Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

BACHELOR'S THESIS

TITLE : IOSharp: .NET Micro Framework on Linux

DEGREE: Bachelor in Telematics Engineering

AUTHOR: Gerard Solé i Castellví

DIRECTOR: Juan López Rúbio

DATE: February 7, 2014

Títol : IOSharp: .NET Micro Framework on Linux

Autor: Gerard Solé i Castellví

Director: Juan López Rúbio

Data: 7 de febrer de 2014

Resum

Cada vegada més l'Internet de les Coses està en auge impulsant tot tipus de dispositius incrustats i sistemes operatius per aquests dispositius. L'objectiu d'aquest projecte de fi de carrera és desenvolupar una llibreria capaç d'executar aplicacions desenvolupades per a .NET Micro Framework (màquina virtual per a dispositius incrustats) en dispositius que puguin executar qualsevol versió de Linux. Això permetria executar aplicacions de .NET Micro Framework en ordinadors d'escriptori o en dispositius embeguts que executin Linux, com per exemple Raspberry Pi.

En aquest document es presenta el plantejament i el desenvolupament d'IOSharp que correspon a la implementació en C# del sistema comentat anteriorment. Un cop aquesta implementació s'ha dut a terme, s'ha passat a la fase de proves amb les aplicacions existents on s'ha pogut determinar que el rendiment de la llibreria no és òptim. En aquest punt és on hi ha hagut una de les altres contribucions principals en el projecte, s'ha participat en el desenvolupament d'una eina anomenada AlterNative capaç de traduir aplicacions escrites en .NET a C++, d'aquesta manera s'avaluarà la millora del rendiment de la llibreria al traduir-la amb l'AlterNative.

Title : IOSharp: .NET Micro Framework on Linux

Author: Gerard Solé i Castellví

Director: Juan López Rúbio

Date: February 7, 2014

Overview

Increasingly, the Internet of Things is promoting all types of embedded devices and operating systems for these devices. The aim of this project is to develop library capable of running applications developed for .NET Micro Framework (virtual machine for embedded devices) on devices that can run any version of Linux. The idea is that using this library a developer can execute any application developed for .NET Micro Framework in any desktop computer or embedded device running Linux, for example Raspberry Pi.

This thesis presents the approach and the development of IOSharp which is an implementation in C# of the system discussed above. Once this implementation has been done, a test phase has been done with existing applications. In this case, it has been proved that the performance of this library is not optimal compared with the original one. At this point is where started another of the main contributions to the project, participating in the development of a tool called AlterNative which is able to translate applications written in .NET to C++, In this way it has been possible to analyse the improvement of this library after translating it with AlterNative.

Un gran agraïment als meus pares, germanes i avis. Sense el suport dels quals hauria estat impossible tot el temps dedicat tant a la carrera com en el projecte.

També a Juan López, Alex Albalá i Carla Iriberri els quals han tingut la paciència suficient per a contribuir en el bon camí d'aquest treball.

A Adrián Galera, Alberto Toro, Jesús Alcober, Marc Bajet, Norbert Nebra, Santi Pérez, Toni Oller, juntament amb Juan López i Alex Albalá. Treballar amb tots vosaltres dona gust.

Finalment a Pau Martínez, Marc Beltrán i Marta Jiménez pel gran suport en les últimes fases d'aquest projecte.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Embedded systems have become more powerful over time passing from 8 bit controllers to 8 or 16 bit microprocessors or even 32 bit ARM microprocessors. Apart from the increase of the processing power, the memory included in this devices has also increased, from tens of Kilobytes to tens or hundreds of Megabytes. One of the reasons for these changes has been the price drop on production. These new embedded systems offer a performance similar to the computers from the nineties and most of them implement operating systems which helps on reducing the difficulty to create embedded applications. With an operating system running over the bare metal of the chip, the developer will get all the underlying hardware abstracted to different APIs and libraries and avoiding low level interaction with the hardware. They also offer interesting features such as memory control and allocation, threading or dynamic program loading. For example, one of the top notch devices nowadays is the Raspberry Pi which provides an ARMv6, 512MB of RAM and some I/O features such as GPIOs, SPI, UART. This hardware can be equivalent to a nineties computer like a Pentium II so they are really powerful regarding the task that they may do.
One of the operating systems available for embedded devices is .NET Micro Framework (NETMF) developed by Microsoft. This system is the smallest version of .NET Framework and is oriented to resource-constrained devices for embedded applications. This system offers different communication protocols and methods like GPIO ports, SPI, UART and I^2C. There is no official implementation or port of Micro Framework capable of running on standard computers (a normal desktop), so any application written for this operating system will not work on Linux or Windows. Although a minimal port of Micro Framework exists for a Linux board called Eddy, it does not offer all the hardware features that Micro Framework does.
The aim of IOSharp is to solve this lack and get Micro Framework applications run on any Linux machine that is capable of running applications using the .NET Framework (the complete stack designed by Microsoft). So this project instead of writing a complete port of the Micro Framework runtime to run on Linux, is an extension to the classes provided by the .NET Framework. Basically, IOSharp offers the I/O functions, methods and classes that are missing on .NET Framework, and although the implementation of the classes is different in IOSharp than in Micro Framework, the namespaces, methods, class naming, etc is equal to the original one so this makes much easier to migrate between Micro Framework to .NET Framework with IOSharp.
The origin of this project resides in the need of migrating the code of an existing Wireless Sensor Network (WSN) gateway implemented on Micro Framework for a Netduino Mini board. One of the problems that this platform has is that the actual device is getting out of system resources, so in order to keep the existing code, different solutions are being researched. In this case IOSharp intends to be one of those solutions achieving the deployment of this WSN gateway software on a Raspberry Pi which runs a Linux operating system.

After achieving the first goal, there is a second step of this project which consists on extending a code translation tool called AlterNative which is being developed by Alex Albalá and Juan López. This translator is capable to get the source code from a C# assembly and then translate it to C++ trying to improve the performance. C++ theoretically performs

better than C# applications, but normally are more platform oriented, so a Windows assembly will not work on a Linux system, but one of the interesting features of AlterNative is the generation of a highly portable code that can be compiled on (and for) any operating system, i.e. Windows, Linux, MacOSX, iOS, Android, etc. Another interesting point of this tool is that the generated code is similar to C# so a developer used to this language will be able to understand or even write programs using the translated code.

CHAPTER 1. PROJECT OVERVIEW

This project was proposed by AlterAid, a company which is working on several ways to help in taking care of the health of our elderly, or in general, anyone that is relevant to our lives.

This company is working on two different projects that combine together. The first one is called aaaida, which consists of a social network where people can stay alert about its relatives, upload information about its health or watch recommendations from doctors or other professionals. On the other side, and on a more hardware oriented development, they are creating a Wireless Sensor Network called HomeSense that once deployed in a house will be able to collect relevant information from those sensors in the home and allow other people to know if the daily life of the resident's house is going normal, or something is happening.

1.1 HomeSense

HomeSense is a Wireless Healthcare Sensor Platform created with the aim of control and care taking of elderly and relatives. Actually it uses a Netduino Mini board which makes the function of the gateway which controls the sensor network, receiving all the data and uploading to aaaida platform through Internet.
In the house, the communication is carried on using little sensors capable of fetching data in different situations (for example the opening of a drawer or a medicine cabinet). It is also possible to install the sensors on doors in order to know if they are opened or closed, or in any place where is interesting to acquire information from the environment, house or residents. These sensors make use of nRF24LE1 SoC with a low-power RF ISM band on 2.4GHz from Nordic Semiconductor.
The communication protocol designed for HomeSense is similar to a star network with multi-hop transmissions so it becomes a tree-star topology. The nodes mainly send information to the gateway because this is on charge of upload the information to the Internet, but they are also able to communicate with other nodes. The gateway system has been entirely developed using .NET Micro Framework and deployed on a Netduino Mini board.

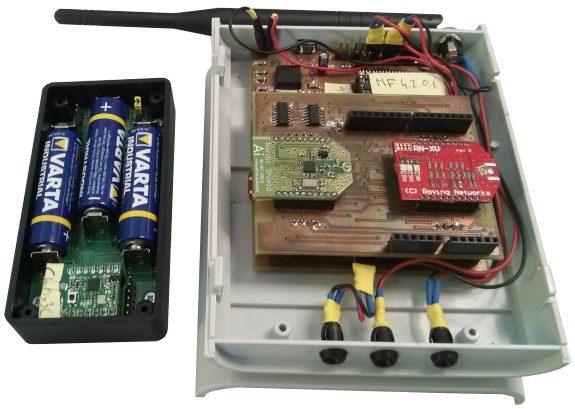

Figure 1.1: Sensor on the left and HomeSense gateway on the right

1.2 AlterNative

AlterNative is a language translating tool created by Alex Albalá and Juan López. It can translate a compiled (.NET) assembly or library to standard C++ code. Basically this program decompiles the file to be translated, then it sketches how the program works, which are its classes, functions, nodes, etc and then start translating step-by-step all the program. After that, it links the necessary C++ libraries to work, ones are from boost library, and the other ones are self-written to behave like the original C# classes.
It is interesting to emphasize that the main difference of this translator between the other existing ones is that it tries to generate a code practically identical to the original C# source code. By doing this, the resulting C++ source is really easy to read for people not used to C++ syntax and language.

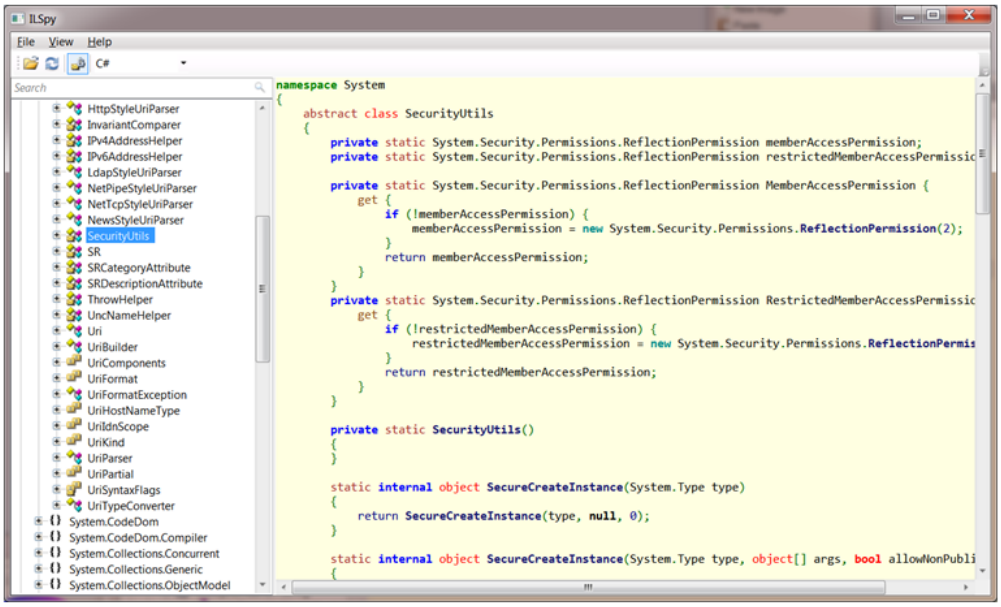

Figure 1.2: AlterNative interface

1.3 Thesis Proposal

After introducing HomeSense and AlterNative is time to explain the thesis proposal itself because it is related to the applications mentioned above. The proposed project is to take the code of the HomeSense gateway, which is written using Micro Framework, and make it run in a Linux device instead of a Netduino Mini. The reason is that the Netduino Mini is getting limited in terms of capabilities, performance and expansion for future characteristics.
The idea is to take the gateway code and port it to other devices capable of use minimum GPIO, interrupts from this ports, the SPI communication protocol and UART because all of them are used in the HomeSense source code. It is important to point that the implemented classes must be similar to the Micro Framework code to avoid major changes on the HomeSense program. Minor changes such as port renaming and communication

module name changing are acceptable, because they do not alter the original execution flow and design architecture. But not only this should be done on HomeSense, it is interesting to make portable between different hardware platforms any code that runs over Micro Framework.
After accomplishing with this first goal, the second part of the thesis is to use AlterNative to translate the IOSharp library to C++ in order to increase and analyse the performance of IOSharp running on C++ instead of C#. To accomplish with this objective, some C++ libraries must be written in order to translate IOSharp.

1.3.1 Objectives

The proposed objectives are listed below:

- **IOSharp:** to be accomplished during the first part of this thesis.
  - **GPIO:** Simple I/O functions (Input, Output ports).
  - **Interrupts:** enable interrupts through GPIOs.
  - **SPI:** get a working SPI bus on the implementation.
  - **UART:** do serial transmissions.
  - **HomeSense:** deploy this WSN as a functional test to show the correct function of this library.
- **AlterNative:** second part of the thesis involving the code translation tool.
  - **Cross-Platform:** although one of the goals of AlterNative is produce a cross-platform source code, it only runs on Windows so Linux or MacOSX users are unable to use this tool. In this case, the code must be analysed and modified to produce a compatible program with any operating system capable of run .NET code.
  - **Library:** write the necessary library methods to translate IOSharp.
  - **Tests:** functional tests to determine the correct functionality of the above methods.
  - **Performance analysis:** do some performance analysis to check the performance increase when the code is translated.
  - **Translate HomeSense:** do a complete translation of the WSN platform using AlterNative.

