DM** for occupancy detection
- A **sensor DM** for multiple environment physical parameters sensing (e.g. Temperature, Humidity)
- A **luminance DM** for luminance sensing
- An **Infra-Red DM** to receive and send infra-red command

The proposed system's architecture and the distribution of the devices through the layered architecture model presented in Section 2.1 is depicted in Figure 4.3.

4.2 Controller

The Controller was implemented using the Arduino platform. Two stackable structures where added to it, one locally designed with modular connectors was added to it to facilitate the connections to the bus, the 6P4C connectors(6 plugged, 4 connected), where two pins correspond to the I2C wires, and the other two are used to supply power to each module on the bus, and other, an Ethernet interface for the Arduino.

To operate the Devices on the bus, an API was developed to provide access to their most important features. This API provides the access to Read/Write operations to determined pins on the DMs. The API functions are described in Table 4.1.

Function	Desctription	Parameters	Returns
analogRead()	Read the value of an analog signal on the desired pin	Pin to read Variable to store error	Value read
analogWrite()	Set the desired pin to output aPWM signal with the chosen duty cycle	Value to write [0,254] Pin number Variable to store error	NA
digitalRead()	Read the value of a digital signalon the desired pin	Pin to read Variable to store error	Value read
digitalWrite()	Set the desired pin to output aa digital signal or not.	Value to write $\{0,1\}$ Pin number Variable to store error	NA
isAvailable()	Sees if the device is connected and stores the correspondent version number which can be used to identify different type of devices	Variable to store version	Error data
begin()	Initializes the object with the chosen address	I2C device address number	NA

Table 4.1: Description of functions that are part of the API.

4.3 Device Modules

The DMs developed during this project can be divided in to main modules, The Network Interface module and the Application module. The Network Interface is the same among all DM whereas the Application module is the piece of hardware that is dependant to the application of that DM, sensor or actuator.

The network interface and corresponding communication protocol implemented in this project where developed for generic purposes with the help of some colleagues.

4.3.1 Bus Interface

This interface was developed as an I/O extension for the Arduino platform, the first prototype is depicted in Figure 4.4. It is built on top of the ATtiny48 micro-controller [4] and features sixteen I/O pins with stackable design, where six are for Analogue signal reading functionalities, e.g. reading sensor signal, other three as Pulse Width Modulation (PWM) signal output pins, e.g. operate motors, and seven pins for digital I/O operations, e.g. operating relays.

To make the wiring and connection simpler, the interface uses modular connectors, the same as explained in Section 4.2, which are used to connect to the bus and get the needed power, as shown in Figure 4.4.

The firmware embedded in the micro-controller has been designed to perform operations according to commands sent by the controller through the bus (Section 4.4).

4.3.2 Application Hardware (Device Driver)

Being of stackable design, it was possible to create DMs, by designing and mounting the application dependant electronics on the network interface. Having the application components sub-module decoupled from the network interface makes the system more flexible and easier to install and the main

advantage is that if a module gets broken repairing will be easier and costs will be lower than if it was coupled in one module. Part or the whole logic above the DM basic Read
Write operations must be embedded in the Server's Application, as exemplified in Section 4.5.

To identify the different types of DM, two hypothesis where considered. The version code could be embedded in the firmware of the DM controller, and retrieved using the function *isAvailable()* (see Section 4.2); or the version code could be set using a DIP switch installed as part of the Hardware, and retrieved using the basic digital reading function available in the API.

Relay DM

This DM is an ON-OFF actuator that aims at controlling some appliance accessible to the system. It will use a solid state relay as shown in Figure 4.5.The Relay is manufactured by Omron [3] and is capable of controlling a AC 240V load of 2A.

(a) Relay Actuator PCB (b) Relay Actuator Photo

Figure 4.5: PCB design of the Relay DM.

Button DM

The Button DM is a sensor that interfaces human actions with the system. It uses a simple touch button, as depicted in Figure 4.6 with the purpose of demonstrating how human actions can be read by the system.

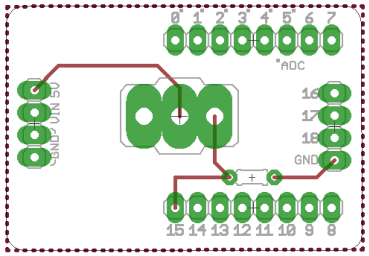


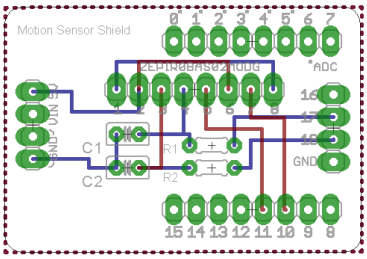
Figure 4.6: PCB design of the Button DM.

Motion SensorDM

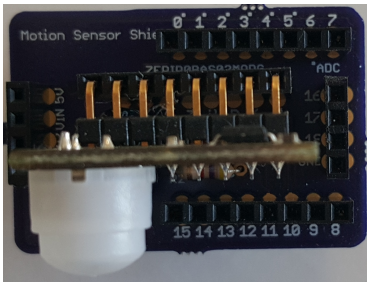
As the name suggests, this DM is a sensor that is capable of detecting motion and make it accessible to the system. It uses a Passive-Infrared based motion detector as depicted in Figure 4.7. The motion

[3] Omron G3MC, `http://www.omron.com/ecb/products/pdf/en-g3mc.pdf`

detector used in this DM is made by Zilog [4].



(a) Motion Sensor PCB

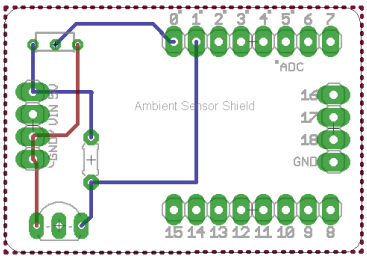


(b) Motion Sensor Photo

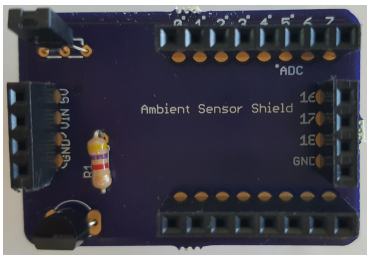
Figure 4.7: PCB design of the Motion sensor DM.

Ambient SensorDM

To gather information about surrounding ambient, this DM, as shown in Figure 4.8 combines to sensors, a precision temperature sensor and a humity sensor, to perform corresponding measurements. The temperature sensor made by Texas Instruments [5] capable of measuring temperatures with an accuracy of +/-5ºC. The Humidity sensor comes from Honeywell [6] and measures humidity with an accuracy of +/-8%RH.



(a) Ambient Sensor PCB



(b) Ambient Sensor Photo

Figure 4.8: PCB design of the Ambient sensor DM.

4.4 Field-Bus Protocol (Application Layer)

The implemented protocol is based on the Master/Slaves communication schema. The Controller acts as Master and the DMs as slaves. There are to type of messages, the Request message and the Response message. The message frame structure for the Request is defined with a header of 2 bytes, for the pin number, register and operation information, a with a data payload of 1 byte, as depicted in Figure 4.9.

The Response message is defined with a header of 1 byte, for the error information and with a data payload of 1 byte also, as depicted in Figure 4.10.

[4] Zilog ZMOTION Detection Module II, http://www.zilog.com/index.php?option=com_cutsheet&task=view&cid=13&id=13&Itemid=112

[5] Texas Instruments LM335A, http://www.ti.com/product/LM335A/datasheet

[6] Honeywell HIH-4010, http://sensing.honeywell.com/honeywell-sensing-hih4010-4020-4021-series-product-sheets-009020-1-en.pdf?name=HIH-4010-002

4.5 Control Server Prototype

The implemented software demonstrates the possibilities of using the system for HA purposes. When developing a software to be used with this system, one important decision to make is what will be the level of complexity associated with the firmware on the Controllers and Server. The easy decision is to put most of the complexity in the Control Server's software, and leaving to the Controllers the task of reacting and responding to commands sent by the Server. This path leads to less programming effort but may negatively affect communication efficiency, as number of messages between Server and Controllers will be high. This impact in communication can even be more or less significant depending on the communication protocol used.

4.5.1 Functionalities and UI

The implemented Control Server Prototype offers basic functionalities such as **Module Type Identification**, **Analog Voltage Readings** and **Digital and Analog Write Operations** depending on the type of the device module, whether it is a sensor or an actuator. These functionalities are visible in the user interface of the prototype as shown in Figure 4.11.

The software is capable of communicating with the controller and automatically receiving all information about the addresses, type and state of the DMs connected to the system. It is also possible to send basic commands to manually operate the actuator DMs. The Server is connected to the Controller through a TCP/IP connection, and the connections of the Controller to the DM is through the system's I2C-based Field-bus, as depicted in Figure 4.12.

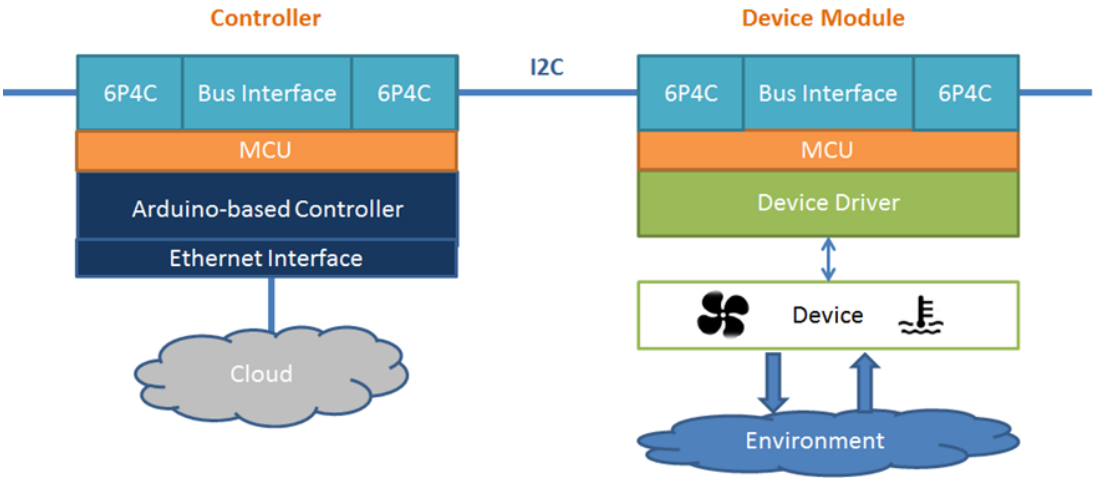


Figure 4.2: Controller and Device Module, with corresponding proposed architecture. The illustration shows the Controller and the DM connected to the bus, through their Bus interface. As evidenced by the lower layers in each device, the Controller's task is to interface the system with the Internet or Server using the Ethernet interface, whereas the DM task is to interface the system with the environment using the device driver.

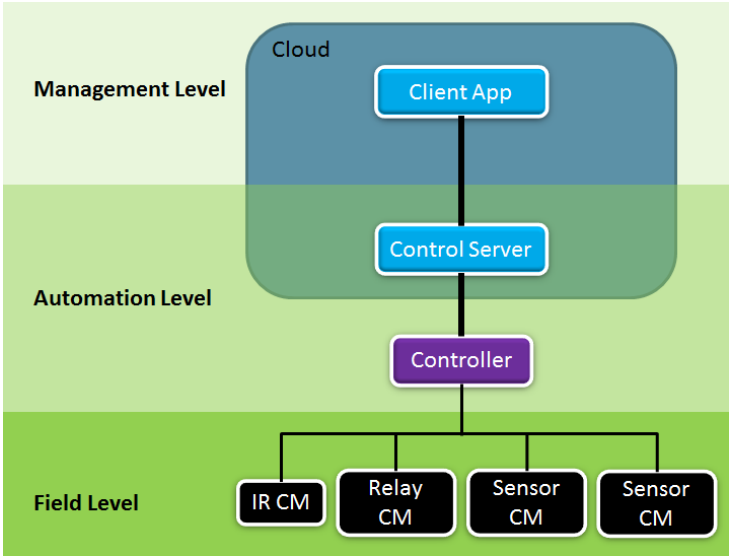


Figure 4.3: The proposed system's architecture and the distribution of the devices through the layered architecture model.

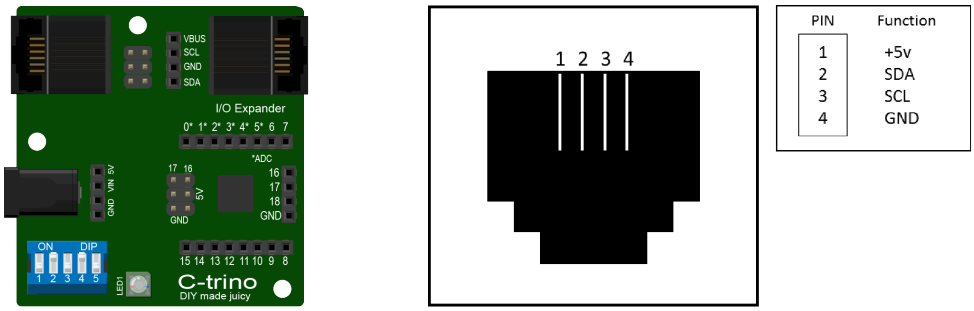


Figure 4.4: Illustration of the interface, named C-trino, where the real layout and parts like the modular connectors, the pins and the dip switch of the first prototype can be seen (Left). Modular connectors pin functions (Right).

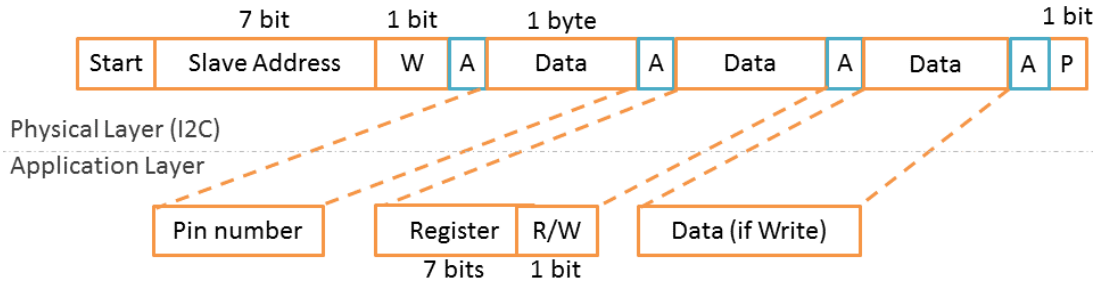


Figure 4.9: Protocol Request message structure. Orange colour corresponds to data flowing from Controller to DM, likewise, blue corresponds to flow from DM to Controller. R-read, W-write, A-Ack, P-stop.

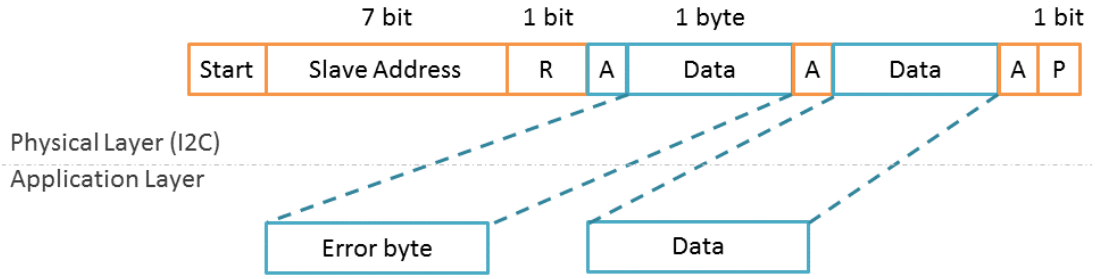


Figure 4.10: Protocol Response message structure. Orange colour corresponds to data flowing from Controller to DM, likewise, blue corresponds to flow from DM to Controller. R-read, W-write, A-Ack, P-stop.

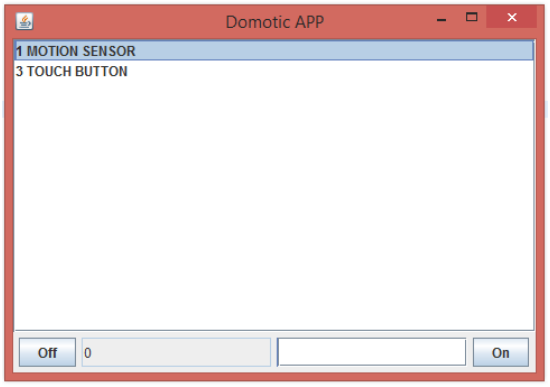


Figure 4.11: Control Server User Interface and corresponding functionalities identification.

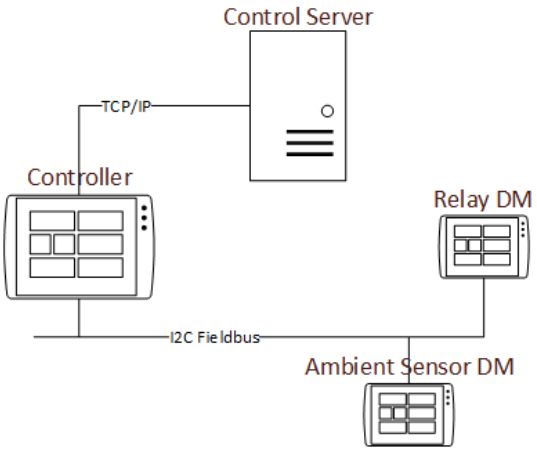


Figure 4.12: Control Server, Controller and DMs connection topology. TCP/IP between Server and Controller, and system's Field-bus between Controller and DMs.

Chapter 5

Evaluation

This work has resulted in system that would take the access to HASs much closer to every person, providing a fair level of functionalities associated with low prices, while maintaining a certain level of responsiveness and effectiveness. This evaluation tries to validate the developed system according to this parameters. In order to test commands effectiveness and responsiveness systematically, a test bed was developed. Operations to the system through the bus where performed under different conditions.

5.1 Testbed

The test bed was developed using two Arduinos, one connected to the bus and issuing commands to the DM, and the other one performing readings and writings to the same DM confirming that the expected result was met. All tests where orchestrated using the Junit unit testing framework. The topology is as depicted in Figure 5.1.

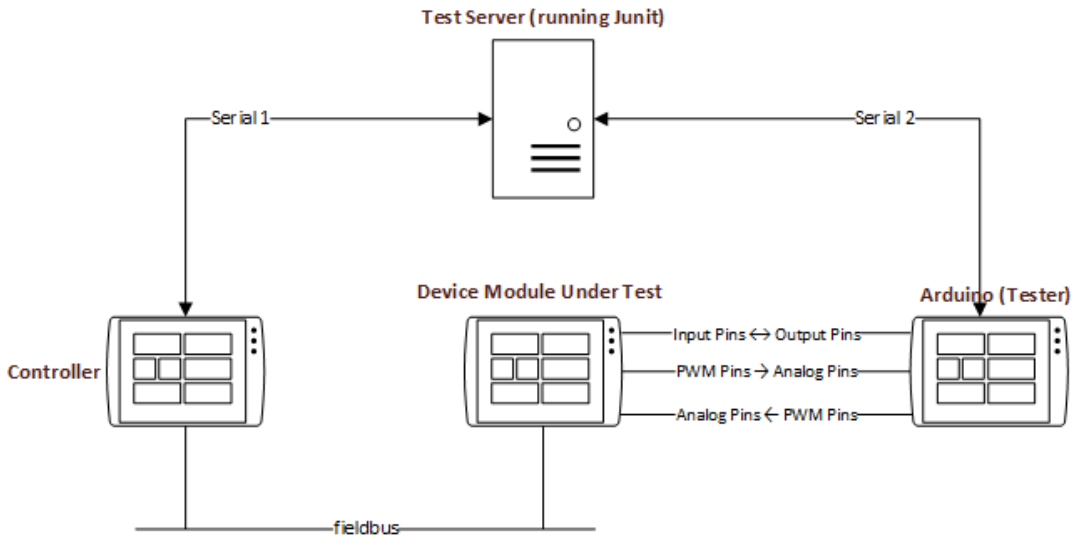


Figure 5.1: Testbed connection diagram. Shows pins interconnections between the Tester and the Device Module. The Test Server is connected to the Tester and to the Controller using two serial interfaces, where commands flow, and the Controller exchanges messages with the Device Module through the system fieldbus.

Two kinds of tests where developed, the Functionality Test and the Error Identification Test. The Functionalities test ensure that the commands are being executed as expected, whereas the Error Identification Test verifies that errors returned by the system given a wrong commands are the expected.

5.1.1 Functionality Test

The goal of this test is to ensure that commands to the bus are executed correctly and that the expected behaviour is met.

This test makes use of the basic Read/Write functions presented in Section 4.2. It starts by loading the pin mappings (DM-Arduino) from a file. This file must be previously created and the mappings have to be stored manually according to the pin connections made in the circuit. The functions that are test and corresponding Tester functions are listed at Table 5.1.

Functions (Controller ->DM)	**Tester Functions (Tester)**
digitalWrite(); values={0;1}	digitalRead(); values={0;1}
digitalRead(); values={0;1}	digitalWrite(); values={0;1}
analogRead(); values=[0;254]	analogWrite(); values=[0;254]
analogWrite(); values=[0;254]	analogRead(); values=[0;254]

Table 5.1: Functions to be tested, corresponding tester functions and valid values set.

digitalWrite Test Case

The digitalWrite Test Case will ensure that the command used to write a digital value to a DM is executed as expected. In this test case, the digital signal output of all the digital pins stored in the map are randomly set to 0 or 1 (0v or 5v) , the value set and corresponding pin are stored to be used for expected value confirmation. After performing this action in all pins on the DM, the Arduino that is used as Tester will perform readings to the corresponding pins and confirm that the value measured is the same value written before, in order to validate that the values written are actually the desired ones, as depicted in the simplified diagram 5.2.

digitalRead Test Case

The digitalRead Test Case will ensure that the command used to read a digital value from a DM is executed as expected. This test case starts by setting the digital pins on the Tester to output a digital signal with value randomly chosen between 0 or 1 (0v or 5v), these values and corresponding pin are stored. After these actions are taken on the Tester, the DM will perform readings to the corresponding pins and confirm that the measured value is the same value written before, in order to validate that the values that are being read are correct.

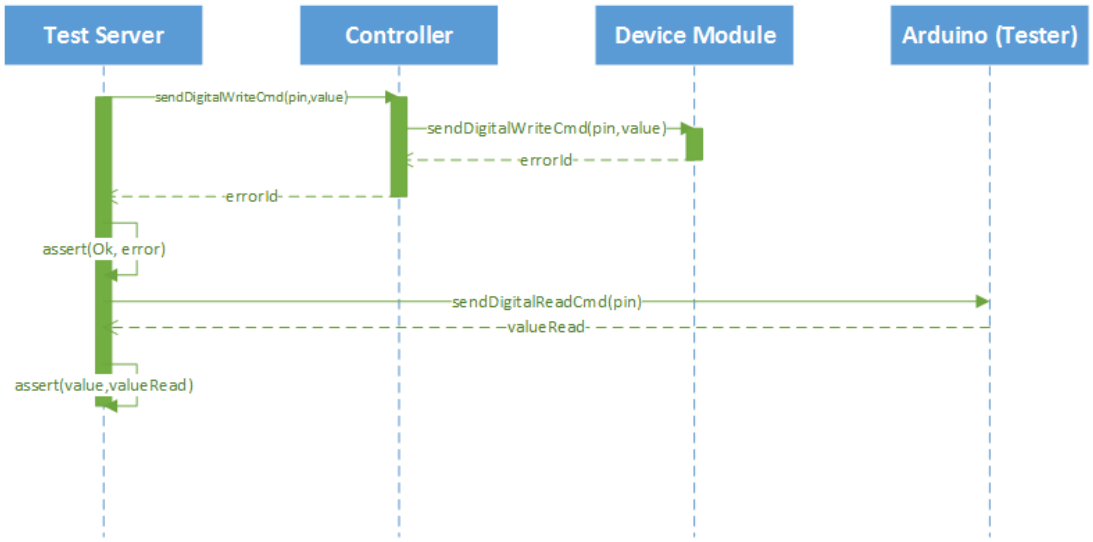


Figure 5.2: Test case for the digital write functionality. Digital values are first written to the DM, then are read by the Tester and finally these values are compared by the Test Server in order to confirm that the values written are correct. The diagram is simplified for easier interpretation.

analogWrite Test Case

The analogWrite Test Case will ensure that the command used to write an analogue value to a DM is executed as expected. This test case is similar to the digitalWrite Test Case, but instead of digital signals this one works with analogue signals and sets random values between 0 and 254 (0v - 5v). The Arduino used as Tester performs readings to the corresponding pins and confirm that the value measured is the same value written before, in order to validate that the values written are actually the desired ones. Being the output a PWM signal, it has to be connected to an Resistor-Capacitor filter, so the signal is converted to a real analogue signal, this conversion is not instantaneous and the test needs to wait between the test takes into account the delay needed for the values to stabilize.

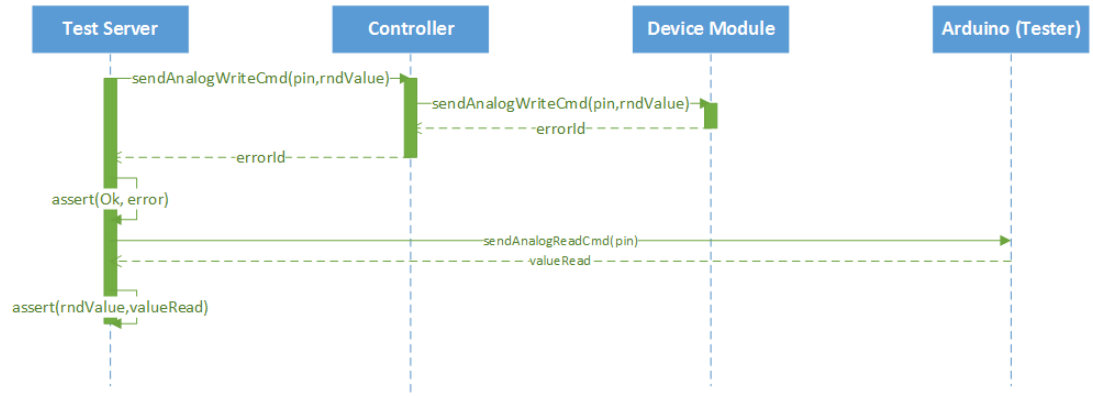


Figure 5.4: Test case for the digital write functionality. Digital values are first written to the DM, then are read by the Tester and finally these values are compared by the Test Server in order to confirm that the values written are correct. The diagram is simplified for easier interpretation.

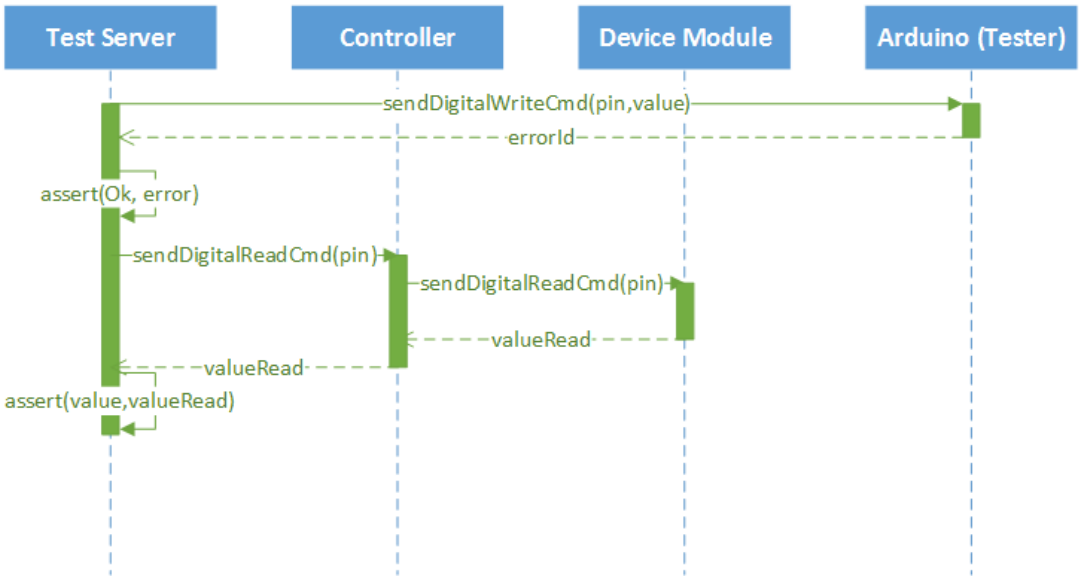


Figure 5.3: Test case for the digital read functionality. Digital values are written to the Tester, then are read by the DM and finally compared by the test server. The diagram has been simplified for easier interpretation.

analogRead Test Case

The analogRead Test Case will ensure that the command used to read an analogue value from a DM is executed as expected. This test case is similar to the digitalread Test Case. The DM will perform readings to the corresponding pins and confirm that the measured value is the same value written before by the Tester, in order to validate that the values written are actually the desired ones. It suffers from the same delay explained in the previous Test Case.