1.4 Document Structure

This document is structured on seven chapters. The first one shows the project definition. Then a state of the art is used to show the current technologies or systems that are similar to this thesis goal. Another chapter sketches the how has been carried out the development. The functional test chapter wants to proofs that IOSharp implementation objectives

are done. Then an introduction to AlterNative is done along with the performance tests chapter which shows the results for the AlterNative part. Finally, the conclusions chapter summarizes the thesis results.

CHAPTER 2. STATE OF THE ART

This chapter briefly sketches out the state of the art of the embedded operating systems and its capabilities.

2.1 Embedded Systems

Embedded Systems nowadays are taking relevance again with the Internet of Things, environment sensing, Wireless Sensor Networks and all new coming technologies that require low power consumption, small size, mobility environments, etc.

In Embedded Systems or Resource Constrained Systems it is interesting to take a look into the Hardware platform and its capabilities, the differences between platforms, and also which tools or unique features they offer to developers.

An operating system (OS) offers an interface with the hardware to make it independent from the applications that the device runs, making easy the interactions between hardware and the programs running on the machine.
Until now, most embedded devices did not make use of operating systems and they where totally oriented to the designated application but as it has been explained in the introduction, the cost reduction in the production of systems has helped to increase the capacities so now is not unusual to embed a Linux system to simplify the manage of the resources.
An OS is an important software that makes easy to develop applications, but it is important to maintain the features that the processor offers, avoiding performance or capabilities degradation. This bachelor thesis is focused on constrained-resource devices, where the processing capabilities and memory resources are limited, is fundamental to respect the above criteria.

2.1.1 Operating Systems Architectures

In general, there are three types of operating system architectures for embedded devices, which are based on how applications are executed or included into the OS.

- **Monolithic:** The OS and the applications are combined into a single program. Normally in this situations the embedded device runs in the same process the OS and the program written to it. This type of architecture makes difficult to include new functions without rewriting much of the code.
- **Modular:** The OS is running as a standalone program in the processor and has the ability of loading programs as modules. In terms of the development, it's possible to develop applications without writing in the core of the OS. Normally using modules developers can expand the capabilities of its software.

- **Virtual-Machine:** The virtual machine creates an abstraction layer of its underlying hardware. This abstracted layer is common in every device that implements that virtual-machine. Using this type of operating system provides a helpful tool to achieve the well known slogan *write once, run anywhere*. Although using virtual-machine devices simplifies the development on multiple devices, the performance of the platform normally will be reduced and in Real Time environments it is not recommended to use it. It is interesting to remark that embedded virtual machines, differently from VMs in desktop or server environments, run on the bare metal so they also act as an operating system.

2.1.2 Embedded Operating Systems

There is a wide range of Embedded Operating Systems each of them has strengths and weaknesses. Below different OS are described and compared.

- **TinyOS** is a popular open source OS for wireless constrained devices, many of them used in wireless sensor networks. It provides software abstractions from the underlying hardware. It is focused on wireless communications offering stacks for 6LoWPAN and ZigBee. It also supports secure networking and implements a RPL taking in mind the forthcoming routing protocol for low power and lossy networks.
  However, TinyOS changes how programs should be developed, it intended to use non-blocking programming which means that it is not prepared for long processing functions. For example, when TinyOS called to send a message the function will return immediately and after a while the send will be processed and then, TinyOS will make a callback to a function, for example `send()`'s callback will be `sendDone()`.

- **FreeRTOS** is a free real-time OS that supports over 34 architectures and it is being developed by professionals under strict quality controls and robustness. It is used from toys to aircraft navigation and it is interesting for its real-time qualities. It has a very small memory footprint (RAM usage) and very fast execution, based on hard real-time interrupts performed by queues and semaphores. Apart from this, there are not constraints on the maximum number of tasks neither the priority levels that can be used on tasks.

- **Contiki** is similar to TinyOS in terms of portability between platforms and its code is open source. It also offers features similar to standard operating systems like threading, timers, file system and command line shell and uses modular architecture, loading or unloading programs from its kernel. Contiki is built on top of the Internet Standards supporting IPv4 and IPv6 and also the new low-power internet protocols which includes 6LoWPAN, RPL and CoAP.
  Contiki uses protothreads, which are designed for event-driven systems running on top of constrained devices, which is the case of Contiki's kernel. It provides blocking without having a real multi-threading system or a stack-switching.

- **Micro Framework .NET** is a solution provided by Microsoft for resource-constrained devices which cannot execute the full .NET stack. It is Virtual-Machine based operating system that has a small implementation of the CLR making available to execute

a small set of .NET classes. Its memory footprint is about 300KB and supports the common embedded peripherals like EEPROM, GPIO, SPI, UART, USB, ...
One of its interesting features is that it offers the advantages of .NET language using Visual Studio and it also offers real-time debugging directly on the device.

2.2 .NET Micro Framework

Micro Framework, also known as NETMF, has its roots in a project called **Smart Personal Objects Technology (SPOT)**. The first devices implementing the SPOT technology were smart-watches from Fossil and Suunto in 2004, but after them, some other devices also made use of SPOT, like kettles, weather stations and even map updates in Garmin devices. Microsoft wanted to create a technology for everyday devices, so they launched together with SPOT the MSN Direct, which was a set of network services capable of delivering information to the SPOT devices using FM radio broadcast signals.
In 2008 the production of SPOT watches was discontinued, and in 2009 Microsoft released the source code of Micro Framework under Apache 2.0 license, making availably to the community, and shortly after this release the MSN Direct services where ceased.

Figure 2.1: SPOT watch

2.2.1 Devices using Micro Framework

Since Microsoft released the Micro Framework source code, companies had created different devices supporting .NET code and this stack.
There are two major vendors producing chips and development kits for this software. Secret Labs produces the netduino family, which consists of the standard netduino, a netduino plus which is an enriched version. This one has better processor and memory, it includes an ethernet port and uses micro sd cards to provide storage. One of the interesting things of this two boards is that the layout is compatible with the arduino shields. Secret Labs has another board called netduino go, which is similar to netduino plus but without storage and ethernet, and it does not use the typical arduino layout, so a globus module is required to use arduino shields. They also have another board called netduino mini whose size is similar to a rubber.
GHI Electronics is another hardware manufacturer that has designed and released different boards implementing Micro Framework or modules for which its target platform is Micro Framework. GHI has a very wide range of products, for example FEZ Cerbuino Bee and FEZ Cerbuino NET, which are similar to the netduino plus in terms of performance.

An interesting thing of FEZ devices over netduino is the possibility of loading native code (C/Assembly) for real-time requirements. For example HomeSense could run its Mesh driver in C and perform better than it performs using C#.

In addition to the mentioned manufacturers, Microsoft Research in Cambridge has defined a hardware reference platform called .NET Gadgeteer, which defines how boards and modules must be in order to allow rapid prototyping of projects. Gadgeteer boards and modules share the same layout and connector schemes and are open to any company that wants to build products using those schematics.

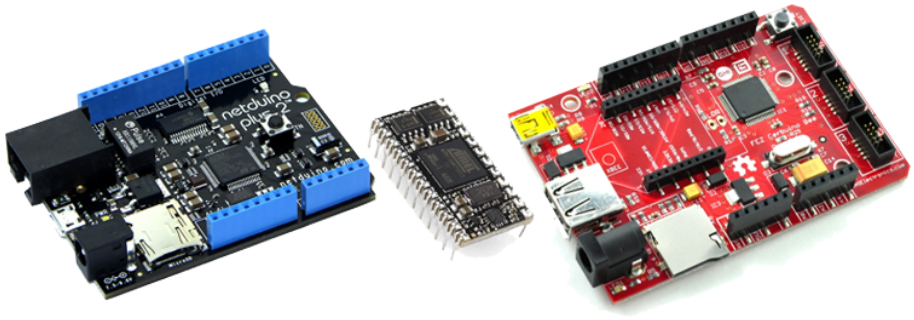

Figure 2.2: Micro Framework devices. Netduino Plus on the left, Netduino Mini on the center and Cerbuino on the right.

2.2.2 NETMF on Linux

After doing some research about implementations of Micro Framework on other devices, or running on top of other operating systems such as Linux, it was found a project that is currently porting NETMF to Linux, but for a very specific device called Eddy. This is a complete port of the virtual Machine so it has the original CLR interpreter.
Eddy is an ARM embedded device board that uses Linux. The port named above has been made as a demonstration of writing NETMF applications using a port on top of other operating systems. One of the majors problems of this port is that some drivers are not working at all, for example UART, SPI and I^2C which are 3 I/O protocols and ports used in HomeSense.
Although this port is for the Eddy board, it can be ported to other devices using the appropriate cross-toolchain. Anyway it seems that there is a lack of possibilities to run Micro Framework code in other devices or operating systems.

2.2.3 NETMF on RaspberryPi

If it is hard to find an implementation of NETMF in Linux, it will be harder to find an implementation for Raspberry Pi. In other words, people in GHI forums are asking for NETMF ports for the Raspberry Pi but no one exists. At time of writing this thesis, a small port was uploaded to Codeplex `http://raspberrypinetmf.codeplex.com/` and uses the bcm2835 library to implement the Micro Framework functions.

Leaving aside the NETMF implementations for the Raspberry Pi, a search on existing libraries to control the features of the board was done. There are existing libraries for many languages including C and C++, Python, Java, Ruby and .NET. The interesting ones are the C and .NET implementations, the first one because can be used in conjunction with .NET code via Platform Invocation Services. The second one is interesting for how has been done the implementation of the bcm2835 in this language, but after analysing it, the conclusion was that there is a lack of libraries written in .NET.

2.3 Conclusions

There is not any real implementation of .NET Micro Framework for Linux. If IOSharp succeeds on implementing a library with the same methods of the framework it will become the first library to execute Micro Framework programs on any standard computer that runs Linux. In other words, this library will make easier the interaction between .NET programs and the underlying hardware and I/O ports.

CHAPTER 3. IOSHARP

In this chapter it will be explained how was defined and architected, designed and implemented the core of IOSharp, disaggregating the different parts and explaining each one.

First of all, it will be explained the project design evaluating the two options at project definition, then the implementation is explained for the different I/O ports and protocol standards used in this project. Finally, it is explained how has been done the port mapping to work between different boards and devices.

3.1 Planning the development

At the beginning of the project, the development was focused on a tiny Linux board called Raspberry Pi. This device was designed by Raspberry Pi Foundation in England taking in mind the kids around the world introducing them in computer science. It is an interesting board for its features like basic I/O through GPIOs, SPI for serial peripheral communication, I^2C and UART interface also for transmissions between external components and the device itself.
In addition to the interfaces mentioned above, it also has some desktop interesting features like USB ports which, practically can accept any device that works on Linux (for instance a Wi-Fi, Bluetooth, ZigBee or any stick for wireless transmissions, HDMI for graphics and Ethernet for network communications). Apart from this I/O characteristics, it also mounts an ARMv6 (CPU) running at 700MHz on stock frequency along with 512MB of RAM, that is enough for normal desktop usage (surfing, emailing and office) and for embedded projects.

After choosing the target device, two implementation options were suitable for this project so each one was analysed. Each one has its own benefits and problems that are going to be explained in the following sections. Some options have been considered, as the use of a specific library designed for the Raspberry Pi which should provide high efficiency. Or develop the implementation in a way that could be executed in any device running a Linux Kernel which makes the project platform independent.

3.1.1 Focused on Raspberry Pi

Initially IOSharp was started focusing on the Raspberry Pi, so a search was done in order to find useful libraries for this project and one of the results was a native C library. This library gives control over all the features provided by the microprocessor integrated on the board which includes a wide range of GPIO pins, different protocol communications like SPI and UART, and other features like PWM.
This method is interesting when achieving high performance on the programs is important, so using the library the CPU features are used on a low-level way by using the CPU registers. This normally let the programs run faster but the programs only work on platforms using the same CPU, or in other words, is a specific hardware implementation.

The library written for the BCM2835 is the one that should be used in the case that a program has high performance requirements. The idea is to make calls from C# to this library using a specific call methodology that will be explained on future sections on this thesis.

3.1.2 Focused on Linux

Thinking on a more wider way it could be interesting to make IOSharp available to much more devices or even real computers. This implies more developers using this software which can provide useful feedback that is interesting for the improvement and evolution of the library. Any computer running some kind of Linux Kernel should be able to use this software facilitating the usage of the the hardware features.
The idea is to use C# combined with C to generate cross-platform assemblies, which making use of Mono, can run practically on any Linux device, but this has a little drawback related with the performance. Mono is a virtual machine which executes .NET code (C#), the implementation of this VM is not as good as the .NET framework on Windows so normally the performance of the programs running on it show some degradation.

3.1.3 Chosen implementation

Finally the chosen strategy was to use the Linux approach because it offers the appropriate tools like the `spidev.h` to manage the SPI or even the GPIO mappings through the SYSFS. To use any of this features the only requirement is that the necessary modules must be loaded into the Kernel. Along with the C# implementation, a C library will be written to interact with the functions provided by the Linux Kernel to control the SPI and the GPIOs. Micro Framework natively offers the required classes to configure the port mapping according to the pins and devices of the underlying hardware. In short, IOSharp will be able to run in any platform that uses the standard Linux Kernel such as the Raspberry Pi, a Cubieboard or even a standard desktop. This also has to be provided by IOSharp to map the Kernel devices into the specific hardware pins.