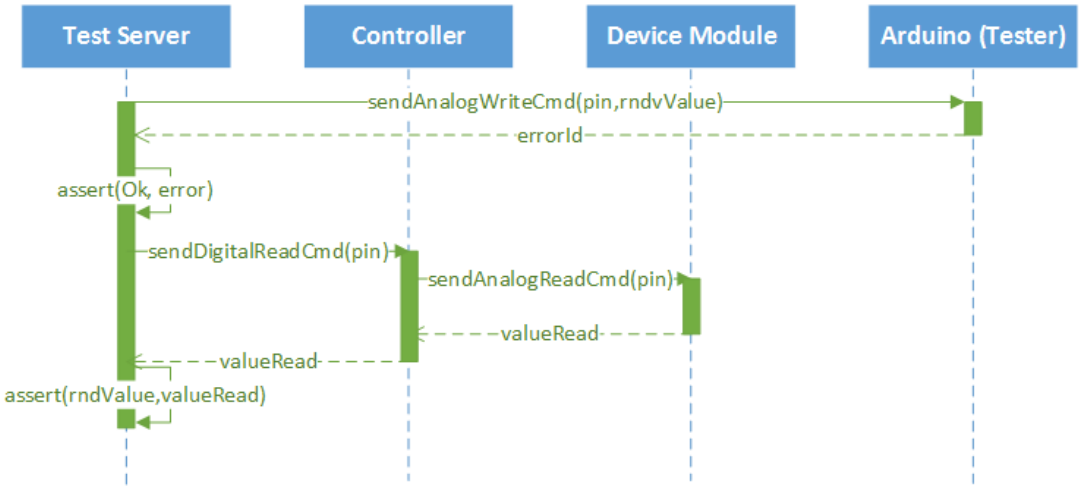


Figure 5.5: Test case for the digital read functionality. Digital values are written to the Tester, then are read by the DM and finally compared by the test server. The diagram has been simplified for easier interpretation.

5.1.2 Error Identification Test

This test was developed to ensure that if any erroneous action taken in the bus the specific error can be identified. The idea is to send erroneous commands that the resultant errors are already known, and verify if the error returned is exactly the same as the expected one. For example, one may try to send commands to a DM address that does not exist or from a DM that is not connected to the bus, this erroneous command must be easily identifiable by an error. Table 5.2 lists the errors that are detected as well as the actions that may result on those errors.

Error	Action
Invalid Board	Command to operate a DM that is not connected
Invalid Pin	Command to operate a pin that does not exist on the DM
Invalid Pin Operation	Command to perform an operation that is not supported by the indicated pin.

Table 5.2: Errors and corresponding erroneous actions

Chapter 6

Conclusions

The field of automation systems attracts high interest in the industrial segment as well as in the residential segment, however in the latter, the demand for such systems is not growing as expected, mainly due to the high costs and complex installation associated. Current systems such as X10, WeMo, etc. that try to overcome these limitations, but they all fail because they simply lack of one or a few of the desired aspects that affect the adoption of these systems for smaller applications. Indeed, since field of HA is very complex in terms of concepts and the task of developing a completely new system is difficult and may not seem a possible path to overcome the existing limitations. However, HA allied to the concepts of IoT and Open-source hardware is possible to create something distinct from the existing and that overcomes their limitations creating a low-cost, yet functionally rich and easy to use solution.

This Thesis developed a prototype low cost HAS that is capable of implementing major HAS tasks and comprises a modular architecture, that benefits flexibility and easy connections and simplified installation procedure. The system consists of two types of modules: Controllers which are responsible for orchestrating all the information that flows within the systems communication bus, and the Device Modules, that interface the system with the physical environment.

To validate this concept, Device Modules where also developed using a modular structure which the components responsible for each sensing or actuating task that are decoupled from the network interface, resulting in lower costs of maintenance, repair and simplicity. The complex logic that manages all the system's functionalities is programmed on a software that may be running on a local server or in the cloud. A simple HA software running on a local server was implemented to demonstrate the basic functionalities and capabilities of the system.

Finally we have a working system capable of most Home Automation tasks, tested and with an API for easier integration and future development.

For future work it is expected to increase the systems capabilities, in terms of distance between the modules for greater flexibility which can be achieved by hardware, as well as creating a more complex system consisting of multiple subsystems of this, with multiple Controllers, for greater scalability. We expect that this work opens a new path for retail automation systems solutions that may help growing the adoption for HASs in lower segment. The next step is to work on the prototype in order to improve

its capabilities and functionalities as the possibilities are endless.

Bibliography

[1] Araki M. Control Systems, Robotics, And Automation - Vol II - PID Control. Technical report, Kyoto University, 2009.

[2] ASHRAE. *BACnet A Data Communication Protocol for Building Automation and Control Networks (Standard 135-2004 – ANSI Approved)*, 2004.

[3] H. M. (Author), T. H. (Author), C. H. (Author), J. B. (Translator), V. M. (Translator), and L. G. (Translator). *Building Automation: Communication systems with EIB/KNX, LON and BACnet (Signals and Communication Technology)*. Springer; 2009 edition (November 30, 2012), 2012. ISBN 978-3642100253.

[4] A. Corporation. Attiny 48/88 microcontroller datasheet. Technical report, 2011.

[5] Daniel P. Bovet & Marco Cesati. *Understanding the LINUX KERNEL*. O'REILLY, 3rd Edition edition, 2006. ISBN 978-0596005658.

[6] Echelon Corporation. Functional Profile: Lamp Actuator. Technical report, LonMark, 1997.

[7] Echelon Corporation. Functional Profile: Switch. Technical report, LonMark, 1997.

[8] Echelon Corporation. *Control Network Protocol Specification (ANSI/EIA/CEA 709.1)*, 1999.

[9] Echelon Corporation. LonTalk Protocol Specification. Technical report, LonMark, 1999.

[10] Echelon Corporation. Introduction to the LONWORKS® System. Technical report, LonMark, 1999.

[11] Echelon Corporation. LONMARK® Application Layer Interoperability Guidelines. Technical report, LonMark, 2001.

[12] Echelon Corporation. LONMARK® SNVT Master List. Technical report, LonMark, 2002.

[13] A. Enocean. Enocean dolphin. In *System architecture for energy harvesting sensors and wireless sensor networks*, 2012.

[14] Harald Sundmaeker, Patrick Guillemin, Peter Friess, Sylvie Woelffle. Vision and challenges for realising the internet of things. In *CERP-IoT*, 2010.

[15] D. I. Inc. An introduction to zigbee. Technical report, Digi International Inc., 2008.

[16] Kastner, W. and Neugschwandtner, G. and Soucek, S. and Newmann, H.M. Communication Systems for Building Automation and Control. *Proceedings of the IEEE*, 93(6):1178 –1203, june 2005. ISSN 0018-9219. doi: {10.1109/JPROC.2005.849726}.

[17] Konnex Association. *KNX Specification Version 3.0*, 2009.

[18] B. Lee and E. A. Lee. Control logic using finite state machines. 1999.

[19] Moore Hawke. Introduction to Fieldbus. Technical report, Moore Industries-International Inc., 2006.

[20] Robert Bosch GmbH. *CAN Specification Version 2.0*, 1991.

[21] N. Semiconductors. True i2c-bus and buffering for long-distance communications. Technical report, 2007.

[22] N. Semiconductors. I2c-bus specification and user manual. Technical report, 2012.

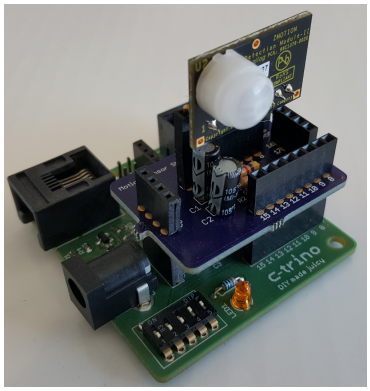
[23] E. Services and T. Association. Lighting control systems 0 to 10v analog control specification. Technical report, 2001.

[24] Wolfgang Granzer, Wolfgang Kastner, Paul Furtak. KNX and OPC UA. *Automation Systems Group*, 2011.

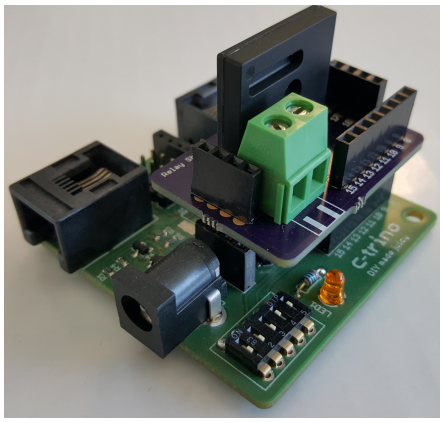
Appendix A

Device Modules

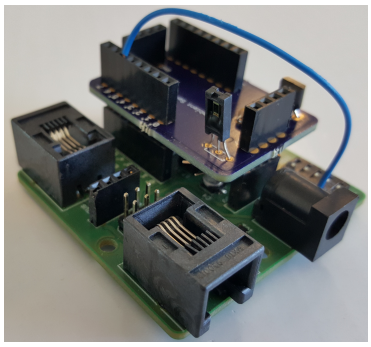
A.1 Photos



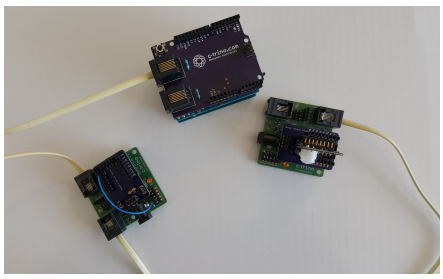
(a) Motion Sensor Device Module



(b) Relay Actuator Device Module



(c) Ambient Sensor Device Module



(d) Device Modules connected to Controller

Figure A.1: Photos of the Device Modules

A.2 Firmware

A.2.1 main.c

```
1
```

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

#include "ctrino.h"
#include "adc.h"
#include "io.h"
#include "TWI_slave.h"


static uint8_t  input_buffer[TWI_BUFFER_SIZE];
static uint8_t  input_buffer_length;
static uint8_t  output_buffer[TWI_BUFFER_SIZE];
static uint8_t  output_buffer_length;

__attribute__((flatten))
static
void ctrino_active(uint8_t input_buffer_length, const uint8_t *input_buffer,
    uint8_t *output_buffer_length, uint8_t *output_buffer)
{
  //        led_mode_transmitting();
  //        led_timer_reset();
  i2c_data_handler(input_buffer_length, input_buffer,
      output_buffer_length, output_buffer);
}

static
int ctrino_init(void)
{
  int ret = 0;
  uint8_t dipsw_value;
  if (io_init()) {
    ret = -1;
    goto out;
  }
  //        if (led_init() || led_mode_idle()) {
  //                ret = -1;
  //                goto out;
  //        }
  if (adc_init()) {
    ret = -1;
    goto out;
  }
  if (dipsw_value_get(&dipsw_value)) {
    ret = -1;
    goto out;
  }
  TWI_Slave_Initialise( (unsigned char)(dipsw_value<<TWI_ADR_BITS) | (TRUE<<TWI_GEN_BIT) );

```

```
51    sei();
52
53    // Start the TWI transceiver to enable reception of the first command from the TWI Master.
54    TWI_Start_Transceiver( );
55
56    out:
57    return ret;
58  }
59
60  int main(void)
61  {
62    if (ctrino_init()) {
63      // TODO: possibly try to put the leds into a very specific state
64      goto out;
65    }
66
67    for(;;)
68    {
69      // Here you can add your own code that should be run while waiting for the TWI to finish
70      // Check if the TWI Transceiver has completed an operation.
71      if ( ! TWI_Transceiver_Busy() )
72      {
73        // Check if the last operation was successful
74        if ( TWI_statusReg.lastTransOK )
75        {
76          // Check if the last operation was a reception
77          if ( TWI_statusReg.RxDataInBuf )
78          {
79            TWI_Get_Data_From_Transceiver(input_buffer, &input_buffer_length);
80            // Check if the last operation was a reception as General Call
81            if ( TWI_statusReg.genAddressCall )
82            {
83            }
84            else // Ends up here if the last operation was a reception as Slave Address Match
85            {// Put own code here.
86              output_buffer_length=0;
87              ctrino_active(input_buffer_length, input_buffer, &output_buffer_length,
      output_buffer);
88              input_buffer_length=0;
89              TWI_Start_Transceiver_With_Data(output_buffer,output_buffer_length);
90            }
91          }
92          else // Ends up here if the last operation was a transmission
93          { // Put own code here.
94          }
95          // Check if the TWI Transceiver has already been started.
96          // If not then restart it to prepare it for new receptions.
97          if ( ! TWI_Transceiver_Busy() )
98          {
```

```
99            TWI_Start_Transceiver();
100         }
101       }
102       else // Ends up here if the last operation completed unsuccessfully
103       {
104       }
105     }
106   }
107
108   out:
109   /* Never reached! */
110   return -1;
111 }
```

A.2.2 board.c

```
1  #include "adc.h"
2  #include "board.h"
3  #include "io.h"
4  #include "ctrino.h"
5
6
7  uint8_t cpcapr_r(uint8_t pin,
8      uint8_t input_buffer_length,
9      const uint8_t *input_buffer,
10     uint8_t *output_buffer,
11     uint8_t *output_buffer_length)
12 {
13   uint8_t ret = 0;
14   uint8_t cpcapr;
15
16   if (pin < ADCMAX) {
17     cpcapr = (1 << CADCA);
18   } else {
19     cpcapr = (1 << CDGOA) | (1 << CDGIA) |
20         (1 << CPUPA) | (1 << TRSTA);
21   }
22
23   output_buffer[0] = cpcapr;
24
25   return ret;
26 }
27
28 uint8_t cpcapr_w(uint8_t pin,
29     uint8_t input_buffer_length,
30     const uint8_t *input_buffer,
31     uint8_t *output_buffer,
32     uint8_t *output_buffer_length)
33 {
34   return CIRWOP;
```

```
}

uint8_t cpstr_r(uint8_t pin,
    uint8_t input_buffer_length,
    const uint8_t *input_buffer,
    uint8_t *output_buffer,
    uint8_t *output_buffer_length)
{
  uint8_t ret = 0;
  uint8_t cpstr;

  if (pin <= NPINS) {
    cpstr = 0 << CPWME |
        0 << CPOUV |
        get_pin_value(pin) << CPINV |
        0 << CPDIR;
  } else {
    ret = CIPIN;
    goto out1;
  }

  output_buffer[0] = cpstr;
  *output_buffer_length+=1;

  out1:
  return ret;
}

uint8_t cpstr_w(uint8_t pin,
    uint8_t input_buffer_length,
    const uint8_t *input_buffer,
    uint8_t *output_buffer,
    uint8_t *output_buffer_length)
{
  uint8_t ret = 0;
  uint8_t value;

  if (pin < NPINS) {
    if (input_buffer_length != 1) {
      ret = CIIDL;
      goto out1;
    }

    value = input_buffer[0];

    if (value>>CPOUV & 0x1)
      set_pin_port(pin,1);
    else
      set_pin_port(pin,0);
```

```
84    }
85    else{
86      ret = CIPIN;
87      goto out1;
88    }
89
90    out1:
91    return ret;
92  }
93
94  uint8_t cpmstr_w(uint8_t pin,
95      uint8_t input_buffer_length,
96      const uint8_t *input_buffer,
97      uint8_t *output_buffer,
98      uint8_t *output_buffer_length)
99  {
100   uint8_t ret = 0;
101   uint8_t value;
102
103   if (pin < NPINS) {
104     if (input_buffer_length != 1) {
105       ret = CIIDL;
106       goto out1;
107     }
108
109     if (PWMMIN<=pin && pin<=PWMMAX){
110       if(pin==16)
111       {
112         TCCR1A&=~(1<<COM1PIN_16);
113         OCRPIN_16=0;
114       }
115       if(pin==17)
116       {
117         TCCR1A&=~(1<<COM1PIN_17);
118         OCRPIN_17=0;
119       }
120     }
121
122     value = input_buffer[0];
123
124     if (value & 0x1)
125       set_pin_ddr(pin,1);
126     else
127       set_pin_ddr(pin,0);
128   }
129   else{
130     ret = CIPIN;
131     goto out1;
132   }
```

```
133
134   out1:
135   return ret;
136 }
137
138 uint8_t cpwmvr(uint8_t pin,
139     uint8_t input_buffer_length,
140     const uint8_t *input_buffer,
141     uint8_t *output_buffer,
142     uint8_t *output_buffer_length)
143 {
144   uint8_t ret = 0;
145   uint8_t value;
146
147   if (PWMMIN<=pin && pin<=PWMMAX){
148     if (input_buffer_length != 1) {
149       ret = CIIDL;
150       goto out1;
151     }
152     value = input_buffer[0];
153     if(value<=255 && value>=0){
154       TCCR1A=(1<<WGM10);
155       TCCR1B=(1<<WGM12)|(1<<CS10);
156
157       if(pin==16)
158       {
159         TCCR1A|=(1<<COM1PIN_16);
160         OCRPIN_16=value;
161       }
162       if(pin==17)
163       {
164         TCCR1A|=(1<<COM1PIN_17);
165         OCRPIN_17=value;
166       }
167     }
168   }
169   else{
170     ret = CIPIN;
171     goto out1;
172   }
173
174   out1:
175   return ret;
176 }
177
178 uint8_t cadcr_r(uint8_t pin,
179     uint8_t input_buffer_length,
180     const uint8_t *input_buffer,
181     uint8_t *output_buffer,
```

```
182     uint8_t *output_buffer_length)
183 {
184   uint8_t ret = 0;
185   uint16_t adc_output;
186 
187   if (! (pin < 6)) {
188     ret = CIREG;
189     goto out1;
190   }
191 
192   adc_set_channel(get_pin_adc_channel(pin));
193   adc_convert(&adc_output);
194 
195   output_buffer[0] = (uint8_t)(adc_output >> 8);
196   output_buffer[1] = (uint8_t)adc_output;
197   *output_buffer_length+=2;
198 
199   out1:
200   return ret;
201 }
202 
203 uint8_t cadcr_w(uint8_t pin,
204     uint8_t input_buffer_length,
205     const uint8_t *input_buffer,
206     uint8_t *output_buffer)
207 {
208   return CIRWOP;
209 }
210 
211 uint8_t cfver_r(uint8_t pin,
212     uint8_t input_buffer_length,
213     const uint8_t *input_buffer,
214     uint8_t *output_buffer,
215     uint8_t *output_buffer_length)
216 {
217   uint8_t ret = 0;
218   uint8_t address=0;
219 
220   if (pin == CTRINO_MAGIC_PIN) {
221     if(!dipsw_value_get(&address)){// Get the I2C address of the board from the dip switch.
222       output_buffer[0]=address;
223       output_buffer[1]=VERSION_CODE;
224       *output_buffer_length+=2;
225     }
226   }
227 
228   return ret;
229 }
230 
```

```
231 ctrino_msghandler ctrino_msghandlers[NREGS * 2] = {
232     cpcapr_w, cpcapr_r, cpstr_w, cpstr_r, cpwmvr, cpwmvr,
233     cadcr_w, cadcr_r, cadcr_r, cadcr_r, cadcr_r, cadcr_r,
234     cadcr_r, cadcr_r, cadcr_r, cpmstr_w, cpmstr_w, cfver_r, cpmstr_w, cpmstr_w,
235 };
```

A.2.3 board.h

```
1  #ifndef BOARD_H_
2  #define BOARD_H_
3
4  #include <stdint.h>
5
6  #define NPINS 19
7  #define PWMMAX 18
8  #define PWMMIN 16
9  #define ADCMAX 5
10 #define ADCMIN 0
11 #define NREGS 10
12
13 typedef uint8_t (*ctrino_msghandler)(uint8_t pin,
14     uint8_t input_buffer_length,
15     const uint8_t *input_buffer,
16     uint8_t *output_buffer,
17     uint8_t *output_buffer_length);
18
19 extern ctrino_msghandler ctrino_msghandlers[NREGS * 2];
20
21 #endif /* BOARD_H_ */
```

A.2.4 ctrino.h

```
1  #ifndef CTRINO_H_
2  #define CTRINO_H_
3
4
5  #include <avr/io.h>
6
7  #define MUX_ADC_CHANNEL 6
8
9  #define DIP1        PA0
10 #define DIP2        PC7
11 #define DIP3        PD0
12 #define DIP4        PD1
13 #define DIP5        PD2
14
15 #define LED         PD7
16
17 #define PORT_LED     PORTD
18 #define PORT_DIP_1     PORTA
```

```
19 #define PORT_DIP_2          PORTC
20 #define PORT_DIP_345    PORTD
21
22 #define DDR_LED         DDRD
23 #define DDR_DIP_1       DDRA
24 #define DDR_DIP_2       DDRC
25 #define DDR_DIP_345     DDRD
26
27 #define PIN_DIP_1       PINA
28 #define PIN_DIP_2       PINC
29 #define PIN_DIP_345     PIND
30
31 #define IOPIN_0             PINC2
32 #define IOPIN_1             PINC3
33 #define IOPIN_2             PINA1
34 #define IOPIN_3             PINC1
35 #define IOPIN_4             PINC0
36 #define IOPIN_5             PINA0
37 #define IOPIN_6             PINB4
38 #define IOPIN_7             PINB3
39 #define IOPIN_8             PIND6
40 #define IOPIN_9             PIND5
41 #define IOPIN_10            PINB7
42 #define IOPIN_11            PINB6
43 #define IOPIN_12            PINA3
44 #define IOPIN_13            PINA2
45 #define IOPIN_14            PIND4
46 #define IOPIN_15            PIND3
47 #define IOPIN_16            PINB2
48 #define IOPIN_17            PINB1
49 #define IOPIN_18            PINB0
50
51 #define OCRPIN_16       OCR1B
52 #define COM1PIN_16          COM1B1
53 #define COM0PIN_16          COM1B0
54
55 #define OCRPIN_17       OCR1A
56 #define COM1PIN_17          COM1A1
57 #define COM0PIN_17          COM1A0
58
59 #define IOPORT_251213     PORTA
60 #define IOPORT_671011161718  PORTB
61 #define IOPORT_0134         PORTC
62 #define IOPORT_891415     PORTD
63
64 #define IOPIN_251213        PINA
65 #define IOPIN_671011161718    PINB
66 #define IOPIN_0134          PINC
67 #define IOPIN_891415        PIND
```

```
68
69 #define IODDR_251213      DDRA
70 #define IODDR_671011161718   DDRB
71 #define IODDR_0134        DDRC
72 #define IODDR_891415      DDRD
73
74 #define PCMSK_DIP_1    PCMSK3
75 #define PCMSK_DIP_2    PCMSK1
76 #define PCMSK_DIP_34   PCMSK2
77
78 /* note: DIP5 is connected to INT0 */
79
80 #define PCINT_DIP1     PCINT24
81 #define PCINT_DIP2     PCINT15
82 #define PCINT_DIP3     PCINT16
83 #define PCINT_DIP4     PCINT17
84 //#define PCINT_DIP5     PCINT12
85
86 #define VERSION_CODE       0x01  /* Define a version code */
87
88 #define CTRINO_MAGIC_PIN          0x00
89
90 #define CPCAPR       0x00 /* c-trino Pin Capabilities Register */
91 #define CDGOA      0 /* Output Available */
92 #define CDGIA      1 /* Digital Input Available */
93 #define CADCA      2 /* ADC Available */
94 #define CPWMA      3 /* PWM Available */
95 #define CPUPA      4 /* Pull-up Available */
96 #define TRSTA      5 /* Tri-state Available */
97
98 #define CPSTR      0x01 /* c-trino Pin State Register */
99 #define CPDIR      0 /* Pin Direction */
100 #define CPINV      1 /* Pin Input Value (PINx equivalent) */
101 #define CPOUV      2 /* Pin Output Value (Portx equivalent) */
102 #define CPWME        3 /* PWM Enable */
103
104 #define CPWMVR       0x02 /* c-trino PWM Values Register */
105 #define CADCRH       0x03 /* c-trino ADC Reading Refister (High byte) */
106 #define CADCRL       0x04 /* c-trino ADC Reading Refister (Low byte) */
107 #define CBSPR      0x06 /* c-trino Board Specific Register */
108 #define CADCREF      0x07  /* TODO */
109 #define CFVER      0x08  /* TODO */
110 #define CPMSTR       0x09  /* TODO */
111
112 /* Refers to Error Byte */
113 #define CIPIN      1 /* Invalid Pin Number */
114 #define CIRWOP       2 /* Invalid I/O Operation */
115 #define CIREG      3 /* Invalid register */
116 #define CIIDL      4 /* Invalid Input Data Length */
```

```
117
118 /* The CPU clock frequency */
119 #ifndef F_CPU
120 #define F_CPU 8000000UL
121 #endif
122
123 #endif /* CTRINO_H_ */
```

A.2.5 io.c

```
1
2 #include <stdint.h>
3
4 #include <avr/interrupt.h>
5
6 #include "board.h"
7 #include "ctrino.h"
8 #include "TWI_slave.h"
9
10 int io_init(void)
11 {
12   /* clear both Pin Change Mask registers */
13   PCMSK1 = 0x00;
14   PCMSK2 = 0x00;
15   PCMSK3 = 0x00;
16
17   DDRB=(1<<PB1)|(1<<PB2);
18   IODDR_251213=(1<<IOPIN_13)|(1<<IOPIN_14);
19   IODDR_671011161718=(1<<IOPIN_16)|(1<<IOPIN_18)|(1<<IOPIN_11);
20
21
22   /* dip switch related stuff */
23   PORT_DIP_345 |= (1<<DIP5)|(1<<DIP4)|(1<<DIP3); /* activate pull-ups */
24   PORT_DIP_2 |= (1<<DIP2); /* activate pull-ups */
25   PORT_DIP_1 |= (1<<DIP1); /* activate pull-ups */
26
27   EICRA |= (0<<ISC01)|(1<<ISC00);
28   EIMSK |= (1<<INT0); /* DIP5 is connected to INT0, any logical change causes an interrupt */
29   PCMSK_DIP_34 |= (1<<PCINT_DIP4)|(1<<PCINT_DIP3); /* any logical change also causes interrupt
       */
30   PCMSK_DIP_2 |= (1<<PCINT_DIP2); /* same for DIP2 */
31   PCMSK_DIP_1 |= (1<<PCINT_DIP1); /* same for DIP1 */
32   PCICR |= (1<<PCIE3)|(1<<PCIE2)|(1<<PCIE1); /* enables interrupts for the 5 pins of DIP
       switch */
33
34   return 0;
35 }
36
37 void i2c_data_handler(const uint8_t input_buffer_length,
38     const uint8_t *input_buffer,
```

```
39      uint8_t *output_buffer_length,
40      uint8_t *output_buffer)
41  {
42    /* input data */
43    uint8_t in_pin; /* byte 0 */
44    uint8_t in_reg; /* byte 1 */
45    /* output data */
46    uint8_t out_error = 0; /* byte 0 */
47    if (input_buffer_length < 2) {
48      out_error = CIIDL;
49      goto out1;
50    }
51
52    in_pin = input_buffer[0];
53    if (in_pin > NPINS) {
54      out_error = CIPIN;
55      goto out1;
56    }
57
58    in_reg = input_buffer[1];
59    out_error = (*ctrino_msghandlers[in_reg])(in_pin,
60        input_buffer_length - 2,
61        &input_buffer[2],
62        &output_buffer[1],
63        output_buffer_length);
64
65    out1:
66    output_buffer[0] = out_error;
67    *output_buffer_length+=1;
68
69    return;
70  }
71
72  uint8_t get_pin_value(uint8_t pin){
73    uint8_t ret=0;
74    switch(pin){
75    case 0:ret=((IOPIN_0134 & (1 << IOPIN_0)) >> IOPIN_0);break;
76    case 1:ret=((IOPIN_0134 & (1 << IOPIN_1)) >> IOPIN_1);break;
77    case 2:ret=((IOPIN_251213 & (1 << IOPIN_2)) >> IOPIN_2);break;
78    case 3:ret=((IOPIN_0134 & (1 << IOPIN_3)) >> IOPIN_3);break;
79    case 4:ret=((IOPIN_0134 & (1 << IOPIN_4)) >> IOPIN_4);break;
80    case 5:ret=((IOPIN_251213 & (1 << IOPIN_5)) >> IOPIN_5);break;
81    case 6:ret=((IOPIN_671011161718 & (1 << IOPIN_6)) >> IOPIN_6);break;
82    case 7:ret=((IOPIN_671011161718 & (1 << IOPIN_7)) >> IOPIN_7);break;
83    case 8:ret=((IOPIN_891415 & (1 << IOPIN_8)) >> IOPIN_8);break;
84    case 9:ret=((IOPIN_891415 & (1 << IOPIN_9)) >> IOPIN_9);break;
85    case 10:ret=((IOPIN_671011161718 & (1 << IOPIN_10)) >> IOPIN_10);break;
86    case 11:ret=((IOPIN_671011161718 & (1 << IOPIN_11)) >> IOPIN_11);break;
87    case 12:ret=((IOPIN_251213 & (1 << IOPIN_12)) >> IOPIN_12);break;
```