3.2 Implementation

In the following sections it will be explained how has been carried out the implementation of the IOSharp library. As a summary IOSharp works as Micro Framework but on a high level basis, so it is not a complete port of the original, instead of this it uses C# to implement the same functions which are exposed by the framework. In this case, the original source code has been downloaded from `http://netmf.codeplex.com/`. From the folder `\Framework\Core\Native_Hardware` can be obtained the necessary files to implement at least the GPIO and the SPI port along with the files for port mapping or even UART. The native files make use of internal implements, so when a method which does an internal implement, is called the virtual machine will take the call and process the function internally. IOSharp instead of implementing the virtual machine implements the functions as a normal method (opening and closing the brackets and inserting the code inside the

method).

The final implementation of IOSharp should look like the figure 3.1. The board uses Linux and then Mono is used to execute the Embedded Application. The application references IOSharp library which it currently has 4 features. The SYSFS is used by the GPIO to control the different I/O ports. The interrupt system and the SPI uses a custom library named IOSharp-C. Finally, the UART uses the SerialPort implementation from the Mono libraries.

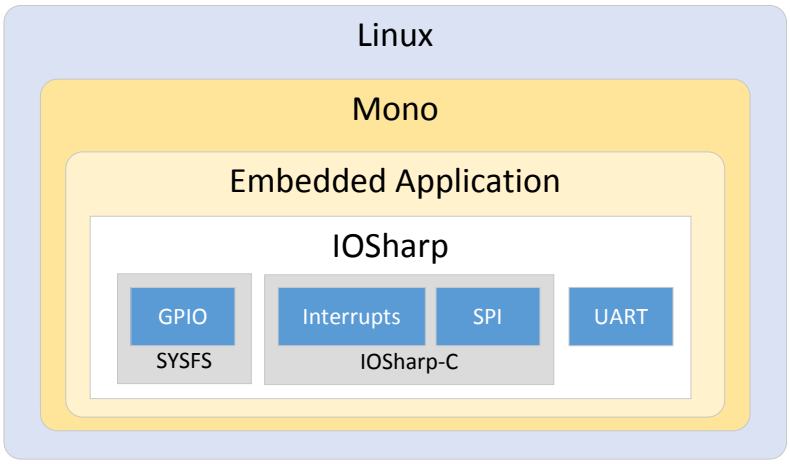

Figure 3.1: Representation of IOSharp

3.2.1 GPIO

GPIO acronym stands for General Purpose Input Output which are Ports on systems that are capable of generating an output or reading an input with a certain level of voltage. Normally embedded systems work at 3.3V, but low power devices can work at 2V.

3.2.1.1 Implementation Options

In order to implement the GPIO ports in Micro Framework, it will be used the `IOPorts.cs` file which contains the structure for Input, Output, Tristate and Interrupt ports. The first three ports will be explained in this section whereas the interrupt port will have a dedicated one.
The GPIOs in Linux can be controlled in several ways. The most common and simple is to use the SYSFS, which stands for a set of directories with readable and writeable files representing the ports of the CPU. On the other hand, the Linux kernel also provides a Kernel API to control the pins.
The decision must be done between these two systems. In order to use any of this solutions the GPIO module must be loaded into the kernel. Many desktop Linux distributions have GPIOs disabled and that's why the kernel must be recompiled enabling this feature.
In case of Linux operating systems designated for embedded devices, for example the Raspberry Pi or CubieBoard, they will have the I/O Ports enabled by default.
It is interesting to point that Android is capable to use GPIO ports. Although it cannot seem

an interesting feature nowadays, android is everywhere and can run in many devices, so this is another reason to try to fetch this sector in future versions of IOSharp.

3.2.1.2 Using GPIO from SYSFS

Since each solution can be used in this project and both are available in any Linux, it was decided to use the SYSFS access because it is much easier to use and test the implementation. It only requires having read/write access to a certain set of files and directories and by reading or writing in these ones is possible to change the port states.

As it was said before, in SYSFS the control of the GPIOs is carried by several files and directories located under the `/sys/class/gpio` directory. In this directory there are two files which are called `export` and `unexport`. The first one is used to enable a GPIO while the second one disables it. After enabling a GPIO, a new folder is created representing the enabled port. For instance, if port 2 is enabled, a folder called gpio2 will be created. Inside this new folder there are several files. The `direction` file describes how the port should work, if the desired function is an input port an `in` must be written in the file whereas `out` is used for an output port. After setting the port direction the next relevant file is `value` which is used to set the port state in case of an Output Port (write 0 for a state-low or a 1 for a state-high) or in case of an Input Port it will read the incoming value through the port.

3.2.1.3 Implementing in NETMF

Keeping in mind that this implementation has to be done over the existing code extracted from the `IOPorts.cs` file, it is important to design it properly. In this case, a `GPIOManager` has been created using a singleton pattern. This class will restrict the number of instantiations that a port can have in order to avoid problems on the hardware. The instance of this singleton is shared across all the code. This manager will be in charge of enabling, disabling and operating the different I/O ports. The figure 3.2 shows the UML diagram of the `IOPorts.cs` file from Micro Framework. Each class uses the `GPIOManager` to control the read/write functions and port creation.

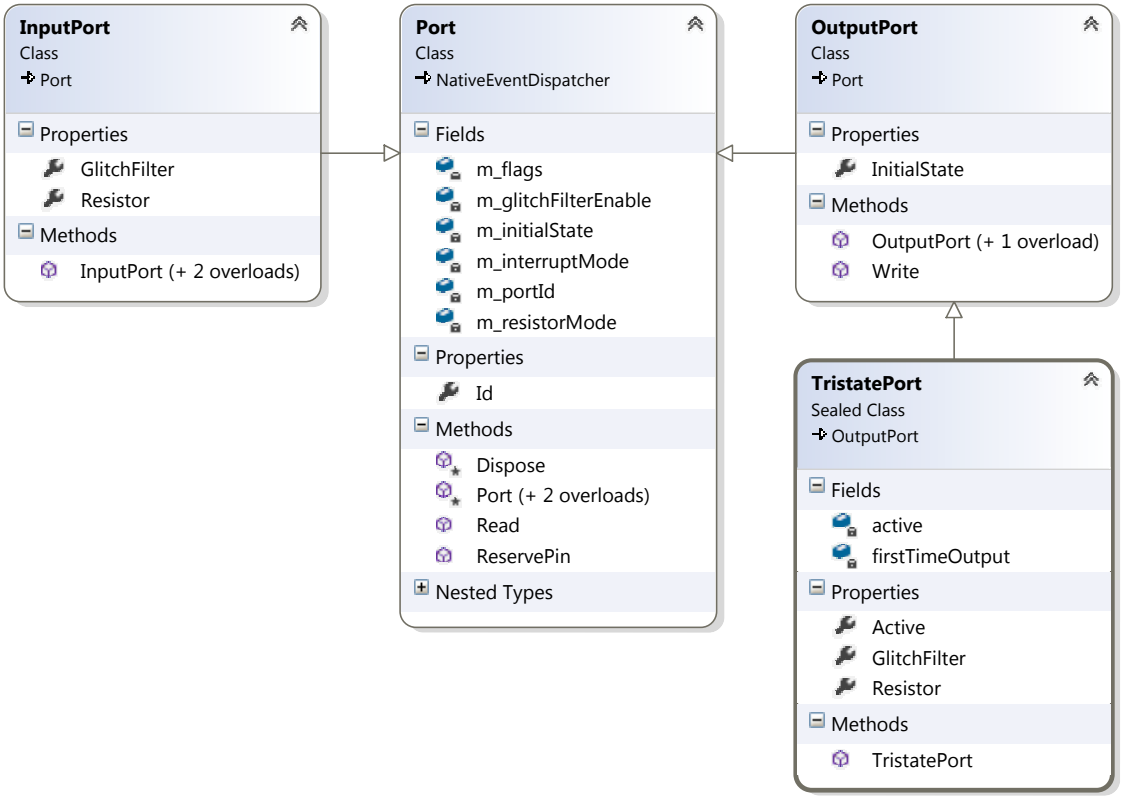

Figure 3.2: UML Diagram of NETMF Port and its inheritance

As it is shown in the figure, each port type inherits from the `Port` object which implements the methods to enable or disable a port. The `Read` method is used to obtain the current state of the port e.g read an input value or even know the configured state in an output port. Finally the `ReservePin` method permits to reserve a pin for a future usage.
The `InputPort` inherits from Port and it does not have any special method apart from its constructor which bases to the `Port` class.
The `OutputPort` which also inherits `Port` implements a new `Write` method which is used to write a state through the port (active-high or active-low).
The `TristatePort` inherits from `OutputPort`. This port can change its functional work between an input or an output mode.

3.2.2 Interrupts

Interrupts are required in the HomeSense program when using the SPI. Although the BCM2835 supports native interrupts via IRQ, at the time this project was developed the Raspberry Pi did not support GPIO interrupts using IRQ, so an alternate mechanism was designed.

3.2.2.1 *Designing the Interrupts*

The interrupt system has been written in C in order to use a function called `poll`, which is commonly used by developers that want to detect GPIO interrupts using the SYSFS. The `poll` function must be configured to block until certain events occur on a file descriptor corresponding to a GPIO port enabled on the SYSFS. This function is configured to wait and block the program execution until a `POLLPRI` event on the file descriptor is detected. This event will be triggered by the OS when the file has urgent data to be read. The poll function will detect the event and then will proceed with the following instructions to read different parameters from the port (i.e. the port state). Then, in C# a delegate pattern will be used to notify to the upper layers of the program that interrupts. The developer will configure the function which acts as the delegate by passing it to the `OnInterrupt` property in the Interrupt Port.

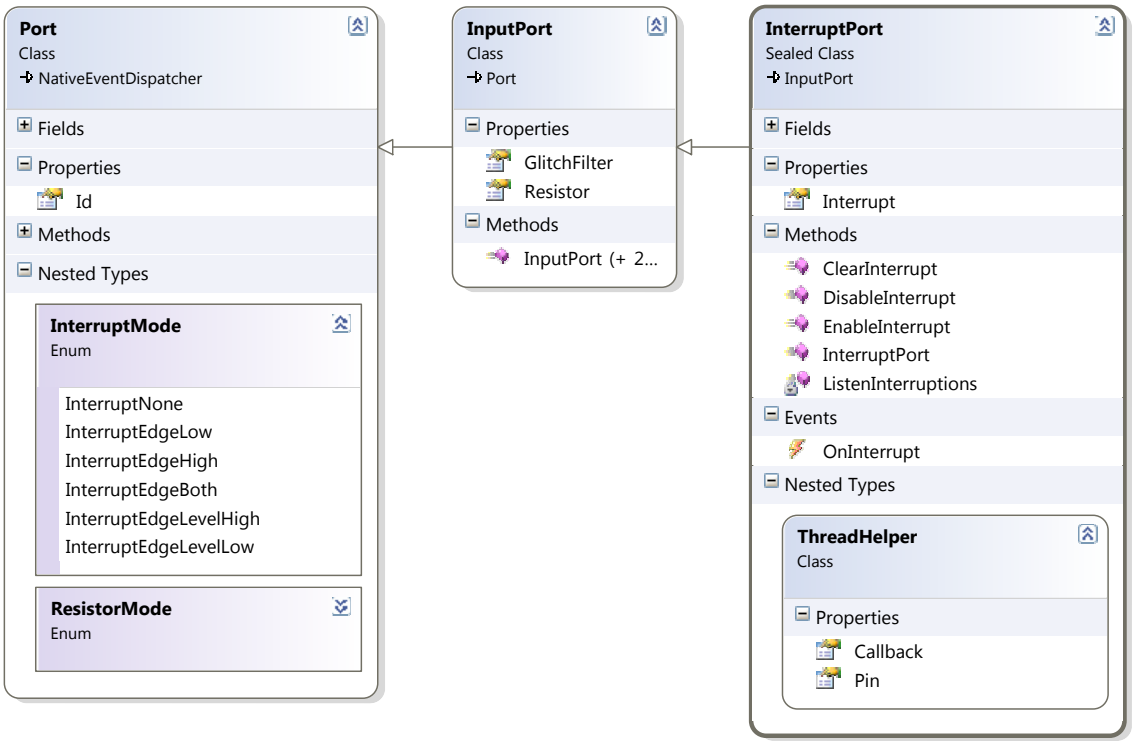

Figure 3.3: UML Diagram of NETMF Interrupt Port

3.2.2.2 *Platform Invocation Services*

In C# it is possible to invoke external libraries which are not written in the same language by using a mechanism called P/Invoke. In this case, the library used for the interrupts is written in C and it will be compiled into a shared library making it available to any program, or in this case, IOSharp. Take a look at the appendix A.1 to know more about the different library types.