```
  case 13:ret=((IOPIN_251213 & (1 << IOPIN_13)) >> IOPIN_13);break;
  case 14:ret=((IOPIN_891415 & (1 << IOPIN_14)) >> IOPIN_14);break;
  case 15:ret=((IOPIN_891415 & (1 << IOPIN_15)) >> IOPIN_15);break;
  case 16:ret=((IOPIN_671011161718 & (1 << IOPIN_16)) >> IOPIN_16);break;
  case 17:ret=((IOPIN_671011161718 & (1 << IOPIN_17)) >> IOPIN_17);break;
  case 18:ret=((IOPIN_671011161718 & (1 << IOPIN_18)) >> IOPIN_18);break;
  }
  return ret;
}



void set_pin_pin(uint8_t pin, const uint8_t value){
  if(value){
    switch(pin){
    case 0:IOPIN_0134|=1<<IOPIN_0;break;
    case 1:IOPIN_0134|=1<<IOPIN_1;break;
    case 2:IOPIN_251213|=1<<IOPIN_2;break;
    case 3:IOPIN_0134|=1<<IOPIN_3;break;
    case 4:IOPIN_0134|=1<<IOPIN_4;break;
    case 5:IOPIN_251213|=1<<IOPIN_5;break;
    case 6:IOPIN_671011161718|=1<<IOPIN_6;break;
    case 7:IOPIN_671011161718|=1<<IOPIN_7;break;
    case 8:IOPIN_891415|=1<<IOPIN_8;break;
    case 9:IOPIN_891415|=1<<IOPIN_9;break;
    case 10:IOPIN_671011161718|=1<<IOPIN_10;break;
    case 11:IOPIN_671011161718|=1<<IOPIN_11;break;
    case 12:IOPIN_251213|=1<<IOPIN_12;break;
    case 13:IOPIN_251213|=1<<IOPIN_13;break;
    case 14:IOPIN_891415|=1<<IOPIN_14;break;
    case 15:IOPIN_891415|=1<<IOPIN_15;break;
    case 16:IOPIN_671011161718|=1<<IOPIN_16;break;
    case 17:IOPIN_671011161718|=1<<IOPIN_17;break;
    case 18:IOPIN_671011161718|=1<<IOPIN_18;break;
    }
  }
  else
  {
    switch(pin){
    case 0:IOPIN_0134 &= ~(1<<IOPIN_0);break;
    case 1:IOPIN_0134 &= ~(1<<IOPIN_1);break;
    case 2:IOPIN_251213 &= ~(1<<IOPIN_2);break;
    case 3:IOPIN_0134 &= ~(1<<IOPIN_3);break;
    case 4:IOPIN_0134 &= ~(1<<IOPIN_4);break;
    case 5:IOPIN_251213 &= ~(1<<IOPIN_5);break;
    case 6:IOPIN_671011161718 &= ~(1<<IOPIN_6);break;
    case 7:IOPIN_671011161718 &= ~(1<<IOPIN_7);break;
    case 8:IOPIN_891415 &= ~(1<<IOPIN_8);break;
    case 9:IOPIN_891415 &= ~(1<<IOPIN_9);break;
```

```
    case 10:IOPIN_671011161718 &= ~(1<<IOPIN_10);break;
    case 11:IOPIN_671011161718 &= ~(1<<IOPIN_11);break;
    case 12:IOPIN_251213 &= ~(1<<IOPIN_12);break;
    case 13:IOPIN_251213 &= ~(1<<IOPIN_13);break;
    case 14:IOPIN_891415 &= ~(1<<IOPIN_14);break;
    case 15:IOPIN_891415 &= ~(1<<IOPIN_15);break;
    case 16:IOPIN_671011161718 &= ~(1<<IOPIN_16);break;
    case 17:IOPIN_671011161718 &= ~(1<<IOPIN_17);break;
    case 18:IOPIN_671011161718 &= ~(1<<IOPIN_18);break;
    }
  }
}

void set_pin_port(uint8_t pin, const uint8_t value){
  if(value){
    switch(pin){
    case 0:IOPORT_0134|=1<<IOPIN_0;break;
    case 1:IOPORT_0134|=1<<IOPIN_1;break;
    case 2:IOPORT_251213|=1<<IOPIN_2;break;
    case 3:IOPORT_0134|=1<<IOPIN_3;break;
    case 4:IOPORT_0134|=1<<IOPIN_4;break;
    case 5:IOPORT_251213|=1<<IOPIN_5;break;
    case 6:IOPORT_671011161718|=1<<IOPIN_6;break;
    case 7:IOPORT_671011161718|=1<<IOPIN_7;break;
    case 8:IOPORT_891415|=1<<IOPIN_8;break;
    case 9:IOPORT_891415|=1<<IOPIN_9;break;
    case 10:IOPORT_671011161718|=1<<IOPIN_10;break;
    case 11:IOPORT_671011161718|=1<<IOPIN_11;break;
    case 12:IOPORT_251213|=1<<IOPIN_12;break;
    case 13:IOPORT_251213|=1<<IOPIN_13;break;
    case 14:IOPORT_891415|=1<<IOPIN_14;break;
    case 15:IOPORT_891415|=1<<IOPIN_15;break;
    case 16:IOPORT_671011161718|=1<<IOPIN_16;break;
    case 17:IOPORT_671011161718|=1<<IOPIN_17;break;
    case 18:IOPORT_671011161718|=1<<IOPIN_18;break;
    }
  }
  else
  {
    switch(pin){
    case 0:IOPORT_0134 &= ~(1<<IOPIN_0);break;
    case 1:IOPORT_0134 &= ~(1<<IOPIN_1);break;
    case 2:IOPORT_251213 &= ~(1<<IOPIN_2);break;
    case 3:IOPORT_0134 &= ~(1<<IOPIN_3);break;
    case 4:IOPORT_0134 &= ~(1<<IOPIN_4);break;
    case 5:IOPORT_251213 &= ~(1<<IOPIN_5);break;
    case 6:IOPORT_671011161718 &= ~(1<<IOPIN_6);break;
    case 7:IOPORT_671011161718 &= ~(1<<IOPIN_7);break;
    case 8:IOPORT_891415 &= ~(1<<IOPIN_8);break;
```

```
    case 9:IOPORT_891415 &= ~(1<<IOPIN_9);break;
    case 10:IOPORT_671011161718 &= ~(1<<IOPIN_10);break;
    case 11:IOPORT_671011161718 &= ~(1<<IOPIN_11);break;
    case 12:IOPORT_251213 &= ~(1<<IOPIN_12);break;
    case 13:IOPORT_251213 &= ~(1<<IOPIN_13);break;
    case 14:IOPORT_891415 &= ~(1<<IOPIN_14);break;
    case 15:IOPORT_891415 &= ~(1<<IOPIN_15);break;
    case 16:IOPORT_671011161718 &= ~(1<<IOPIN_16);break;
    case 17:IOPORT_671011161718 &= ~(1<<IOPIN_17);break;
    case 18:IOPORT_671011161718 &= ~(1<<IOPIN_18);break;
    }
  }
}

void set_pin_ddr(uint8_t pin, const uint8_t value){
  if(value){
    switch(pin){
    case 0:IODDR_0134|=1<<IOPIN_0;break;
    case 1:IODDR_0134|=1<<IOPIN_1;break;
    case 2:IODDR_251213|=1<<IOPIN_2;break;
    case 3:IODDR_0134|=1<<IOPIN_3;break;
    case 4:IODDR_0134|=1<<IOPIN_4;break;
    case 5:IODDR_251213|=1<<IOPIN_5;break;
    case 6:IODDR_671011161718|=1<<IOPIN_6;break;
    case 7:IODDR_671011161718|=1<<IOPIN_7;break;
    case 8:IODDR_891415|=1<<IOPIN_8;break;
    case 9:IODDR_891415|=1<<IOPIN_9;break;
    case 10:IODDR_671011161718|=1<<IOPIN_10;break;
    case 11:IODDR_671011161718|=1<<IOPIN_11;break;
    case 12:IODDR_251213|=1<<IOPIN_12;break;
    case 13:IODDR_251213|=1<<IOPIN_13;break;
    case 14:IODDR_891415|=1<<IOPIN_14;break;
    case 15:IODDR_891415|=1<<IOPIN_15;break;
    case 16:IODDR_671011161718|=1<<IOPIN_16;break;
    case 17:IODDR_671011161718|=1<<IOPIN_17;break;
    case 18:IODDR_671011161718|=1<<IOPIN_18;break;
    }
  }
  else
  {
    switch(pin){
    case 0:IODDR_0134 &= ~(1<<IOPIN_0);break;
    case 1:IODDR_0134 &= ~(1<<IOPIN_1);break;
    case 2:IODDR_251213 &= ~(1<<IOPIN_2);break;
    case 3:IODDR_0134 &= ~(1<<IOPIN_3);break;
    case 4:IODDR_0134 &= ~(1<<IOPIN_4);break;
    case 5:IODDR_251213 &= ~(1<<IOPIN_5);break;
    case 6:IODDR_671011161718 &= ~(1<<IOPIN_6);break;
    case 7:IODDR_671011161718 &= ~(1<<IOPIN_7);break;
```

```
235     case 8:IODDR_891415 &= ~(1<<IOPIN_8);break;
236     case 9:IODDR_891415 &= ~(1<<IOPIN_9);break;
237     case 10:IODDR_671011161718 &= ~(1<<IOPIN_10);break;
238     case 11:IODDR_671011161718 &= ~(1<<IOPIN_11);break;
239     case 12:IODDR_251213 &= ~(1<<IOPIN_12);break;
240     case 13:IODDR_251213 &= ~(1<<IOPIN_13);break;
241     case 14:IODDR_891415 &= ~(1<<IOPIN_14);break;
242     case 15:IODDR_891415 &= ~(1<<IOPIN_15);break;
243     case 16:IODDR_671011161718 &= ~(1<<IOPIN_16);break;
244     case 17:IODDR_671011161718 &= ~(1<<IOPIN_17);break;
245     case 18:IODDR_671011161718 &= ~(1<<IOPIN_18);break;
246     }
247   }
248 }
249
250 uint8_t get_pin_adc_channel(uint8_t pin){
251   switch(pin){
252   case 0:return 2;
253   case 1:return 3;
254   case 2:return 7;
255   case 3:return 1;
256   case 4:return 0;
257   case 5:return 6;
258   }
259   return -1;
260 }
261
262 /* Get the I2C address of the board from the dip switch. */
263 int dipsw_value_get(uint8_t *out)
264 {
265   uint8_t value = 0;
266
267   const uint8_t pin_dip_1 = PIN_DIP_1;
268   const uint8_t pin_dip_2 = PIN_DIP_2;
269   const uint8_t pin_dip_345 = PIN_DIP_345;
270
271   if (! (pin_dip_345 & (1 << DIP5))) {
272     value |= 0x01;
273   }
274   if (! (pin_dip_345 & (1 << DIP4))) {
275     value |= 0x02;
276   }
277   if (! (pin_dip_345 & (1 << DIP3))) {
278     value |= 0x04;
279   }
280   if (! (pin_dip_2 & (1 << DIP2))) {
281     value |= 0x08;
282   }
283   if (! (pin_dip_1 & (1 << DIP1))) {
```

```
284     value |= 0x10;
285   }
286
287   *out = value;
288
289   return 0;
290 }
291
292 /* Dip switch was touched... */
293 ISR(PCINT1_vect)
294 {
295   uint8_t twi_address;
296
297   //  led_mode_idle();
298   if (! dipsw_value_get(&twi_address)) {
299     TWI_Slave_Initialise( (unsigned char)((twi_address<<TWI_ADR_BITS) | (TRUE<<TWI_GEN_BIT) ))
      ;
300     TWI_Start_Transceiver( );
301   }
302 }
303 ISR( INT0_vect, ISR_ALIASOF(PCINT1_vect) );
304 ISR( PCINT2_vect, ISR_ALIASOF(PCINT1_vect) );
```

A.2.6 io.h

```
1 #ifndef IO_H_
2 #define IO_H_
3
4
5 #include <stdint.h>
6
7 int io_init(void);
8 void i2c_data_handler(uint8_t input_buffer_length, const uint8_t *input_buffer,
9                       uint8_t *output_buffer_length, uint8_t *output_buffer);
10 uint8_t get_pin_value(uint8_t pin);
11 void set_pin_pin(uint8_t pin, const uint8_t value);
12 void set_pin_port(uint8_t pin, const uint8_t value);
13 void set_pin_port(uint8_t pin, const uint8_t value);
14 void set_pin_ddr(uint8_t pin, const uint8_t value);
15 uint8_t get_pin_adc_channel(uint8_t pin);
16 int dipsw_value_get(uint8_t *out);
17
18 #endif /* IO_H_ */
```

A.2.7 adc.c

```
1
2
3 #include <avr/interrupt.h>
4 #include <avr/io.h>
```

```
5  #include <avr/sleep.h>
6  #include <stdint.h>
7
8  int adc_init( void )
9  {
10   uint8_t adcsra = 0;
11   uint8_t admux = 0;
12
13   adcsra |= 1 << ADEN; /* Enable the ADC */
14   adcsra |= 1 << ADSC; /* Start the first conversion... */
15   adcsra |= (1<<ADPS1)|(1<<ADPS0); /* Set clock to FCPU/128 (prescaler 128) */
16   ADCSRA = adcsra;
17
18   ADMUX  |= (1<<REFS0); /* Voltage ref = AVCC (5V) */
19
20   ADCSRA |= (1<<ADSC); /* start the fisrt conversion */
21   while (ADCSRA & (1 << ADSC)); /* ...wait until it first conversion completes */
22
23   return 0;
24 }
25
26 int adc_set_channel (const uint8_t channel )
27 {
28   int ret = 0;
29   uint8_t admux = 0;
30
31   if (channel >= 6) {
32     ret = -1;
33     goto out;
34   }
35   admux = ADMUX;
36   admux &= 0x60; /* clear channel bits on ADMUX */
37   admux |= channel;
38   ADMUX=admux;
39   out:
40   return ret;
41 }
42
43 int adc_convert(uint16_t *const out)
44 {
45   ADCSRA |= 1 << ADSC; /* Start the conversion... */
46   while (ADCSRA & (1 << ADSC)); /* ...and wait for the conversion to complete */
47
48   *out = (uint16_t)ADC;
49
50   return 0;
51 }
```

A.2.8 adc.h

```c
#ifndef ADC_H_
#define ADC_H_

#include <stdint.h>

int adc_init(void);
int adc_set_channel(uint8_t channel);
int adc_convert(uint16_t *out);

#endif /* ADC_H_ */
```

Appendix B

Controller

B.1 Photos

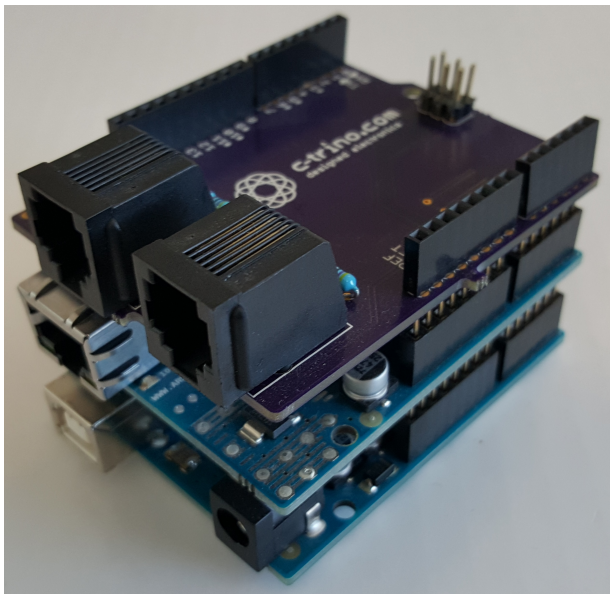


Figure B.1: Controller Module

B.2 Firmware

B.2.1 controller.ino

```
#include <SPI.h>
#include <Ethernet.h>
#include <Wire.h>
#include "Ctrino.h"
#include "app.h"

#define BUF_LEN 15
#define MAX_MSG 3
```

```
9
10 #define PIN_MODE 0
11 #define DIGITAL_READ 1
12 #define ANALOG_READ 2
13 #define DIGITAL_WRITE 3
14 #define ANALOG_WRITE 4
15 #define GET_VERSION 5
16
17 /* c-trino object */
18 Ctrino board;
19 int address=0;
20
21 char buffer[BUF_LEN];
22
23 int executeFunction(int*);
24 int extractCommand(int*, char*, int);
25
26 //Network
27 IPAddress serverIP(192,168,1,111); // IP Adress to our Server
28 int serverPort=993;
29 EthernetClient client;
30 byte mac[] = {
31     0x90, 0xA2, 0xDA, 0x0F, 0x4B, 0xFB };
32
33 void setupEthernetComm();
34 int msg;
35
36 void setup() {
37         Serial.begin(115200);
38         Wire.begin();
39   setupEthernetComm();
40
41 //   displayHeader();
42 //   enterMenu();
43 }
44
45 void loop()
46 {
47         int msg_len=0;
48         int error=0;
49         int value=0;
50         char buffer[BUF_LEN];
51         int cmd[4];
52
53         msg_len=client.available();
54         if(msg_len){
55           for(int i=0;i<msg_len;i++){
56               buffer[i]=client.read();
57               Serial.print(buffer[i]);
```

```
            if(buffer[i]=='S'){
              msg_len=9;
              break;
            }
          }
        }
        if(extractCommand(cmd, buffer, msg_len)){
           value=executeFunction(cmd,error);
           if(error==NO_BOARD || error==CMD_OK)
             client.println(String(error));
           else if(error=CMD_OK_ANS)
             client.println(String(String(error)+"."+String(value)));
           else if(error)
             client.println(String(MISC_ERROR));

           cmd[0]=0;cmd[1]=0;cmd[2]=0;cmd[3]=0;value=0;
         }

  // if the server's disconnected, stop the client:
  if (!client.connected()) {
    Serial.println();//report it to the serial
    Serial.println("disconnected");
    client.stop();

    // do nothing forevermore:
    for(;;){
                        // if you get a connection to the Server
                if (client.connect(serverIP, serverPort)) {
                  Serial.println("Reconnected");//report it to the Serial
                              break;
                }
                      delay(20);
               }
  }
  delay(30);
}

void setupEthernetComm(){
  // start the Ethernet connection:
  if (Ethernet.begin(mac) == 0) {
    Serial.println("Failed to configure Ethernet using DHCP");
    // no point in carrying on, so do nothing forevermore:
    for(;;)
      ;
  }
  // print your local IP address:
  Serial.print("My IP address: ");
  for (byte thisByte = 0; thisByte < 4; thisByte++) {
    // print the value of each byte of the IP address:
```

```
    Serial.print(Ethernet.localIP()[thisByte], DEC);
    Serial.print(".");
  }
  Serial.println();

  // give the Ethernet shield a second to initialize:
  delay(1000);

  Serial.println("connecting to Server ...");

  // if you get a connection to the Server
  if (client.connect(serverIP, serverPort)) {
    Serial.println("connected");//report it to the Serial
  }
  else {
    // if you didn't get a connection to the server:
    Serial.println("connection failed");
    // do nothing forevermore:
    for(;;)
      ;
  }
}

int executeFunction(int* cmd,int& error)
{       //cmd= {function,pin,address,value}
        int val=0; uint8_t version_code=0;
        error=0;
        if(cmd[2]==0){
          switch (cmd[0]) {
              case DIGITAL_READ:
                val=digitalRead(cmd[1]);
                error=CMD_OK_ANS;
                return val;
              case ANALOG_READ:
                val=round(analogRead(cmd[1])*0.249); //10bit to 8bit resolution
                error=CMD_OK_ANS;
                return val;
              case DIGITAL_WRITE:
                digitalWrite(cmd[1],cmd[3]);
                error=CMD_OK;
                return 0;
              case ANALOG_WRITE:
                analogWrite(cmd[1],cmd[3]);
                error=CMD_OK;
                return 0;
              case PIN_MODE:
                pinMode(cmd[1],cmd[3]);
                error=CMD_OK;
                return 0;
```

```
156           }
157         }
158         else if(cmd[2]>0){
159           board.begin(cmd[2]);
160           if(!board.isAvailable(version_code)){
161             error=NO_BOARD;
162             return error;
163           }
164           switch (cmd[0]) {
165               case DIGITAL_READ:
166                 val=board.digitalRead(cmd[1]);
167                 error=CMD_OK_ANS;
168                 return val;
169               case ANALOG_READ:
170                 val=round(board.analogRead(cmd[1])*0.249); //10bit to 8bit resolution
171                 error=CMD_OK_ANS;
172                 return val;
173               case DIGITAL_WRITE:
174                 board.digitalWrite(cmd[1],cmd[3],error);
175                 error=CMD_OK;
176                 return 0;
177               case ANALOG_WRITE:
178                 board.analogWrite(cmd[1],cmd[3],error);
179                 error=CMD_OK;
180                 return 0;
181               case PIN_MODE:
182                 board.pinMode(cmd[1],cmd[3]);
183                 error=CMD_OK;
184                 return 0;
185               case GET_VERSION:
186                 error=CMD_OK_ANS;
187                 return version_code;
188
189           }
190         }
191
192 }
193
194  int ADCtoMilivolts(int ADCval)
195  {
196    int x, y;
197
198    x = ((long)ADCval * 5000) / 1024; /* converts from 0-1023 to milivolts (0-5000) */
199
200    y = x % 10;
201    x /= 10;
202    x *= 10;
203
204    /* rounds to the nearest decade */
```

```
205     if (y >= 5)
206     {
207       return x + 10;
208     }
209     else
210     {
211       return x;
212     }
213   }
214
215 int extractCommand(int* cmd, char* buf, int length){
216       int i=0;
217       int j=0;
218       if(length!=9)
219         return 0;
220       buffer[9]='\0';
221       for(i=3,j=6;i>=0;i--,j=j-2){
222         cmd[i]=atoi(buf+j);
223         buf[j]='\0';
224       }
225       return 1;
226 }
```

B.2.2 Ctrino.cpp

```
1  #include <Wire.h>
2
3  #include "Ctrino.h"
4
5  Ctrino::Ctrino()
6  {
7      /* do nothing! */
8  }
9
10 void Ctrino::begin(uint8_t address)
11 {
12     _address = address;
13 }
14
15 void Ctrino::begin(int address)
16 {
17     begin((uint8_t)address);
18 }
19
20 bool Ctrino::isAvailable(uint8_t& version_code)
21 {
22   bool ret = true;
23   uint8_t in_buf[2];
24
25   if (doRead(CTRINO_MAGIC_PIN, CFVER, in_buf, 2)) {
```

```
26      ret = false;
27      goto out;
28    }
29
30    if (in_buf[0] != _address) {
31      ret = false;
32      goto out;
33    }
34          version_code=in_buf[1];
35
36   out:
37      return ret;
38  }
39
40  void Ctrino::pinMode(uint8_t pin, uint8_t mode)
41  {
42      uint8_t const out_buf[] = { mode << CPDIR };
43
44      doWrite(pin, CPMSTR, out_buf, 1);
45  }
46
47  void Ctrino::digitalWrite(uint8_t pin, uint8_t value)
48  {
49      int temp;
50
51      digitalWrite(pin, value, temp);
52  }
53
54  void Ctrino::digitalWrite(uint8_t pin, uint8_t value, int& error)
55  {
56      uint8_t const out_buf[] = { value << CPOUV };
57
58      error = (int)doWrite(pin, CPSTR, out_buf, 1);
59  }
60
61  int Ctrino::digitalRead(uint8_t pin)
62  {
63      int temp;
64
65      return digitalRead(pin, temp);
66  }
67
68  int Ctrino::digitalRead(uint8_t pin, int& error)
69  {
70      uint8_t in_buf[1];
71
72      error = doRead(pin, CPSTR, in_buf, 1);
73
74      return (int)(in_buf[0] >> CPINV);
```

```
75 }
76
77 void Ctrino::analogWrite(uint8_t pin, int value)
78 {
79     int temp;
80
81     analogWrite(pin, value, temp);
82 }
83
84 void Ctrino::analogWrite(uint8_t pin, int value, int& error)
85 {
86     uint8_t const out_buf[] = { value };
87
88     error = doWrite(pin, CPWMMR, out_buf, 1);
89 }
90
91 int Ctrino::analogRead(uint8_t pin)
92 {
93     int temp;
94
95     return analogRead(pin, temp);
96 }
97
98 int Ctrino::analogRead(uint8_t pin, int& error)
99 {
100     uint8_t in_buf[2];
101
102     error = doRead(pin, CADCRH, in_buf, 2);
103
104     return (int)(in_buf[0] << 8 | in_buf[1]);
105 }
106
107 void Ctrino::analogReference(uint8_t mode)
108 {
109     uint8_t const out_buf[] = { mode };
110
111     doWrite(CTRINO_MAGIC_PIN, CADCREF, out_buf, 1) ;
112 }
113
114 uint8_t Ctrino::doRead(uint8_t pin, uint8_t reg, uint8_t in_buf[],
115                        size_t len)
116 {
117     uint8_t ret;
118     size_t i;
119
120     Wire.beginTransmission(_address);
121     Wire.write(pin);
122     Wire.write(reg << 1 | CTRINO_READ);
123     ret = Wire.endTransmission(false /* don't send STOP */);
```

```
124     if (ret) {
125         ret = CTRINO_MISC_ERROR;
126         goto out;
127     }
128 
129     ret = Wire.requestFrom(_address, (uint8_t)(len + 1), (uint8_t)true /* send STOP */);
130     if (ret != len + 1) {
131         ret = CTRINO_MISC_ERROR;
132         goto out;
133     }
134 
135     ret = Wire.read(); /* the error byte */
136     if (ret)
137         goto out;
138 
139     for (i = 0; Wire.available() && i < len; i++)
140         in_buf[i] = Wire.read();
141 
142 out:
143     return ret;
144 }
145 
146 uint8_t Ctrino::doWrite(uint8_t pin, uint8_t reg, uint8_t const out_buf[],
147                         size_t len)
148 {
149     uint8_t ret;
150 
151     Wire.beginTransmission(_address);
152     Wire.write(pin);
153     Wire.write(reg << 1 | CTRINO_WRITE);
154     Wire.write(out_buf, len);
155     ret = Wire.endTransmission(false /* don't send STOP */);
156     if (ret) {
157         ret = CTRINO_MISC_ERROR;
158         goto out;
159     }
160 
161     ret = Wire.requestFrom(_address, (uint8_t)1, (uint8_t)true /* send STOP */);
162     if (ret != 1) {
163         ret = CTRINO_MISC_ERROR;
164         goto out;
165     }
166 
167     ret = Wire.read(); /* the error byte */
168 
169 out:
170     return ret;
171 }
```

B.2.3 Ctrino.h

```
#ifndef CTRINO_H_
#define CTRINO_H_

#define CTRINO_MAGIC_PIN          0x00
#define CTRINO_MISC_ERROR         0x01

#define CTRINO_WRITE   0x00
#define CTRINO_READ    0x01

#define CPCAPR       0x00  /* c-trino Pin Capabilities Register */
#define CDGOA      0      /* Output Available */
#define CDGIA      1      /* Digital Input Available */
#define CADCA      2      /* ADC Available */
#define CPWMA      3      /* PWM Available */
#define CPUPA      4      /* Pull-up Available */
#define TRSTA      5      /* Tri-state Available */

#define CPSTR      0x01  /* c-trino Pin State Register */
#define CPDIR      0      /* Pin Direction */
#define CPINV      1      /* Pin Input Value (PINx equivalent) */
#define CPOUV      2      /* Pin Output Value (Portx equivalent) */
#define CPWME        3      /* PWM Enable */

#define CPWMVR       0x02  /* c-trino PWM Values Register */
#define CADCRH       0x03  /* c-trino ADC Reading Refister (High byte) */
#define CADCRL       0x04  /* c-trino ADC Reading Refister (Low byte) */
#define CBSPR      0x06  /* c-trino Board Specific Register */
#define CADCREF      0x07  /* TODO */
#define CFVER      0x08  /* TODO */
#define CPMSTR       0x09  /* TODO */

/* Refers to Error Byte */
#define CIPIN      1      /* Invalid Pin Number */
#define CIRWOP       2      /* Invalid I/O Operation */
#define CIREG      3      /* Invalid register */
#define CIIDL      4      /* Invalid Input Data Length */

#ifndef CTRINO_FIRMWARE
class Ctrino
{
 private:
    uint8_t _address;

    uint8_t doRead(uint8_t pin, uint8_t reg, uint8_t in_buf[],
                   size_t len);
    uint8_t doWrite(uint8_t pin, uint8_t reg, uint8_t const out_buf[],
                    size_t len);
```

```
48 public:
49     Ctrino();
50
51     void begin(int address);
52     void begin(uint8_t address);
53     inline uint8_t getAddress() { return _address; }
54     bool isAvailable(uint8_t& version_code);
55
56     void pinMode(uint8_t pin, uint8_t mode);
57     void digitalWrite(uint8_t pin, uint8_t value);
58     void digitalWrite(uint8_t pin, uint8_t value, int& error);
59     int digitalRead(uint8_t pin);
60     int digitalRead(uint8_t pin, int& error);
61     void analogWrite(uint8_t pin, int value);
62     void analogWrite(uint8_t pin, int value, int& error);
63     int analogRead(uint8_t pin);
64     int analogRead(uint8_t pin, int& error);
65     void analogReference(uint8_t mode);
66
67     String getCtrinoModel();
68     uint8_t getCtrinoVersion();
69
70     /*
71       TODO:
72       ** query pin functionality
73      */
74 };
75
76 inline void digitalWrite(Ctrino ctrino, uint8_t pin, uint8_t value)
77 {
78     ctrino.digitalWrite(pin, value);
79 }
80
81 inline void digitalWrite(Ctrino ctrino, uint8_t pin, uint8_t value, int& error)
82 {
83     ctrino.digitalWrite(pin, value, error);
84 }
85
86 inline int digitalRead(Ctrino ctrino, uint8_t pin)
87 {
88     return ctrino.digitalRead(pin);
89 }
90
91 inline int digitalRead(Ctrino ctrino, uint8_t pin, int& error)
92 {
93     return ctrino.digitalRead(pin, error);
94 }
95
96 inline void analogWrite(Ctrino ctrino, uint8_t pin, int value)
```

```
97  {
98      ctrino.analogWrite(pin, value);
99  }
100
101 inline void analogWrite(Ctrino ctrino, uint8_t pin, int value, int& error)
102 {
103     ctrino.analogWrite(pin, value, error);
104 }
105
106 inline int analogRead(Ctrino ctrino, uint8_t pin)
107 {
108     return ctrino.analogRead(pin);
109 }
110
111 inline int analogRead(Ctrino ctrino, uint8_t pin, int& error)
112 {
113     return ctrino.analogRead(pin, error);
114 }
115 #endif // CTRINO_FIRMWARE
116
117 #endif // CTRINO_H_
```

Appendix C

Control Server

C.1 Source Code

C.1.1 App.java