P/Invokes in .NET make use of dynamic loaded libraries in order to use the contained functions. The implementation difficulty of a P/Invoke increases on how complex is the function to be called regarding its parameters. For basic type parameters such as `int`,

`long`, `byte` it is really simple to make a P/Invoke call, but when passing object parameters things get much more difficult because this requires doing marshalling for these objects. Marshalling is similar to serializing but maintaining some information related to the object. The marshalling is used to pass from managed to unmanaged code and sometimes it is impossible without using intermediate structs as interchange objects.

Below the important parts of the implementation of this library are shown together with how P/Invoke is done in C#. This first block shows the function which is used to detect the interrupts on the GPIOs. The interesting parts are commented explaining what they do or what some macros mean.

Listing 3.1: IOSharp.c - Polling function

```
uint64_t start_polling(int pin) {
    struct pollfd fdset;
    int nfds = 1;
    int gpio_fd, timeout, rc;
    char * buf[MAX_BUF], c;
    int len, count, i;
    long t;

    // Get the File Descriptor for the GPIO Port. See function on the Library.
    gpio_fd = gpio_fd_open(pin);

    // Clear any initial pending interrupts
    ioctl(gpio_fd, FIONREAD, & count);
    for (i = 0; i < count; ++i)
        read(gpio_fd, & c, 1);

    // Fill fdset which is a struct for pollfd which is used to describe the polling system.
    // In this case the File Descriptor for the GPIO port is entered, and then the POLLPRI (Data↩
        Urgent to Read) is configured as the event type.
    fdset.fd = gpio_fd;
    fdset.events = POLLPRI;

    read(fdset.fd, & buf, 64);

    // Start polling the File Descriptor. POLL_TIMEOUT variable contains (-1) which stands for ↩
        infinite blocking until event.
    rc = poll( & fdset, 1, POLL_TIMEOUT);

    // Close the GPIO Port. See function on the Library.
    gpio_fd_close(gpio_fd);
    return t;
}
```

After writing the function this must be exposed using a header file which is shown below.

Listing 3.2: IOSharp.h - Header file for the library

```
#ifndef IOSHARP_H_INCLUDED
#define IOSHARP_H_INCLUDED

// Define the polling function
uint64_t start_polling(int pin);

#endif
```

And finally this block represents P/Invoke is done in a C# program. The function is exposed using a `public static extern` and then an attribute is attached which corresponds to the `DllImport` which specifies the shared library to call.

Listing 3.3: GPIOManager.cs - P/Invoke section

```
// The function which calls the external function
private void Listen(object obj) {
    ThreadHelper th = (ThreadHelper) obj;
    while (true) {
        int pin = (int) th.Pin;
        // Call the function. See down.
        ulong cback = GPIOManager.start_polling(pin);
        th.Callback(4, (uint) 0, DateTime.Now);
    }
}

// External function represents a function on an external library, in this case the library is ↩
    the libIOSharp-c.so. The function naming and functions parameters are equal to the original ↩
    function, but taking into account that a ulong in C# is a uint64_t on C.
[DllImport("libIOSharp-c.so", CallingConvention = CallingConvention.StdCall)]
public static extern ulong start_polling(int gpio);
```

3.2.2.3 Final implementation

The program flow is shown on Figure 3.4 which represents the steps that IOSharp does for the interrupts system. The Interrupt Port inherits from Input Port as Figure 3.3 shows.

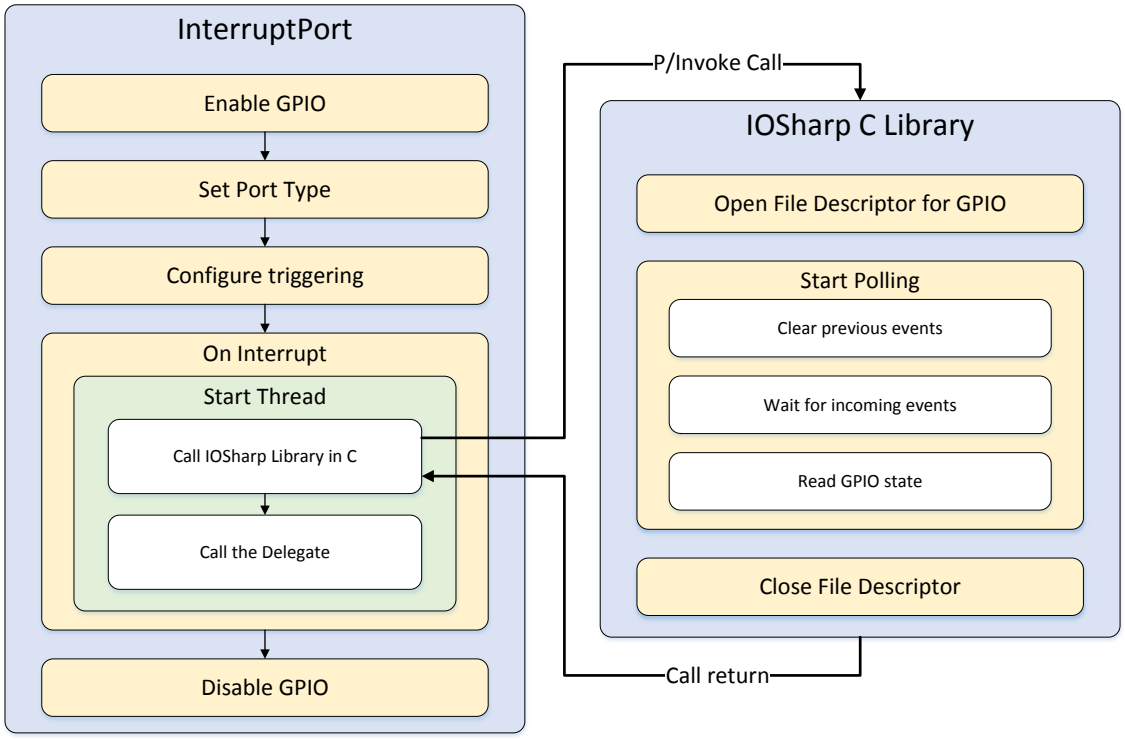

Figure 3.4: Representation of the Interrupt Port flow

The idea is to do the exact same steps the other ports do. The port enabling is done by the Port implementation which is the base class for the Interrupt Port. Then the port type is set to be an Interrupt Port. After doing this, the triggering must be configured. Micro Framework boards usually support four kinds of interrupts whereas Linux supports a few less. In the table 3.1 are shown the supported ones. It is important to know that the edges

are the point where the read state changes from one level to the other one. The Level is used when the state is maintained several time without changing.

Trigger Type	Micro Framework	Linux
InterruptNone	X	X
InterruptEdgeLow	X	X
InterruptEdgeHigh	X	X
InterruptEdgeBoth	X	X
InterruptEdgeLevelHigh	X	
InterruptEdgeLevelLow	X	

Table 3.1: Interrupt Trigger Types

Like the GPIO, the triggering is configured on the SYSFS on a file called `edge` located inside of the enabled GPIO folder. This file can be configured on three different ways. `EdgeLow` interrupts require the word *falling* be written in that file, in case of `EdgeHigh` *rising* is used and finally *both* is for `EdgeBoth`.
The polling thread works as an infinite loop P/Invoking to C and waiting for the return call, when this occurs the delegated function is called and this indicates that an interrupt event occurred.

3.2.3 SPI

The SPI is one of the important features to be implemented on IOSharp. This protocol is used in embedded systems to communicate boards and components in a Master-Slave way. It offers a full-duplex communication channel where the master and the slave can write and read at the same time. Normally this protocol accepts transmission frequencies in the range of 10 kHz to 100 MHz and in order to operate the master configures its clock using a frequency less or equal to the maximum frequency supported by the slave which wants to communicate.
Figure 3.5 shows how a SPI bus is. A master device can have several slaves attached to its ports. The minimal system is composed by three ports which are the bus itself, the MOSI is the channel where the Master device writes and from where Slaves read the written information. The MISO is the channel where the Master reads the information that the Slave writes. The SCLK is used to set the clock of the system between the Master and the Slave. Finally for each Slave it must be CS in order to select the slave that must active during the transmission.

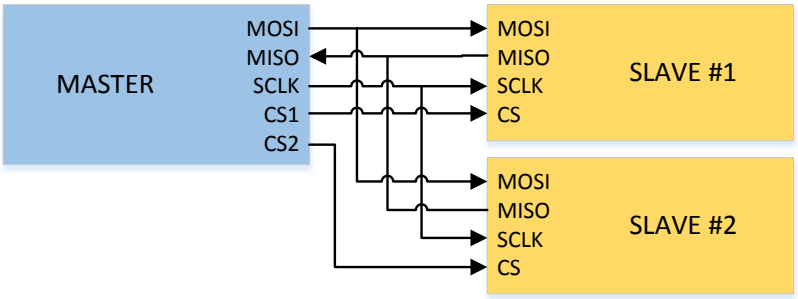

Figure 3.5: SPI bus setup with one master and two slaves

3.2.3.1 Designing the SPI

The SPI implementation has been carried out like the Interrupt Port explained in section 3.2.2. In this case, the Linux Kernel has been used by using the `<linux/spi/spidev.h>` library which makes the control of a SPI device very easy.
SPI devices are mapped under `/dev/` directory with a naming like `/dev/spidevX.Y` the `X` is an integer and represents the device (a CPU can have multiple SPI devices so this number will indicate the device number), then the `Y`, also an integer, enumerates the CS (SPI devices have multiple chip selects, so they can have more than one slave).

Before doing a transaction via the SPI this must be configured. First of all, we start by defining the operational mode. Modes are defined with the parameters Clock Polarity (CPOL) and Clock Phase (CPHA). Both are related to the sampling edge according to the clock (SCLK) used in the communication. The CPOL defines the polarity of the clock so the sampling will be done when the clock is in edge low or in edge high according to the configured parameter, CPHA defines in which phase the sample must be done. This concept is much easier to understand using the figure 3.6 which shows the different CPHA and CPOL options with the equivalent SPI modes required to be configured in the C library.

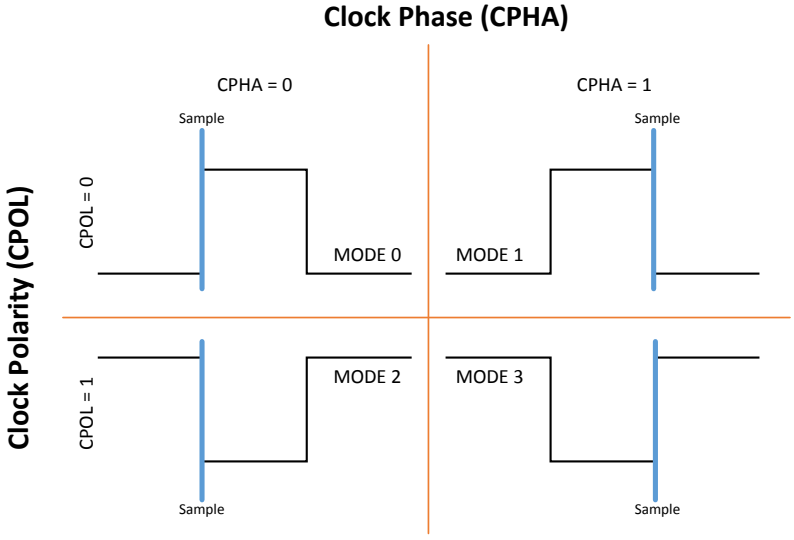

Figure 3.6: SPI modes are defined with the parameters "CPOL" and "CPHA" to the data sampling acording to the System Clock (SCLK) state.

These modes must be configured using the ioctl function passing the file descriptor of the SPI device and the desired CS. Then the preprocessor macro `SPI_IOC_WR_MODE` is used to specify which parameter will be configured. In this case, it is the SPI mode that has been explained before. The last parameter corresponds to another macro which defines the operational mode. The figure 3.6 shows each mode and the macro that must be passed to the ioctl function, this modes are `SPI_MODE_0`, `SPI_MODE_1`, `SPI_MODE_2` and `SPI_MODE_3`.

Listing 3.4: IOSharp.c - SPI Mode configuration

```
uint8_t mode;
  int ret;

//The mode variable can be SPI_MODE_0, SPI_MODE_1, SPI_MODE_2 and  SPI_MODE_3
  ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
  if (ret == -1)
    pabort("can't set spi mode");
```

Once the SPI mode has been configured, a struct defining the transaction must be filled. The struct type of is `spi_ioc_transfer` specified in the `spidev.h` library. This struct contains different variables. The `tx_buf` and `rx_buf` are configured with the write and read buffers. Apart from the buffers, the transmission length is configured using the variable `len`. Then the delay is configured, using the `delay_usecs` variable. This indicates how many microseconds the SPI driver must wait before starting the transmission. This is important because some slaves take some time from when they are selected until they are able to communicate. Another parameter to configure is the clock by using `speed_hz`. In order to deselect a device before a transfer, the parameter `cs_change` must be true. Finally, to configure or override the wordsize of a transmission the `bits_per_word` is used.