```
1  package domoServer;
2
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6  import java.io.IOException;
7  import java.util.HashSet;
8  import java.util.Set;
9
10 import javax.swing.BorderFactory;
11 import javax.swing.Box;
12 import javax.swing.BoxLayout;
13 import javax.swing.DefaultListModel;
14 import javax.swing.JButton;
15 import javax.swing.JComponent;
16 import javax.swing.JFrame;
17 import javax.swing.JList;
18 import javax.swing.JPanel;
19 import javax.swing.JScrollPane;
20 import javax.swing.JSeparator;
21 import javax.swing.JTextField;
22 import javax.swing.ListSelectionModel;
23 import javax.swing.SwingConstants;
24 import javax.swing.Timer;
25 import javax.swing.event.ListSelectionEvent;
26 import javax.swing.event.ListSelectionListener;
27
28 import data.Commands;
29 import data.Devices;
30 import devices.FieldDevice;
```

```
31
32 /* ListDemo.java requires no other files. */
33 public class App extends JPanel
34 implements ListSelectionListener {
35   /**
36    *
37    */
38   private static Arduino arduino=null;
39   private static final long serialVersionUID = 1L;
40   private static final int DELAY = 1000;
41   private static final String onString = "On";
42   private static final String offString = "Off";
43
44   private JList<String> list;
45   public DefaultListModel<String> listModel;
46
47   private JButton offButton;
48   private JButton onButton;
49
50   private JTextField stateValue;
51   private JTextField newValue;
52   private Timer pollTimer;
53   private Set<String> knownAddresses=new HashSet<String>();
54
55   public App() {
56     super(new BorderLayout());
57
58     listModel = new DefaultListModel<String>();
59     listModel.addElement("");
60
61     //Create the list and put it in a scroll pane.
62     list = new JList<String>(listModel);
63     list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
64     list.setSelectedIndex(0);
65     list.addListSelectionListener(this);
66     list.setVisibleRowCount(15);
67     JScrollPane listScrollPane = new JScrollPane(list);
68
69     onButton = new JButton(onString);
70     onButton.setActionCommand(onString);
71     onButton.addActionListener(new OnListener());
72
73     offButton = new JButton(offString);
74     offButton.setActionCommand(offString);
75     offButton.addActionListener(new OffListener());
76
77     stateValue = new JTextField(20);
78     newValue = new JTextField(20);
79
```

```
80      stateValue.setEditable(false);
81
82      //Create a panel that uses BoxLayout.
83      JPanel buttonPane = new JPanel();
84      buttonPane.setLayout(new BoxLayout(buttonPane,BoxLayout.LINE_AXIS));
85      buttonPane.add(offButton);
86      buttonPane.add(Box.createHorizontalStrut(5));
87      buttonPane.add(stateValue);
88      buttonPane.add(Box.createHorizontalStrut(5));
89      buttonPane.add(new JSeparator(SwingConstants.VERTICAL));
90      buttonPane.add(newValue);
91      buttonPane.add(Box.createHorizontalStrut(5));
92      buttonPane.add(onButton);
93      buttonPane.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
94
95      add(listScrollPane, BorderLayout.CENTER);
96      add(buttonPane, BorderLayout.PAGE_END);
97      startPolling();
98
99      listModel.clear();
100
101   }
102
103   private void startPolling(){
104     ActionListener pollPerformer = new ActionListener() {
105       public void actionPerformed(ActionEvent e){
106         if(arduino!=null){
107           Set<String> tmp=new HashSet<String>();
108           tmp.addAll(Devices.getAddressesandFunctions());
109           Devices.setChanged(false);
110           tmp.removeAll(knownAddresses);
111           for(String s : tmp)
112             listModel.addElement(s);
113           knownAddresses.addAll(tmp);
114         }
115         int index = list.getSelectedIndex();
116         if (index != -1) {
117           int address=Integer.parseInt( listModel.elementAt(index).substring(0, 2).trim());
118           FieldDevice device = Devices.getDevice(address);
119           stateValue.setText(Integer.toString(device.getState()));
120         }
121       }
122     };
123     pollTimer = new Timer(DELAY, pollPerformer); // "cannot find symbol"
124     pollTimer.start();                           // "cannot find symbol"
125   }
126
127   class OffListener implements ActionListener {
128     public void actionPerformed(ActionEvent e) {
```

```
129       //This method can be called only if
130       //there's a valid selection
131       //so go ahead and remove whatever's selected.
132       int index = list.getSelectedIndex();
133       int address=Integer.parseInt( listModel.elementAt(index).substring(0, 2).trim());
134
135       Commands.executeCommand(address,Commands.TOGGLE_CODE, 0);
136
137       list.ensureIndexIsVisible(index);
138     }
139   }
140
141   //This listener is shared by the text field and the on button.
142   class OnListener implements ActionListener {
143     public void actionPerformed(ActionEvent e) {
144       //This method can be called only if
145       //there's a valid selection
146       int index = list.getSelectedIndex();
147       int address=Integer.parseInt( listModel.elementAt(index).substring(0, 2).trim());
148
149       Commands.executeCommand(address,Commands.TOGGLE_CODE, 1);
150
151       list.ensureIndexIsVisible(index);
152     }
153   }
154
155   //This listener is shared by the text field and the on button.
156   class SetPointListener implements ActionListener {
157     public void actionPerformed(ActionEvent e) {
158       //This method can be called only if
159       //there's a valid selection
160       int index = list.getSelectedIndex();
161       int address=Integer.parseInt( listModel.elementAt(index).substring(0, 2).trim());
162
163       Commands.executeCommand(address,Commands.TOGGLE_CODE, 1);
164
165       list.ensureIndexIsVisible(index);
166     }
167   }
168
169   //This method is required by ListSelectionListener.
170   public void valueChanged(ListSelectionEvent e) {
171     if (e.getValueIsAdjusting() == false) {
172
173       if (list.getSelectedIndex() == -1) {
174         //No selection, disable buttons.
175         offButton.setEnabled(false);
176         onButton.setEnabled(false);
177
```

```
178       } else {
179         //Selection, enable the buttons.
180         offButton.setEnabled(true);
181         onButton.setEnabled(true);
182 
183       }
184     }
185   }
186 
187   /**
188    * Create the GUI and show it.  For thread safety,
189    * this method should be invoked from the
190    * event-dispatching thread.
191    */
192   private static void createAndShowGUI() {
193     //Create and set up the window.
194     JFrame frame = new JFrame("Domotic APP");
195     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
196 
197     //Create and set up the content pane.
198     JComponent newContentPane = new App();
199     newContentPane.setOpaque(true); //content panes must be opaque
200     frame.setContentPane(newContentPane);
201 
202     //Display the window.
203     frame.pack();
204     frame.setVisible(true);
205   }
206 
207 
208   public static void main(String[] args) {
209     //Schedule a job for the event-dispatching thread:
210     //creating and showing this application's GUI.
211     javax.swing.SwingUtilities.invokeLater(new Runnable() {
212       public void run() {
213         createAndShowGUI();
214       }
215     });
216     try {
217       Arduino.start();
218       try {
219         Thread.sleep(2000);
220       } catch (InterruptedException e) {
221         e.printStackTrace();
222       }
223       arduino=new Arduino();
224 
225     } catch (IOException e) {
226       System.out.println("Server problem");
```

```
227         e.printStackTrace();
228       }
229     }
230 }
```

C.1.2 Arduino.java

```
1  package domoServer;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import java.io.OutputStream;
7  import java.net.ServerSocket;
8  import java.net.Socket;
9
10 public class Arduino
11 {
12   private static OutputStream output;
13   private static BufferedReader input;
14   public static final short MAX_BOARDS=32;
15
16   public static void start() throws IOException
17   {
18     ServerSocket serverSocket = null;
19
20     try {
21       serverSocket = new ServerSocket(993);
22     }
23     catch (IOException e)
24     {
25       System.err.println("Could not listen on port.");
26       System.exit(1);
27     }
28
29     Socket clientSocket = null;
30
31     try {
32       System.out.println ("Waiting for Client");
33       clientSocket = serverSocket.accept();
34     }
35     catch (IOException e)
36     {
37       System.err.println("Accept failed.");
38       System.exit(1);
39     }
40
41     try {
42       output = clientSocket.getOutputStream();
43       input = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
```

```
44      } catch (IOException e) {
45        System.err.println("Get streams failed");
46        System.exit(1);
47      }
48
49      serverSocket.close();
50      System.out.println ("Server socket closed!");
51    }
52
53    public static String pinModeCmd(int board, int pin, int mode){
54      return String.format("%02d"+"%02d"+"%02d"+"%03d",Functions.PIN_MODE, pin, board, mode)+"S"
        ;
55    }
56    public static String digitalReadCmd(int board, int pin){
57      return String.format("%02d"+"%02d"+"%02d"+"%03d",Functions.DIGITAL_READ, pin, board, 0)+"S
        ";
58    }
59    public static String analogReadCmd(int board, int pin){
60      return String.format("%02d"+"%02d"+"%02d"+"%03d",Functions.ANALOG_READ, pin, board, 0)+"S"
        ;
61    }
62    public static String digitalWriteCmd(int board, int pin, int value){
63      return String.format("%02d"+"%02d"+"%02d"+"%03d",Functions.DIGITAL_WRITE, pin, board,
        value)+"S";
64    }
65    public static String analogWriteCmd(int board, int pin, int value){
66      return String.format("%02d"+"%02d"+"%02d"+"%03d",Functions.ANALOG_WRITE, pin, board, value
        )+"S";
67    }
68    public static String getVersionCmd(int board){
69      return String.format("%02d"+"%02d"+"%02d"+"%03d",Functions.GET_VERSION, 0, board, 0)+"S";
70    }
71
72    public static int pinMode(int board, int pin, int mode){
73      try {
74        output.write(Arduino.pinModeCmd(board, pin, mode).getBytes());
75      } catch (Exception e) {
76        System.out.println("could not write to port");
77      }
78      try {
79        int[] answer;
80        String inputLine = input.readLine();
81        answer=splitMessage(inputLine);
82        if(answer[0]==Error.CMD_OK_ANS)
83          return answer[1];
84        else
85          return answer[0];
86      } catch (Exception e) {
87        System.err.println(e.toString());
```

```
88      }
89      return Error.UNKNOWN_ERROR;
90    }
91
92    public static int digitalRead(int board, int pin){
93      try {
94        output.write(Arduino.digitalReadCmd(board, pin).getBytes());
95      } catch (Exception e) {
96        System.out.println("could not write to port");
97      }
98      try {
99        int[] answer;
100       String inputLine = input.readLine();
101       answer=splitMessage(inputLine);
102       if(answer[0]==Error.CMD_OK_ANS)
103         return answer[1];
104       else
105         return answer[0];
106     } catch (Exception e) {
107       System.err.println(e.toString());
108     }
109     return Error.UNKNOWN_ERROR;
110   }
111
112   public static int analogRead(int board, int pin){
113
114     try {
115       output.write(Arduino.analogReadCmd( board, pin).getBytes());
116     } catch (Exception e) {
117       System.out.println("could not write to port");
118     }
119     try {
120       int[] answer;
121       String inputLine = input.readLine();
122       answer=splitMessage(inputLine);
123       if(answer[0]==Error.CMD_OK_ANS)
124         return answer[1];
125       else
126         return answer[0];
127     } catch (Exception e) {
128       System.err.println(e.toString());
129     }
130     return Error.UNKNOWN_ERROR;
131   }
132
133   public static int digitalWrite(int board, int pin, int value){
134     try {
135       output.write(Arduino.digitalWriteCmd(board, pin, value).getBytes());
136     } catch (Exception e) {
```

```
137       System.out.println("could not write to port");
138     }
139     try {
140       int[] answer;
141       String inputLine = input.readLine();
142       answer=splitMessage(inputLine);
143       if(answer[0]==Error.CMD_OK_ANS)
144         return answer[1];
145       else
146         return answer[0];
147     } catch (Exception e) {
148       System.err.println(e.toString());
149     }
150     return Error.UNKNOWN_ERROR;
151   }
152
153   public static int analogWrite(int board, int pin, int value){
154     try {
155       output.write(Arduino.analogWriteCmd(board, pin, value).getBytes());
156     } catch (Exception e) {
157       System.out.println("could not write to port");
158     }
159     try {
160       int[] answer;
161       String inputLine = input.readLine();
162       answer=splitMessage(inputLine);
163       if(answer[0]==Error.CMD_OK_ANS)
164         return answer[1];
165       else
166         return answer[0];
167     } catch (Exception e) {
168       System.err.println(e.toString());
169     }
170     return Error.UNKNOWN_ERROR;
171   }
172
173   public static int getVersion(int board){
174     try {
175       output.write(Arduino.getVersionCmd(board).getBytes());
176     } catch (Exception e) {
177       System.out.println("could not write to port");
178     }
179     try {
180       int[] answer;
181       String inputLine = input.readLine();
182       answer=splitMessage(inputLine);
183       if(answer[0]==Error.CMD_OK_ANS)
184         return answer[1];
185       else
```

```
186         return answer[0];
187       } catch (Exception e) {
188         System.err.println(e.toString());
189       }
190       return Error.UNKNOWN_ERROR;
191     }
192
193     private static int[] splitMessage(String inputLine) {
194       //inputline=(error,value)
195       int[] parts=new int[2];
196       String[] splited = inputLine.split("\\.");
197       parts[0]=Integer.valueOf(splited[0]);
198       if(splited.length>1)
199         parts[1]=Integer.valueOf(splited[1]);
200       return parts;
201     }
202
203
204     public OutputStream getOutputStream(){
205       return output;
206     }
207     public void setOutputStream(OutputStream out){
208       output=out;
209     }
210 }
```

C.1.3 FieldDevice.java

```
1  package devices;
2
3  import java.io.Serializable;
4
5  import domoServer.Arduino;
6
7  public class FieldDevice implements Serializable
8  {
9    private static final long serialVersionUID = 1L;
10
11   public static final int MOTION=1;
12   public static final int TBUTTON=2;
13   public static final int RELAY=3;
14   public static final int AMBIENT=4;
15
16   private int address;
17   private int function;
18   private int state;
19   // A trick to help with debugging
20   private boolean debug;
21
22   public static FieldDevice createModuleInstance(int address, int function, int value)
```

```
23  {
24    FieldDevice ret=null;
25
26    if(function==FieldDevice.MOTION)
27      ret = new MotionSensor(address, value);
28    else if(function==FieldDevice.RELAY)
29      ret = new RelayActuator(address, value);
30    else if(function==FieldDevice.TBUTTON)
31      ret = new ButtonSensor(address, value);
32    else if(function==FieldDevice.AMBIENT)
33      ret = new AmbientSensor(address, value);
34
35    return ret;
36  }
37
38  public void dprint (String s)
39  {
40    // print the debugging string only if the "debug"
41    // data member is true
42    if (debug)
43      System.out.println("Debug: " + s);
44  }
45
46  public void setDebug (boolean b)
47  {
48    debug = b;
49  }
50
51  public FieldDevice(int address, int function, int value)
52  {
53    this.address=address;
54    this.function=function;
55    this.state=value;
56
57    // turn off debugging
58    debug = false;
59  }
60
61  public FieldDevice ()
62  {
63    this (0,0,0);
64  }
65
66  public void setAddress(int newAddress)
67  {
68    dprint ("setAddresss(): Changing state from " + this.address + " to " + newAddress );
69    this.address = newAddress;
70  }
71
```