Listing 3.5: IOSharp.c - SPI struct configuration

```
struct spi_ioc_transfer tr = {
    .tx_buf = (unsigned long)writeBuffer,
    .rx_buf = (unsigned long)readBuffer,
    .len = writeCount,
    .delay_usecs = spi.delay,
    .speed_hz = spi.speed,
    .cs_change = spi.cs_change,
    .bits_per_word = 8,
  };
```

Once the struct is configured, another ioctl call is done which makes the transfer itself. In this case, along with the File Descriptor corresponding to the SPI device, another preprocessor macro is passed as a parameter. This one is called `SPI_IOC_MESSAGE` and must include the number of transfers that will be executed together. In case of IOSharp there is only one transfer per call, so the parameter will look like `SPI_IOC_MESSAGE(1)`. Finally the struct commented above is included in the ioctl call.

Listing 3.6: IOSharp.c - SPI transfer

```
// Pass the preprocessor macro and the spi_ioc_transfer struct.
ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
```

3.2.3.2 *Implementation in C#*

After writing the library part in C it is time to modify IOSharp to add the necessary calls to this library. In this case, the calls will be done in a similar way as it has been done in the interrupts (3.2.2). However, some data will need to be serialized in order to pass the configuration of the SPI from C# to C.
Essentially the method will be doing a P/Invoke from C# in order to call the functions in the C library. To facilitate the data exchange between the program and the library a struct is used. This contains the basic information in order to do the configuration explained on the previous section. This struct must be written in the header file of the library and also must be written in the C# code. The SPI implementation, in Micro Framework, is divided into two blocks. The first one represents the port configuration which is used to set the different properties that can be used with the SPI (for example the clock rate, the setup time of the slave, the SPI modes (CPHA and CPOL), etc). The second block, which is the SPI itself uses the configuration commented above to create an SPI instance, along with the required pins for the MISO, MOSI, SCLK and CS.
When the SPI instance is created, its methods will be able to be used. Basically, there are different overloads of the `Write` and the `WriteRead` methods, but all of them end calling the same internal function which will do the P/Invoke to the C library.

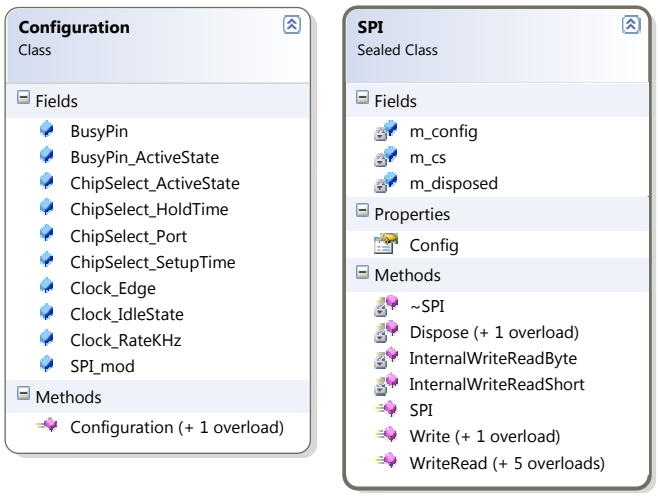

Figure 3.7: UML representation of the SPI Configuration Class (Left) and the SPI Port (Right)

In order to pass the configuration to the library, the configuration object is converted to a struct which its variables are the same as the struct from the header file of the library.
The code shown below corresponds to the header file and represents the struct that will be interchanged between the two languages.

Listing 3.7: IOSharp.h - spi_config struct

```
typedef struct spi_config
{
  int mode;
  uint32_t speed;
  int cs_change;
```

```
  uint16_t delay;
} SPI_CONFIG;
```

The listing below represents the C# implementation of the C struct used for the pin configuration. It contains the same parameters as the C version and also implements a constructor which simplifies the conversion between the Configuration class and this structure. It is important to note the attribute located over the method `[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]`. This attribute is used by the P/Invoke to know how the marshalling must be done when is passed to the C library.

Listing 3.8: SPI.cs - spi_config struct

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
public struct spi_config {
    public int mode;
    public uint speed;
    public int cs_change;
    public ushort delay;

    public spi_config(Configuration config) {
        this.cs_change = (config.ChipSelect_ActiveState) ? 1 : 0;
        this.delay = (ushort) config.ChipSelect_HoldTime;

        if (config.Clock_Edge && !config.Clock_IdleState)
            this.mode = 0;
        else if (!config.Clock_Edge && !config.Clock_IdleState)
            this.mode = 1;
        else if (config.Clock_Edge && config.Clock_IdleState)
            this.mode = 2;
        else
            this.mode = 3;
        this.speed = config.Clock_RateKHz * 1000;
    }
}
```

It is interesting to remark that the TX/RX buffers are not returned from C to C# , instead, their pointers are sent from C# to C, so the same memory is reused in both environments.

3.2.4 UART

UART is a really simple protocol that uses an asynchronous serial communication between two devices. As figure 3.8 shows, each device have two ports which are the TX for transmissions and RX for receptions. The transmission port must be connected to the reception port on the other device. Both devices must share the ground.

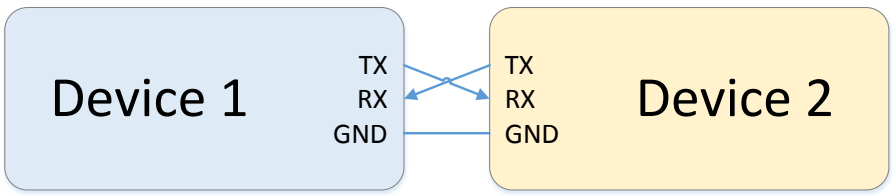

Figure 3.8: UART communication schema

Unlike the above features which were not implemented on the standard .NET Framework, the UART (SerialPort) exists on that implementation with exactly the same name and in the namespace. This is problematic in case of a class reimplementation is needed because it is impossible to maintain the original namespace (`System.IO.Ports`).
First of all and trying to avoid a new implementation of a SerialPort, the classes from the Micro Framework and the .NET Framework were compared. After doing this, it was realised that the two classes were so similar that IOSharp did not require a new reimplementation. The are two reasons to avoid this reimplementation, the first one is that any reuse of code is better rather than writing again the same feature, because .NET Framework is much more stable and tested than a code written from scratch. The other reason is that, although the .NET Framework class is not exactly as the Micro Framework class, it has all the required methods that are needed for HomeSense.

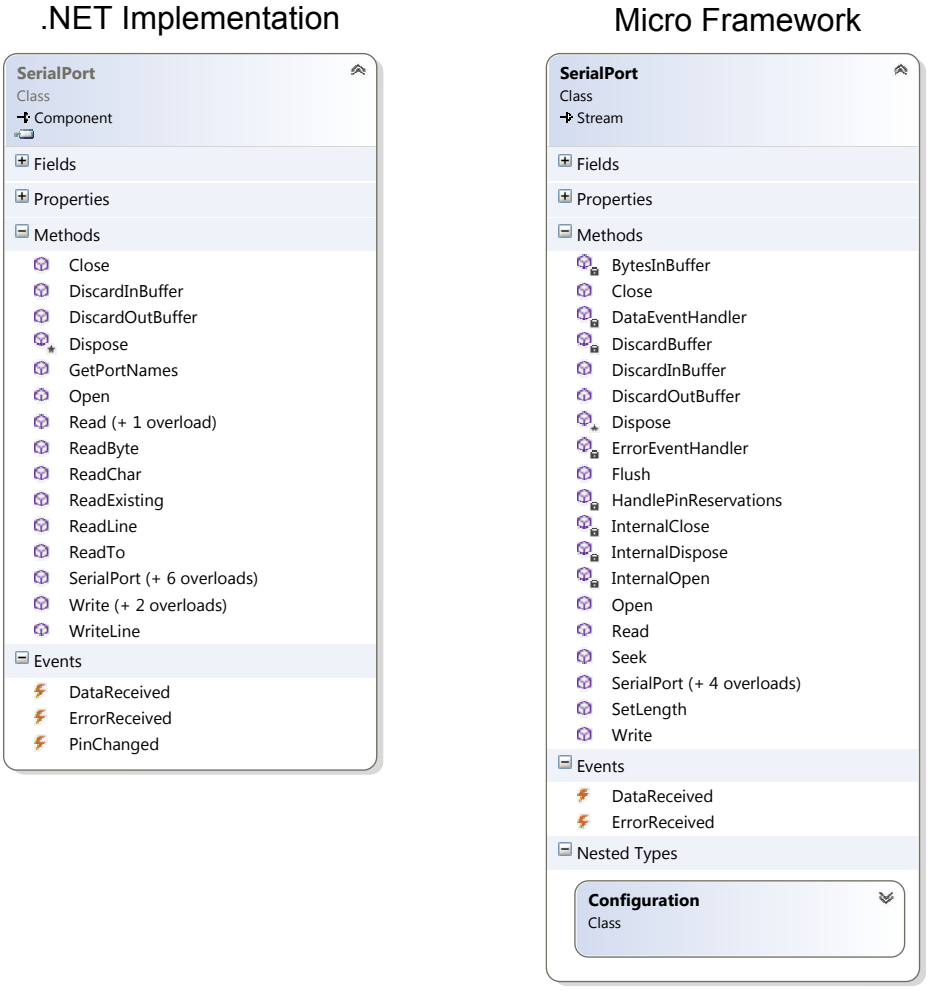

Figure 3.9: UML SerialPort representation of the original .NET Framework (Left) and .NET Micro Framework (Right)

It is important to remark that IOSharp, in Linux, runs on the Mono implementation of .NET Framework classes, and in this case the SerialPort class has some disadvantages on the Mono version, basically, it does not support the `DataReceived` or `ErrorReceived` events because that functions are not implemented on that virtual machine. But it is not a real problem because HomeSense does not use that events.

3.3 Port Mapping

Although IOSharp is a cross-platform library some features require a specific configuration when deploying on different boards. Every board has its own port mapping and naming, so it is necessary to have a specific library to describe that board. This is similar to the HAL and PAL concept:

- **HAL:** Hardware Abstraction Layer. In this case Linux acts as a HAL which offers simple APIs to use features likes the kernel driven SPI device or the GPIO exposed in the SYSFS.
- **PAL:** Platform Abstraction Layer. It is the library that must be implemented in order to exploit the HAL functionalities. In this case, the Raspberry Pi requires a library to map the GPIO pins or the SPI devices to the appropriate names for the HAL.

3.3.1 HardwareProvider

In fact, the original Micro Framework supports a hardware descriptor which represents the PAL of the specific boards and is called `HardwareProvider`. Raspberry Pi is the target platform for this project so a hardware descriptor was written. Below are represented the pins of the Raspberry Pi, on the left the revision 1.0 and on the right the revision 2.0. In the image are also shown where are the different pins used for the SPI and the I^2C.

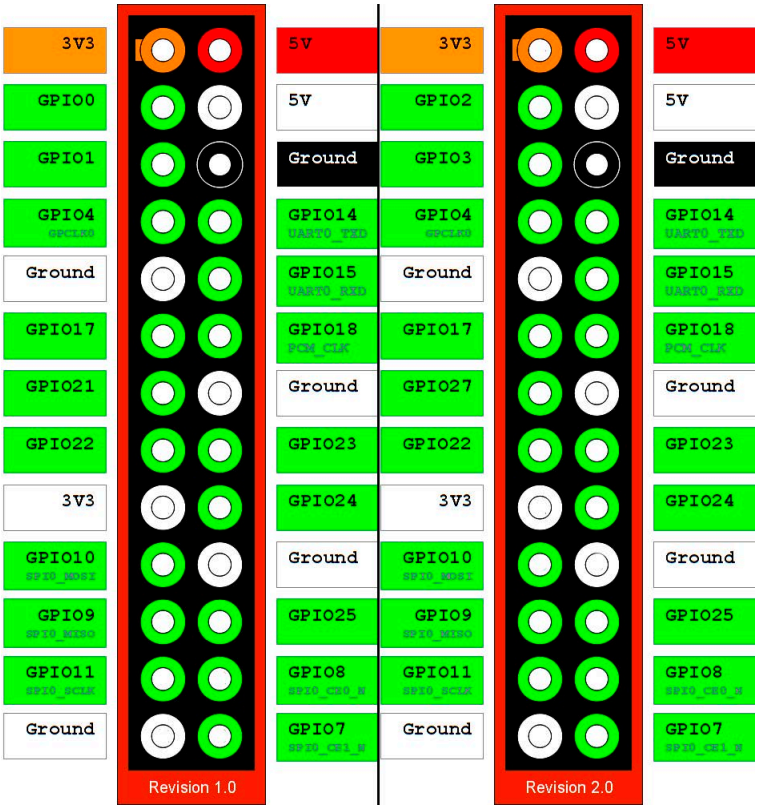

Figure 3.10: Raspberry Pi pin mapping

The mapping is divided into two different classes, the first one gives the Pin names that must be used on Linux to control the I/O ports. The second one, is related to the required pins to use a specific protocol. For example, it configures the pins for the SPI or the UART. In Linux the SPI and UART are known as devices so they are located under `/dev/` directory.

CHAPTER 4. FUNCTIONAL TESTS

Simultaneously with the development of IOSharp some small tests were created so the developed features could be tested to verify the properly operation of the implementation. Two functional test had been carried out in order to test the correct functioning of the library. The first one uses a Micro Framework project which consists of a RFID card reader. The second one is mainly the deployment of HomeSense.

4.1 SPI. RFID and IOSharp

This was the first test to verify the SPI in a real environment using a RFID card reader connected through the SPI bus. This test derives from a proof of concept made for VR gym in Argentona under the Spanish project "Activat" from "Proyecto Impacto". This project requested to develop a method to authenticate and authorize users for a secure entrance on a complex by using a RFID card reader and different cards, tags and wristbands with an ID which can be associated with a certain user.
The project was developed using a Netduino Plus which uses Micro Framework. The card reader used in this test mounts a MFRC522 chip from NXP which can use different communication protocols like SPI, UART and I^2C. But that chip is mounted on a MF522-AN board from Mifare which only offers a SPI interface.
An API was written in order to communicate the Netduino with the card reader, so the MFRC522 datasheet was used (see on Appendix A the section A.3.1). The program creates an instance of this API and then starts the SPI configuration. Following this configuration a Timer is instantiated which triggers a function at the scheduled cycle. The called function uses the API to communicate with the card reader so it can retrieve the card ID and then, with another tranmission, obtain the Serial Number located in the card. Finally it prints this data in the console.

4.1.1 Micro Framework version

The original example uses Micro Framework and the Secretlab classes for the Netduino Plus so the resulting binary will only work on this board. The port configuration for this example is really simple, because it only uses a pin for the CS. The MOSI, MISO and SCLK are defined by the board schema so the pins will be selected and configured internally on the virtual machine.

Listing 4.1: SPIApi.cs - Configuring SPI for the MFRC522 in Netduino Plus

```
public void ConfigureSPI() {
    SPI.Configuration xSPIConfig;
    Cpu.Pin pin = Pins.GPIO_PIN_D9;

    xSPIConfig = new SPI.Configuration(pin, //Chip Select pin
        false, //Chip Select Active State
        50, //Chip Select Setup Time
        0, //Chip Select Hold Time
```

```
        false, //Clock Idle State
        true, //Clock Edge
        1000, //Clock Rate (kHz)
        SPI.SPI_module.SPI1); //SPI Module
    spiDevice = new SPI(xSPIConfig);
}
```

As it is shown above, the CS pin is the `Pins.GPIO_PIN_D9` which corresponds to a digital pin.

4.1.2 Migrating to Linux

To use IOSharp instead of Micro Framework there is not a big requirement, basically the project must be converted to .NET Framework and then reference the IOSharp library. Beside this, the mapping classes according to the deployment platform must also be referenced. The next step is change the pin for the CS to the according one. Normally in Linux each SPI device have one or more CS pins but not every pin is suitable to work with that SPI device, so it is important to check the appropriate pin.
Taking in mind that using this test can also prove that IOSharp can work with the original code by doing a minimal set of changes, it was tried to use one of the features that the Visual Studio projects offers. Declaring two solution files (`*.sln`) which each one calls two different project files (`*.csproj`) make possible to have one solution with the Micro Framework classes for the Netduino while the other one contains the references for the IOSharp and the .NET Framework. This will create two different projects from the same code, one being able to run on Netduino and the other one in Linux.
These were the major changes, and they cannot be considered real changes to the original code, because is possible to take the application program and create a new .NET Framework project with that code. This changes proves that is possible to maintain the same code for different platform deployments.
Using conditional compiling is possible to set the required pin for each board and each solution. In this case, the symbol used for the conditional compiling is `MF` which is present on the Netduino version of the `*.csproj` whereas not in the Raspberry Pi. Taking a look into the above code, the Netduino uses `Pins.GPIO_PIN_D9` whereas the Raspberry Pi uses `Cpu.Pin.GPIO_Pin9`.

Listing 4.2: SPIApi.cs - Conditional compiling symbol for NETMF and IOSharp

```
public void ConfigureSPI() {
    SPI.Configuration xSPIConfig;
    Cpu.Pin pin = Cpu.Pin.GPIO_NONE;
    // In this case, the conditional compiling symbol used is MF, true for Micro Framework or ↩
        false for IOSharp
    #if MF
      pin = Pins.GPIO_PIN_D9;
    #else
        pin = Cpu.Pin.GPIO_Pin9;
    #endif

    xSPIConfig = new SPI.Configuration(pin, //Chip Select pin
        false, //Chip Select Active State
        50, //Chip Select Setup Time
        0, //Chip Select Hold Time
        false, //Clock Idle State
        true, //Clock Edge
```