```
72  public void setFunctionCode(int newCode)
73  {
74    dprint ("setAddresss(): Changing state from " + this.function + " to " + newCode );
75    this.address = newCode;
76  }
77
78  public int getAddress()
79  {
80    return this.address;
81  }
82
83  public int getFunctionCode()
84  {
85    return this.function;
86  }
87
88  public int getState()
89  {
90    refreshState();
91    return this.state;
92  }
93
94  public void setState(int value)
95  {
96    this.state=value;
97  }
98
99  public void refreshState()
100 {
101   int value=0;
102   switch(function)
103   {
104   case TBUTTON:
105     value=Arduino.digitalRead(this.getAddress(),10);
106     break;
107   case MOTION:
108     value=Arduino.digitalRead(this.getAddress(),MotionSensor.DETECTION_PIN);
109     break;
110   case RELAY:
111     Arduino.pinMode(this.getAddress(), RelayActuator.CONTROL_PIN, 0);
112     value=Arduino.digitalRead(this.getAddress(),RelayActuator.CONTROL_PIN);
113     break;
114   case AMBIENT:
115     value=Arduino.analogRead(this.getAddress(),AmbientSensor.TEMPERATURE_PIN);
116     break;
117   }
118   if(value>=0)
119     this.setState(value);
120 }
```

```
121
122   public String getFunctionCodeString()
123   {
124     if(function==RELAY)
125       return "RELAY ACTUATOR";
126     if(function==MOTION)
127       return "MOTION SENSOR";
128     if(function==TBUTTON)
129       return "TOUCH BUTTON";
130     if(function==AMBIENT)
131       return "AMBIENT SENSOR";
132     return "";
133   }
134 }
```

C.1.4 ButtonSensor.java

```
1  package devices;
2
3  import java.io.Serializable;
4
5  public class ButtonSensor extends FieldDevice implements Serializable
6  {
7    private static final long serialVersionUID = 1L;
8    public static final int BUTTON_PIN=5;
9
10   public ButtonSensor(int address, int state)
11   {
12     super(address,FieldDevice.TBUTTON,state);
13   }
14
15   public ButtonSensor ()
16   {
17     super (0,FieldDevice.TBUTTON,0);
18   }
19
20   public ButtonSensor (ButtonSensor btn)
21   {
22     super (btn.getAddress(),FieldDevice.TBUTTON,btn.getState());
23   }
24 }
```

C.1.5 MotionSensor.java

```
1  package devices;
2
3  import java.io.Serializable;
4
5  public class MotionSensor extends FieldDevice implements Serializable
6  {
```

```
  private static final long serialVersionUID = 1L;
  public static final int DETECTION_PIN=11;
  public static final int SLEEP_PIN=10;

  public MotionSensor(int address, int state)
  {
    super(address,FieldDevice.MOTION,state);
  }

  public MotionSensor ()
  {
    super (0,FieldDevice.MOTION,0);
  }

  public MotionSensor (MotionSensor btn)
  {
    super (btn.getAddress(),FieldDevice.MOTION,btn.getState());
  }
}
```

C.1.6 RelayActuator.java

```
package devices;

import java.io.*;

public class RelayActuator extends FieldDevice implements Serializable
{
  private static final long serialVersionUID = 1L;
  public static final int CONTROL_PIN=11;

  public RelayActuator(int address, int value)
  {
    super(address,FieldDevice.RELAY,value);
  }

  public RelayActuator ()
  {
    super (0,FieldDevice.RELAY,0);
  }

  public RelayActuator (RelayActuator btn)
  {
    super (btn.getAddress(),FieldDevice.RELAY,btn.getState());
  }
}
```

C.1.7 AmbientSensor.java

```
package devices;
```

```
2
3  import java.io.Serializable;
4
5  public class AmbientSensor extends FieldDevice implements Serializable
6  {
7    private static final long serialVersionUID = 1L;
8    public static final int HUMIDITY_PIN=5;
9    public static final int TEMPERATURE_PIN=5;
10
11   public AmbientSensor(int address, int state)
12   {
13     super(address,FieldDevice.AMBIENT,state);
14   }
15
16   public AmbientSensor ()
17   {
18     super (0,FieldDevice.AMBIENT,0);
19   }
20
21   public AmbientSensor (AmbientSensor btn)
22   {
23     super (btn.getAddress(),FieldDevice.AMBIENT,btn.getState());
24   }
25 }
```

Appendix D

Test Suite

D.1 Photos

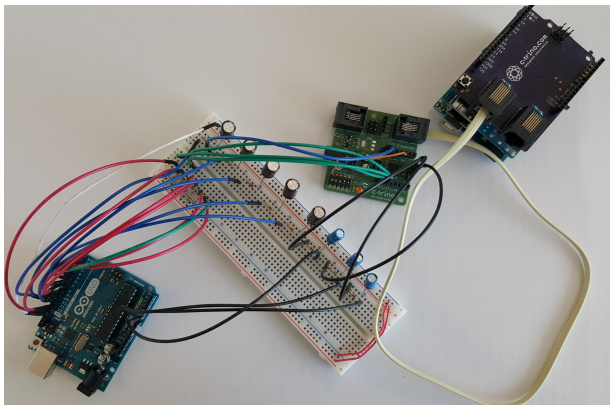


Figure D.1: Testbed connections

D.2 Source Code

D.2.1 TestAnalogOperations.java

```
1  package test;
2
3  import static org.junit.Assert.assertEquals;
4
5  import java.util.HashMap;
6  import java.util.Iterator;
7  import java.util.Random;
8
9  import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import server.Arduino;
```

```
14 import server.Error;
15
16 public class TestAnalogOperations{
17
18   private int ctrino;
19   private int arduino;
20   private int tolerance;
21
22   //Ctrino-Arduino connections
23   private HashMap<Integer, Integer> PWMToAnalogMap;
24
25   private Arduino master;
26   private Arduino slave;
27
28   @Before
29   public void initialize() {
30     tolerance=10;
31
32     master = new Arduino("COM8","Master");
33     slave = new Arduino("COM7","Subject");
34     master.initialize();
35     slave.initialize();
36     ctrino=3;
37     arduino=0;
38
39     try {
40       Thread.sleep(2000); //Wait  for connection
41     } catch (InterruptedException e) {
42       e.printStackTrace();
43     }
44   }
45
46   @After
47   public void finalize() {
48     master.close();
49     slave.close();
50   }
51
52   @Test
53   public void testAnalogWrite()
54   {
55     initAnalogWrite();
56     HashMap<Integer, Integer> analogValues = new HashMap<Integer, Integer>();
57
58     Random rnd=new Random();
59     int value=0;
60
61     Iterator<Integer> pinSetIterator = PWMToAnalogMap.keySet().iterator();
62     while(pinSetIterator.hasNext()){
```

```
63      int pin = pinSetIterator.next();
64      value=rnd.nextInt(Arduino.MAX_VALUE);
65
66      assertEquals(Error.CMD_OK, slave.pinMode(ctrino, pin, Arduino.OUTPUT));
67      assertEquals(Error.CMD_OK, slave.analogWrite(ctrino, pin, value));
68      analogValues.put(pin, value);
69    }
70
71    try {
72      Thread.sleep(6000); //wait for voltage to stabilize, caused by capacitor charge delay
73    } catch (InterruptedException e) {
74      e.printStackTrace();
75    }
76
77    int pin=0;
78    Iterator<Integer> keySetIterator = PWMToAnalogMap.keySet().iterator();
79
80    while(keySetIterator.hasNext()){
81      int key = keySetIterator.next();
82      pin=PWMToAnalogMap.get(key);
83
84      assertEquals(Error.CMD_OK, master.pinMode(arduino, pin, Arduino.INPUT));
85      assertEquals(analogValues.get(key), master.analogRead(arduino, pin),tolerance);
86    }
87  }
88
89
90  @Test
91  public void testAnalogRead()
92  {
93    initAnalogRead();
94    HashMap<Integer, Integer> analogValues = new HashMap<Integer, Integer>();
95
96    Random rnd=new Random();
97    int value=0;
98
99    Iterator<Integer> pinSetIterator = PWMToAnalogMap.keySet().iterator();
100   while(pinSetIterator.hasNext()){
101     Integer pin = pinSetIterator.next();
102     value=rnd.nextInt(254);
103
104     assertEquals(Error.CMD_OK, master.pinMode(arduino, pin, Arduino.OUTPUT));
105     assertEquals(Error.CMD_OK, master.analogWrite(arduino, pin, value));
106     analogValues.put(pin, value);
107   }
108   try {
109     Thread.sleep(6000); //wait for voltage to stabilize, caused by capacitor charge delay
110   } catch (InterruptedException e) {
111     e.printStackTrace();
```

```
112     }
113
114     int pin=0;
115     Iterator<Integer> keySetIterator = PWMToAnalogMap.keySet().iterator();
116
117     while(keySetIterator.hasNext()){
118       int key = keySetIterator.next();
119       pin=PWMToAnalogMap.get(key);
120
121       assertEquals(Error.CMD_OK, slave.pinMode(ctrino, pin, Arduino.INPUT));
122       assertEquals(analogValues.get(key), slave.analogRead(ctrino, pin),tolerance);
123     }
124   }
125
126   public void initAnalogRead(){
127     //(key)Arduino->(value)Ctrino
128     PWMToAnalogMap=new HashMap<Integer, Integer>();
129     PWMToAnalogMap.put(11, 0);
130
131   }
132
133   public void initAnalogWrite(){
134     //(key)Ctrino->(value)Arduino
135     PWMToAnalogMap=new HashMap<Integer, Integer>();
136 //    PWMToAnalogMap.put(17, 1);
137 //    PWMToAnalogMap.put(18, 0);
138     PWMToAnalogMap.put(16, 2);
139   }
140
141 }
```

D.2.2 TestDigitalOperations.java

```
1  package test;
2
3  import static org.junit.Assert.assertEquals;
4
5  import java.util.HashMap;
6  import java.util.Iterator;
7  import java.util.Random;
8
9  import org.junit.After;
10 import org.junit.Before;
11 import org.junit.Test;
12
13 import server.Arduino;
14 import server.Error;
15
16 public class TestDigitalOperations {
17   private int ctrino;
```

```
18  private int arduino;
19
20  //Ctrino-Arduino connections
21  private HashMap<Integer, Integer> IOMap;
22
23  private Arduino master;
24  private Arduino slave;
25
26  @Before
27  public void initialize() {
28
29    master = new Arduino("COM8","Master");
30    slave = new Arduino("COM7","Subject");
31    master.initialize();
32    slave.initialize();
33    ctrino=3;
34    arduino=0;
35
36    try {
37      Thread.sleep(2000); //Wait  for connection
38    } catch (InterruptedException e) {
39      e.printStackTrace();
40    }
41  }
42
43  @After
44  public void finalize() {
45    master.close();
46    slave.close();
47  }
48
49  @Test
50  public void testDigitalWrite()
51  {
52    initDigitalWrite();
53    HashMap<Integer, Integer> digitalValues = new HashMap<Integer, Integer>();
54
55    Random rnd=new Random();
56    int value=0;
57
58    Iterator<Integer> pinSetIterator = IOMap.keySet().iterator();
59    while(pinSetIterator.hasNext()){
60      int pin = pinSetIterator.next();
61      value=rnd.nextInt(2);
62
63      assertEquals(Error.CMD_OK, slave.pinMode(ctrino, pin, Arduino.OUTPUT));
64      assertEquals(Error.CMD_OK, slave.digitalWrite(ctrino, pin, value));
65      digitalValues.put(pin, value);
66    }
```

```
67
68    int pin=0;
69    Iterator<Integer> keySetIterator = IOMap.keySet().iterator();
70
71    while(keySetIterator.hasNext()){
72      int key = keySetIterator.next();
73      pin=IOMap.get(key);
74
75      assertEquals(Error.CMD_OK, master.pinMode(arduino, pin, Arduino.INPUT));
76      assertEquals(digitalValues.get(key).intValue(), master.digitalRead(arduino, pin));
77    }
78  }
79
80
81  @Test
82  public void testDigitalRead()
83  {
84    initDigitalRead();
85    HashMap<Integer, Integer> digitalValues = new HashMap<Integer, Integer>();
86
87    Random rnd=new Random();
88    int value=0;
89
90    Iterator<Integer> pinSetIterator = IOMap.keySet().iterator();
91    while(pinSetIterator.hasNext()){
92      Integer pin = pinSetIterator.next();
93      value=rnd.nextInt(2);
94
95      assertEquals(Error.CMD_OK, master.pinMode(arduino, pin, Arduino.OUTPUT));
96      assertEquals(Error.CMD_OK, master.digitalWrite(arduino, pin, value));
97      digitalValues.put(pin, value);
98    }
99
100   int pin=0;
101   Iterator<Integer> keySetIterator = IOMap.keySet().iterator();
102
103   while(keySetIterator.hasNext()){
104     int key = keySetIterator.next();
105     pin=IOMap.get(key);
106
107     assertEquals(Error.CMD_OK, slave.pinMode(ctrino, pin, Arduino.INPUT));
108     assertEquals(digitalValues.get(key).intValue(), slave.digitalRead(ctrino, pin));
109   }
110 }
111
112 public void initDigitalRead(){
113   //(key)Arduino->(value)Ctrino
114   IOMap=new HashMap<Integer, Integer>();
115   IOMap.put(13, 15);
```

```
116     IOMap.put(12, 14);
117     IOMap.put(8, 13);
118   }
119
120   public void initDigitalWrite(){
121     //(key)Ctrino->(value)Arduino
122     IOMap=new HashMap<Integer, Integer>();
123     IOMap.put(15, 13);
124     IOMap.put(14, 12);
125     IOMap.put(13, 8);
126   }
127
128 }
```

D.2.3 TestErrorIdentification.java

```
1  package test;
2
3  import static org.junit.Assert.assertEquals;
4
5  import org.junit.After;
6  import org.junit.Before;
7  import org.junit.Test;
8
9  import server.Arduino;
10 import server.Error;
11
12 public class TestErrorIdentification {
13   private int ctrino;
14   private Arduino slave;
15
16   @Before
17   public void initialize() {
18     slave = new Arduino("COM7","Subject");
19     slave.initialize();
20     ctrino=3;
21     try {
22       Thread.sleep(2000); //Wait  for connection
23     } catch (InterruptedException e) {
24       e.printStackTrace();
25     }
26   }
27
28   @After
29   public void finalize() {
30     slave.close();
31   }
32
33   @Test
34   public void testNoBoardError()
```

```
35    {
36      int board=15;
37      while(board<20){
38        assertEquals(Error.NO_BOARD, slave.pinMode(board, 13, Arduino.OUTPUT));
39        board++;
40      }
41    }
42
43
44    @Test
45    public void testMiscError()
46    {
47      int pin=28;
48      while(pin<30){
49        assertEquals(Error.MISC_ERROR, slave.pinMode(ctrino, pin, Arduino.OUTPUT));
50        pin++;
51      }
52    }
53  }
```