```
            1000, //Clock Rate (kHz)
            SPI.SPI_module.SPI1); //SPI Module
        spiDevice = new SPI(xSPIConfig);
        //MFRC522Init();
    }
```

After doing this, this project can be opened as Micro Framework in order to deploy in a Netduino or open the Linux version. Deploying this application in any of these boards will result in a working program like the figure 4.1 shows.

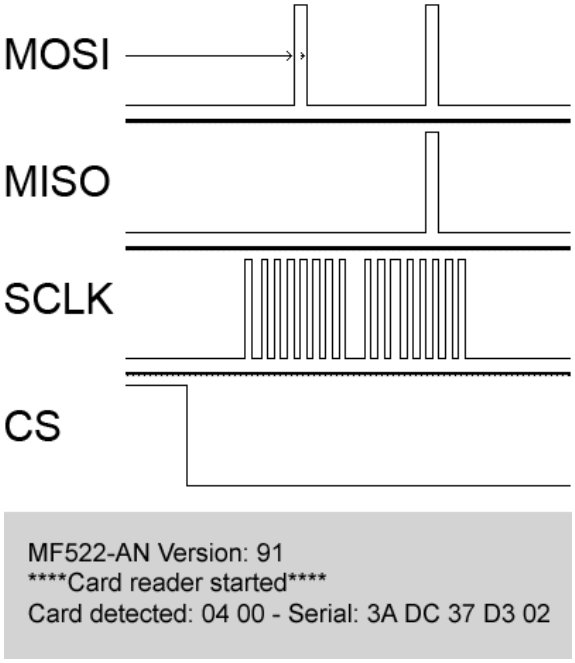

Figure 4.1: Data exchange using the SPI

4.2 HomeSense

HomeSense is a Wireless Healthcare Sensor Platform designed by AlterAid. The current implementation uses a Netduino Mini board which is the gateway. This board controls the sensor network, receiving all the data and uploading it to internet. Some sensors are spread around the house which are used to fetch data. With all of those sensors, it is possible to acquire information from the environment and then send that information to internet using the gateway. These sensors are programmed using C and all of them use the SoC nRF24LE1 which mounts a low-power RF ISM band (2.4 GHz) from Nordic Semiconductor.

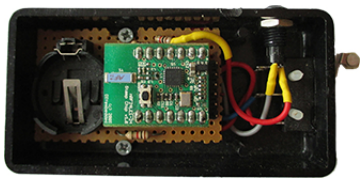

Figure 4.2: Sensor for a medicine cabinet

The communication protocol designed for HomeSense is similar to a star network with multi-hop transmissions so it becomes a tree-start topology. The nodes try to fetch the gateway because this is on charge of upload the information to the internet.

4.2.1 Gateway

The program running on the gateway is coded using Micro Framework and is deployed on a Netduino Mini. Using IOSharp has been possible to deploy it on a Raspberry Pi running Linux and Mono. The software running on the gateway controls the communication hardware and also the mesh network using the designed protocol for this platform.
As it uses all the features developed for the library it is a good functional test to try the whole system.

- **UART**: the Serial Port is used to transmit data between the Wifly and the Raspberry Pi. The Wifly is an XBee socket type module created by Roving Networks, and is used to connect to Wi-Fi networks. The communication between this module and the board where is connected is carried out using UART with a protocol described on the specification paper provided by the manufacturers.
- **SPI**: The Nordic nRF24L01+ is controlled using the SPI protocol. This chip is used to create the physical meshed network which connects with the different nodes. The channel created using the Nordic is half-duplex so there is only one communication at a time either transmission or reception.
- **Interrupts**: When the gateway's Nordic receives information form other nodes it must alert that new data is waiting to be read, to make this alert an interruption is used on a GPIO

Figure 4.3 summarizes the connections and devices used with the Raspberry Pi and HomeSense. On the left is shown the Nordic which uses the SPI protocol and it send interrupts to the Raspberry Pi. On the right, the Wifly uses the UART protocol to exchange data.

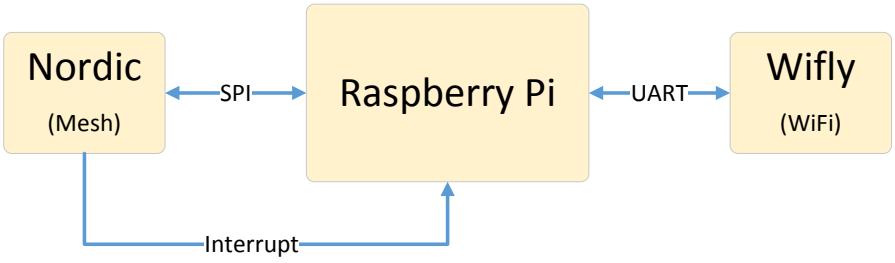

Figure 4.3: Diagram showing the protocols and devices used in HomeSense

With all of these devices attached to the shield used in the Raspberry Pi it will be possible to test all the functions working at the same time and how the system response is compared to the original one.

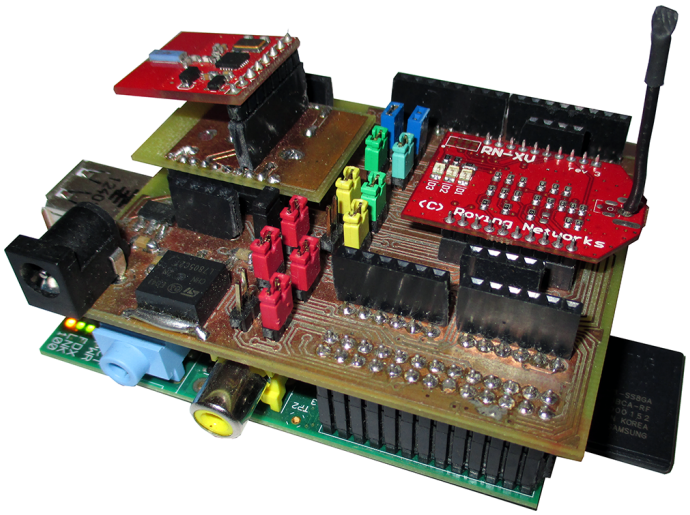

Figure 4.4: Raspberry Pi with the modules needed to run HomeSense

4.2.2 Working

As IOSharp has implemented all the features required in HomeSense, it is possible to change the project type from Micro Framework to .NET Framework and then reference this library. Finally, the ports need to be remapped according to the shield used with the Raspberry Pi. The strategy to do this mapping is include the Hardware class of the Raspberry Pi and then rename the ports to the according ones. SerialPort device also must be renamed from `COM1` to `/dev/ttyAMA0`. After changing the references and the port names is possible to start testing the program. So it has been really easy to migrate from the Netduino Mini board to the Raspberry Pi.

HomeSense has a program which shows the network status, showing its gateway and the different sensors connected to the network. On the following figure a connection is shown. The aaaida logo represents the gateway whereas the node is represented by a door. In this case, both are linked as an arrow connects both of them. With this program is possible to know the network addresses of both devices. Also it shows other information like battery state or the fetched information by any sensor located on a node.

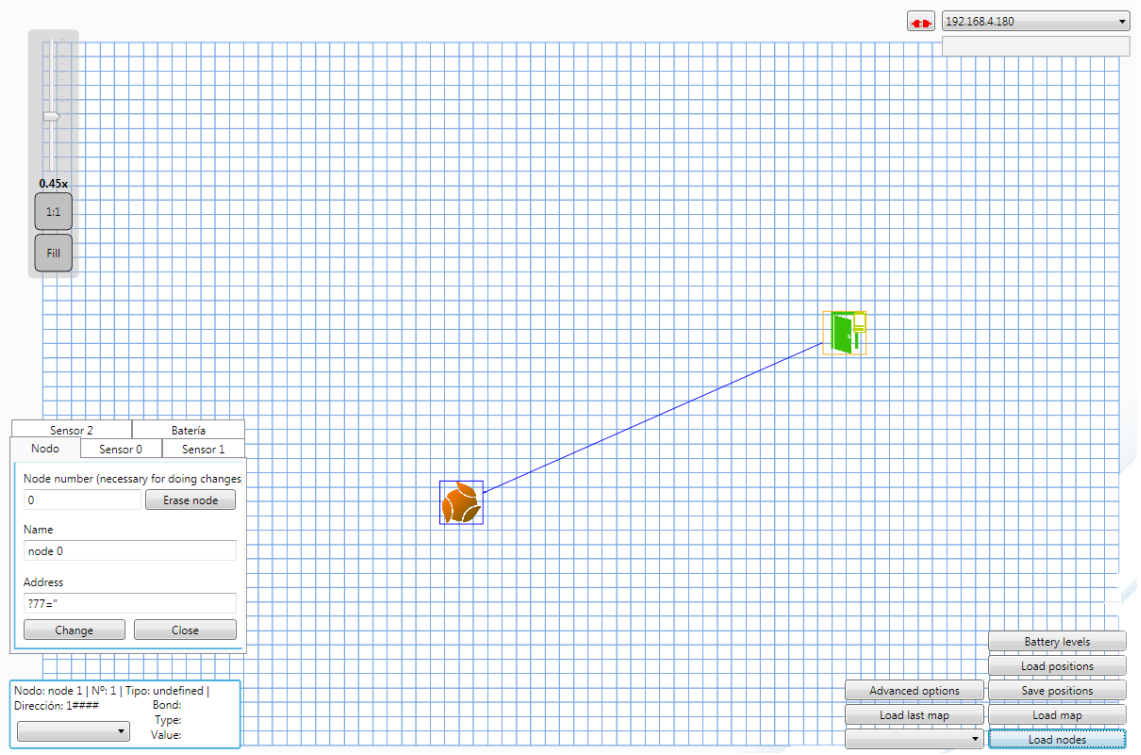

Figure 4.5: HomeSense dashboard. The aaaida logo on the left represents the Raspberry Pi (gateway) whereas the door represents a sensor

4.3 Conclusions

All the functional tests show that IOSharp has been successfully done. Each test works on the Raspberry Pi with doing only minor changes on the code. With IOSharp working and being able to be used on existing Micro Framework projects it is possible to make them run on other platforms, so the development milestone has been done. But, after testing the different parts of IOSharp and deploying HomeSense on the Raspberry Pi it was seen that the performance was not as good as it was expected. Raspberry Pi, need more time to send the network events than the Netduino and this is probably related to the time needed to attend the interrupts on Mono. In this case the IOSharp with Mono needs two transmissions to send the same information that is sent with only one transmission in the Netduino. To try to solve these issues with the performance of the library, IOSharp will be translated to C++ using a translating tool called AlterNative which is capable of translate .NET assemblies to C++ while maintaining a similar C# syntax.

CHAPTER 5. ALTERNATIVE

This chapter is a walk-through AlterNative explaining its concept, how it can translate the code, then an example using IOSharp with some use cases that can be applied to this tool.

5.1 Concept

The concept of AlterNative is to maximize the idea of Internet of Things by providing a tool to port applications from high-level languages (such as .NET) to native languages (such as C++) easily. Most of the actual systems are C++ compatible, thus if the application is ported to this language, it can be executed in several platforms (i.e. smartphones, tablets, embedded systems, computers with different operating systems).
With this tool a developer can take the advantages of fast developing in a high-level languages such as C# and then gain the advantage of performance related the low-level languages like C++. Apart from this, it also gives the chance to get native code capable of working in several systems, in other words, this philosophy is similar to the WORA (Write Once, Run Anywhere) slogan created by Sun Microsystems to illustrate the cross-platform benefits of Java Virtual Machine. The difference is that AlterNative is focused on the final performance because it outputs the code in native language avoiding the dependence of underlying virtual machines.

5.2 Process

AlterNative process is divided in three steps: decompilation, translation and recompilation. To summarize the following sections the figure 5.1 shows the process that is done from the original assembly, the decompilation, translation and finally the recompilation to an another assembly.

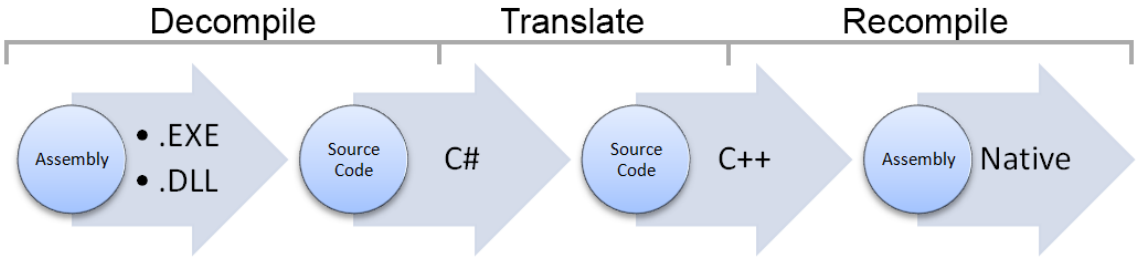

Figure 5.1: AlterNative process

To explain the AlterNative process will be used the `Main` method which both languages make use of it. In the following table is shown how this method looks like in each language.

Language	Method	AST
C#	`void Main(string[] args){}`	Figure 5.2
C++	`void Main(string args[]){}`	Figure 5.3

Table 5.1: Representation in C# and C++ of the Main method of a program

5.2.1 Decompilation

First of all the assembly is passed through a decompiler in order to extract the source code. In this case, the original code is in C# and to decompile it is used a program called ILSpy (an open-source .NET assembly decompiler). Instead of extracting the code in text format is extracted as an AST (Abstract Syntax Tree) that is an abstract representation with nodes and hierarchies of the original code. This representation is organized in a tree format from the top-level (assembly) until the low-level (instructions, types and constants).

The figure 5.2 shows how `void Main(string[] args){}` method would look like in AST format. The top node designates that the child nodes correspond to a method, then in the first row child is described the primitive type (what method returns) which in this case is a `void`, the identifier which shows the method name (`Main`) and finally the parameter declaration which describes the parameters that the function takes. The example method has one parameter corresponding to an array of strings. The parameter declaration node has two nodes defining it, the first one describes that `string[]` is a composed type which is also defined by a string and an array specifier, the second node is the identifier name of the parameter, in this case, `args`.

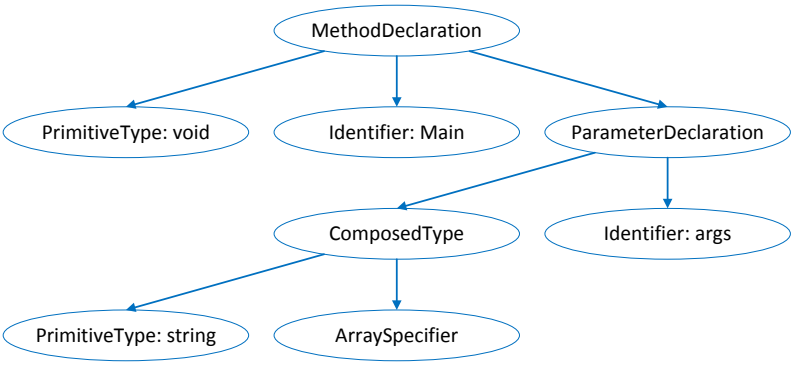

Figure 5.2: AST representation of the main method in C#

The translator will generate the C++ code from the AST representation of the original code.

5.2.2 Translation

To proceed with the translation, it is important to know how is the translated language typed, then in order to achieve that, some modifications need to be applied to the generated AST in the decompilation step. After doing the pertinent AST modifications a second

one will be obtained representing the source code of the desired language. After doing this conversions, the following step is start writing the files containing the text representation of the tree. AlterNative translates the code to C++ so the AST changes must be done in a way that the resulting AST corresponds to that language. It is important to remark that C++ is language that provides source and headers, so both need to be written to files.

Following the example explained in the previous section, now the AST will be transformed to an equivalent AST but for C++ language. In C# the choosen method looks like `void Main(string[] args){}` but in C++ the syntax is different because the array specifier moves from the primitive type to the identifier. The figure 5.3 shows in red the changes done to the original tree in order to achieve an AST that looks like `void Main(string args[]){}` (C++ method). The original identifier changes to a composed identifier with two child nodes, the first one is the identifier `args` and the second one is the array specifier which has been moved from the original composed type to the composed identifier in this case. As there is no composed type now the node is deleted and the parameter declaration is directly linked to the primitive type string.

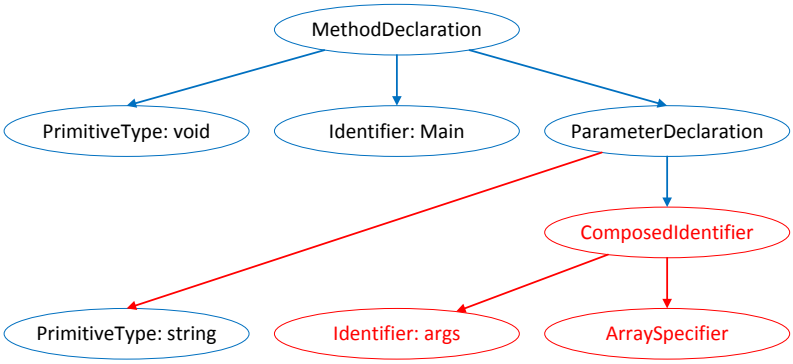

Figure 5.3: AST representation of the main method in C++

5.2.3 Recompilation

The final step consists on compile the C++ code into a new assembly. This final assembly maintains the same functionalities of the first one but taking the benefits of the performance that gives native code. Although it could seem that this software is focused on the code translation it is not its real finality because its aim is to maintain all the features of the original code like the garbage collector, specific expressions or even language syntax. Maintaining this characteristic on mind the programs translated using AlterNative will run much faster than the original because a virtual machine is not required to execute native code.
As mentioned, AlterNative is able to provide most of the features of the original code to the final code using external open-source libraries like the boehm GC for the garbage collector, the boost library for implementing characteristics like threading and delegation and finally a proprietary library which implements all the `System` namespace of C#. With all of this the final assembly is fully compatible with any system like Windows, Linux, Android or any device capable of execute C++ assemblies.

5.3 Use Cases

There are two clearly use cases of AlterNative together with IOSharp, the first one is performance and the second one is cross-platform for embedded systems.

5.3.1 Performance

AlterNative gets the major advantage when it is applied to programs or libraries which are very complex in a computational way (for example image processing, mathematics or complex algorithms) where languages running on virtual machines are not as fast as the developer needs. The more complex the target is, the more benefit that can be obtained. Taking into account that the performance seen on IOSharp running on Mono is not as good as it should be compared to the native Micro Framework running on a Netduino. It is supposed that passing a program created with IOSharp through the translator a faster binary will be generated (the virtual machine is not longer needed). It is known that Mono implementation is not as fast as .NET Framework (Windows).

5.3.2 Cross-Platform in embedded systems

Other advantage of the standard low level languages like C++ is that a high percentage of device processors can execute this code. And at this point is where it fits with the implementation of IOSharp because translating the library with AlterNative a new one will be obtained but instead of being written in C# it will be in C++. By doing this, the generated source can be compiled into a assembly, the unique requirement to execute the resulting code is a Linux running on the deployment machine. The developer will get all the benefits of Micro Framework with the speed of C++. The code will be written using the Micro Framework syntax which makes easy the development of embedded applications, and then they could use AlterNative to gain performance or achieve more platforms.

5.4 Contributions to AlterNative

To achieve the translation (or partial translation) of IOSharp through AlterNative, some work was done on the proprietary implementation of `System` libraries which are the C++ libraries that externally look like the C# ones but the methods are implemented using standard C++ functions or third party libraries like boost or Boehm GC.

At the beginning, AlterNative only worked on Windows because some classes from the ILSpy had mixed the view and the decompiler functions, so it was unable to run in Linux or MacOSX because of the non-compilable parts. In order to solve that, an AlterNative.Core was created which could work in any system able to run C# programs. This core version has its own solution and csproj file but they are pointed to the same code files of the original program. In this way, AlterNative can run either on .NET Framework in Windows or on

Mono in Linux and MacOSX.

The first attempt to translate IOSharp was unsuccessful, but it helped to determine how near the translation was at being successful. There where several major features that AlterNative was not able to translate like the threading, the delegates used in the interrupts, the P/Invokes or some functions like DateTime, Timer and file reading/writing. Many of those features where required to be implemented on the proprietary library because this is the one that is linked to the translated program.

To implement this features, the C++ boost library has been used. This uses standard C++ functions so it can work practically in any device or operating system. The classes currently implemented either totally or partially are shown in the Appendix A.4. Where marked in red are the libraries that where contributed to the AlterNative in order to translate IOSharp. Essentially the written ones are:

- **DateTime:** used for storing the time stamp of the interrupt.
- **TimeSpan:** to be used on the Timer.
- **File:** used to write or read values in the SYSFS in order to control the GPIO.
- **Encoding:** used to properly encode strings to UTF-8.

In the figure 5.4 is shown the differences between the original Micro Framework, the original IOSharp and the translated one. Each one includes some new layers on the stack, either to implement the functionalities of the Micro Framework as IOSharp layer does. Or to replace Mono as AlterNative System library does. The first one shows the original implementation which runs on the bare metal of the board as it has been explained on the state of the art. The second column represents the IOSharp original implementation, in this case, the underlying hardware has an operating system (Linux) and then it uses a virtual machine (Mono) to run the program that makes use of IOSharp. Finally, the translated version swaps Mono to AlterNative System classes and it also contains the C++ translation of IOSharp and the program, but it maintains the original IOSharp-C library.

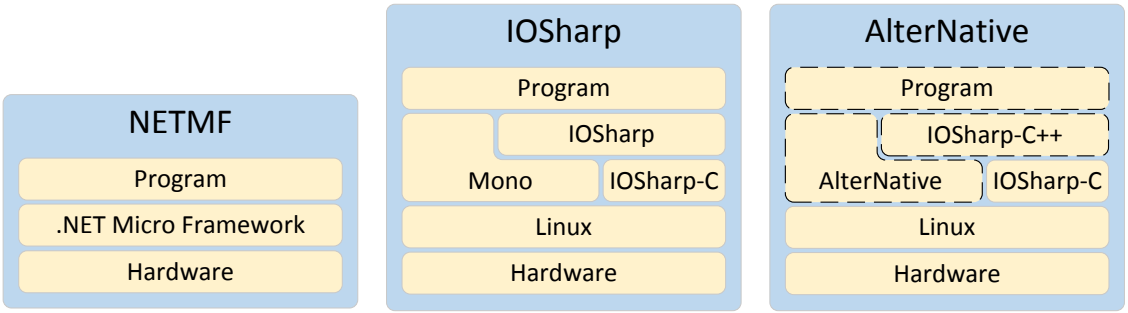

Figure 5.4: Stack in the original .NET Micro Framework, the IOSharp implementation and the AlterNative translation

5.5 Example

The best example to show in this section will be some part of IOSharp translated to C++. In the moment of writing this thesis the only component being 100% translated automatically was the GPIO ports, which basically they make use of SYSFS and the interrupt port which uses cross-language calls.

As a short example of this translation, the following pieces of code represents the InputPort in two different languages. This first one is the original implementation of IOSharp as it can be downloaded from GitHub while the second one is the translated representation.

Listing 5.1: InputPort in C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.SPOT.Hardware;
using IOSharp.NETMF.RaspberryPi.Hardware;
using System.Threading;

namespace raspberrypi
{
    public class InputPort : Port
    {
      // As it has been explained in the IOSharp seccion, all the GPIO have an inheritance with ↩
          the Port class, so all of them base to Port.
        public InputPort(Cpu.Pin portId, bool glitchFilter, ResistorMode resistor)
            : base(portId, glitchFilter, resistor, InterruptMode.InterruptNone)
        {
          // In this case, using the GPIOManager is configured the port as an InputPort
            GPIOManager.Instance.SetPortType(portId, PortType.INPUT);
        }

        protected InputPort(Cpu.Pin portId, bool glitchFilter, ResistorMode resistor, ↩
             InterruptMode interruptMode)
            : base(portId, glitchFilter, resistor, interruptMode)
        {
            GPIOManager.Instance.SetPortType(portId, PortType.INPUT);
        }

        protected InputPort(Cpu.Pin portId, bool initialState, bool glitchFilter, ResistorMode ↩
             resistor)
            : base(portId, initialState, glitchFilter, resistor)
        {
            GPIOManager.Instance.SetPortType(portId, PortType.INPUT);
        }

        public ResistorMode Resistor { get; set; }

        public bool GlitchFilter { get; set; }

    }
}
```

After translating the project the code is represented in C++ which uses header files, so the following code represents the header for the InputPort.

Listing 5.2: Header file for the InterruptPort in C++

```
#pragma once
```

```
#include "System/System.h"
#include "Port.h"
#include "Cpu.h"
#include "GPIOManager.h"
#include "PortType.h"
#include "System/Exception/SystemException/NotImplementedException.h"

using namespace Microsoft::SPOT::Manager;
using namespace System;
namespace Microsoft {
  namespace SPOT {
    namespace Hardware {
      class InputPort : public virtual Port, public virtual Object
      {
        public:
          Port::ResistorMode getResistor();
        public:
          void setResistor(Port::ResistorMode value);
        public:
          bool getGlitchFilter();
        public:
          void setGlitchFilter(bool value);
        public:
          InputPort(Cpu::Pin portId, bool glitchFilter, Port::ResistorMode resistor);
        protected:
          InputPort(Cpu::Pin portId, bool glitchFilter, Port::ResistorMode resistor, Port::↩
              InterruptMode interruptMode);
        protected:
          InputPort(Cpu::Pin portId, bool initialState, bool glitchFilter, Port::ResistorMode ↩
              resistor);

        // The C# properties in C++ are automatically converted to variables
        public:
          Port::ResistorMode Resistor_var;
        public:
          bool GlitchFilter_var;
      };
    }
  }
}
```

Finally this is the file containing the implementation of the header.

Listing 5.3: Implementation of the InterruptPort in C++

```
#include "InputPort.h"
namespace Microsoft {
  namespace SPOT {
    namespace Hardware {

      // So the converted variables need to be accessed using getters and setters. Like is shown↩
          below
      Port::ResistorMode InputPort::getResistor(){
        return Resistor_var;
      }
      void InputPort::setResistor(Port::ResistorMode value)
      {
        Resistor_var = value;
      }
      bool InputPort::getGlitchFilter()
      {
        return GlitchFilter_var;
      }
      void InputPort::setGlitchFilter(bool value)
      {
        GlitchFilter_var = value;
      }
      InputPort::InputPort(Cpu::Pin portId, bool glitchFilter, Port::ResistorMode resistor) : ↩
          Port(portId, glitchFilter, resistor, Port::InterruptMode::InterruptNone)
      {
```

```
        GlitchFilter_var = (bool)(0);
        Resistor_var = (Port::ResistorMode)(0);
        GPIOManager::getInstance()->SetPortType(portId, PortType::INPUT);
      }
      InputPort::InputPort(Cpu::Pin portId, bool glitchFilter, Port::ResistorMode resistor, Port↩
          ::InterruptMode interruptMode) : Port(portId, glitchFilter, resistor, interruptMode)
      {
        GPIOManager::getInstance()->SetPortType(portId, PortType::INPUT);
      }
      InputPort::InputPort(Cpu::Pin portId, bool initialState, bool glitchFilter, Port::↩
          ResistorMode resistor) : Port(portId, initialState, glitchFilter, resistor)
      {
        throw new NotImplementedException();
      }
    }
  }
}
```

Some performance tests had been carried out to quantify how much faster is the translation compared with the original one running on Mono. The explanation of the tests and its results are on the next chapter.

CHAPTER 6. PERFORMANCE TESTS

In this chapter are shown the performance results for two tests carried out on a Raspberry Pi using the original implementation of IOSharp, and the C++ version, which is a direct translation using AlterNative.
Theoretically, C++ has a major performance compared to C# but this tests will be used to determine how much gains C++ over C#.
The Logic16 has been used to make the timing measurements. This is a channel analyser used to record, view, and measure digital signals.

6.1 Compilation types

When a compiler generates a binary from a source code normally tries to do some changes to optimize some attributes of the program. The most common requirement is to optimize the time taken to execute that program, another one is optimize the amount of memory required by the program. Also, some optimizations can be used to make a program consume less power and this is interesting because nowadays the Internet of Things is growing and many sensors have a small battery so reaching low-power consumption is great because it ensures a longest battery life or at least the sensor can operate more time. All of this optimizations are carried out by a sequence of optimizing transformations and algorithms which are applied by optimizing compilers.
Normally compilers can generate programs optimized or non-optimized, depending on how is configured the build system. If the compiler uses optimizations the compile time will grow but the program will be much more optimized than if optimizations are not applied.

- **Non-optimized or debug:** this mode is used by developers who want to debug applications in execution time. In this case the whole symbol information, which is used by the debugger to stop at the break points (designated instructions), is attached to the generated assembly. For example the `*.pdb` files from Visual Studio are created by the compiler and have the information to debug the created assembly. On the other hand, the debug mode will not allow some optimizations because are incompatible with the debugging functionality.
- **Optimized or release:** this mode is used to generate an optimized assembly to run or perform much faster than the debug one. In this case, the compiler performs different transformations to the original code, for example two typical optimizations that cannot be applied to a debug build are:
  - **Loop unrolling:** the compiler analyses how many times the loop is executed and then it copies the inner code the same number of times. This helps avoiding the maintenance of the loop variables.
  - **Inlining:** the compiler places the method on the place of the call avoiding the stack overhead produced by a method's call.

Listing 6.1: Inline example

```
inline int Max(int x, int y)
{
   return (x > y)? x : y;
}
int main( )
{
   int a = 100;
   int b = 1010;
   cout << "Max (a, b): " << Max(a, b) << endl;
   return 0;
}

/* The Max(int, int) function is inlined to the main Max call in the following way:↩
    */
int main( )
{
   int a = 100;
   int b = 1010;
   cout << "Max (a,b): " << (a>b)? a : b << endl;
   return 0;
}
```

6.2 GPIO

GPIO test consists on how much time takes the board to perform a certain number of iterations changing an output port between the high and low states. Two channels are used in this test, the first one will activate the Logic16 to start sniffing the second channel which will be the one that performs the changes.

The following code shows the test using the C# implementation. This is the minimum test to analyse the performance of Mono and C++ when I/O is involved using the original IOSharp and its translation. In this case, it is measured the whole time that system requires to change the state of an output port in a certain number of iterations.

Listing 6.2: GPIO Performance test in C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.SPOT.Hardware;
using IOSharp.NETMF.RaspberryPi.Hardware;
using System.Threading;

namespace raspberrypi
{
    class Program
    {
        public static void Main()
        {
            Debug.Print("START");
            OutputPort bar = new OutputPort(Pins.V2_GPIO17, false);
            bar.Write(false);
            bool foo = false;
            OutputPort o = new OutputPort(Pins.V2_GPIO11, false);

            bar.Write(true);
            for (int i = 0; i < 10000; i++)
            {
```

```
                foo = !foo;
                o.Write(foo);
            }
            bar.Write(false);
            Debug.Print("END");
        }
    }
}
```

The following code corresponds to the C++ translation.

Listing 6.3: GPIO Performance translated to C++

```
#include "Program.h"
namespace raspberrypi {

  void Program::Main(){
    Program* p = new Program();
    p->Run();
  }

  void Program::Run(){
    Debug::Print(new String("START"));
    OutputPort* bar = new OutputPort(Cpu::Pin::GPIO_Pin17, false);

    bar->Write(false);
    bool foo = false;
    OutputPort* o = new OutputPort(Cpu::Pin::GPIO_Pin11, false);
    bar->Write(true);
    for (int i = 0; i < 200; i += 1) {
      Debug::Print(new String(i));
      foo = !foo;
      o->Write(foo);
    }
    bar->Write(false);
    Debug::Print(new String("END"));
  }
}
```

This test has been executed varying the number of iterations between 200 and 10000. Each iteration swaps the port state between high and low. Apart from this iteration increase, the assembly is compiled using both compiling modes.

6.2.1 200 Iterations

The result produced by Mono with the optimized assembly shows an irregular pattern consisting of two small pulses followed by a wide pulse, and then two small pulses followed by a wide gap. In the figure 6.1 this pattern is coloured in blue and as it can be seen, it is regularly repeated across all the test. The wide pulses and gaps are supposed to be caused by the garbage collector and the thread round-robin that Mono does.
The small pulses are around 1 ms and the wide gaps/pulses are 2 ms long. Each block is repeated every 10 ms.

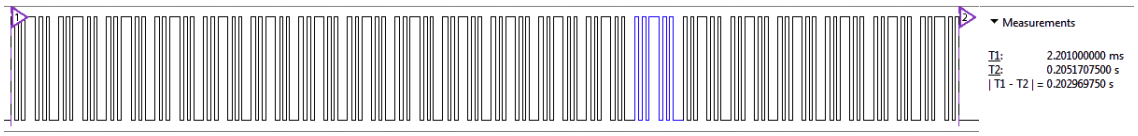

Figure 6.1: 200 Iterations using C# with optimizations

The figure 6.1 represents this 200 changes in the output state. The required time to make this iterations is the elapsed time between the marker 1 and the marker 2. The Raspberry Pi with Mono needs 202 ms in order to do all the iterations.

After testing the performance in Mono it was time to try out the code generated by AlterNative, which is also compiled in release mode. The resulting test is shown and explained below.

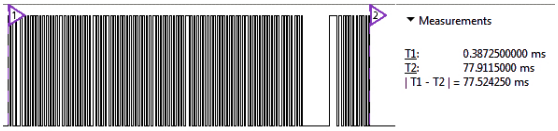

Figure 6.2: 200 Iterations using C++ with optimizations

This above image uses the same scale as the figure 6.1 shows, so the magnitude of the elapsed time can be compared.

In the case of C++ test, it can be observed that the pulses are much more regular than the Mono test, but at certain point around the pulse 89 a big gap is observed probably due to the lack of a garbage collector (AlterNative programs currently do not have a working garbage collector implemented). It is relevant to remark that C++ needs only 77 ms to perform the same test, and each pulse is 0.43 ms on average.

After doing some tests on debug and release mode in both languages a graphic could be sketched, the performance in 200 iterations on Mono using both compile types was the practically the same, it took around 200 ms to complete all the test, however in C++ the time varies a bit depending on the compilation type, without using optimizations it takes 100 ms but when optimization is applied the time decreases to 78 ms.
From the figure 6.3 can be concluded that the C++ version is a 62% faster than the Mono one.

Figure 6.3: Graph showing the elapsed time for the 200 iteration test. Blue is for C++ while orange is C#. On the left is represented the non-optimized compilations and on the right the optimized ones

6.2.2 10K Iterations

After testing the 200 iterations another one was done, but increasing the number of iterations to 10000 or in a factor of fifty. In this magnitude the compiler optimizations should be visible enough to observe some kind of improvement on the different compilation and language types.
In the figure 6.4 is represented the elapsed time for 10k iterations, the block on the left is the Mono version and lasts 10 seconds while the second one is the C++ version and the iterations are done in approximately 3 seconds being approximately three times faster.

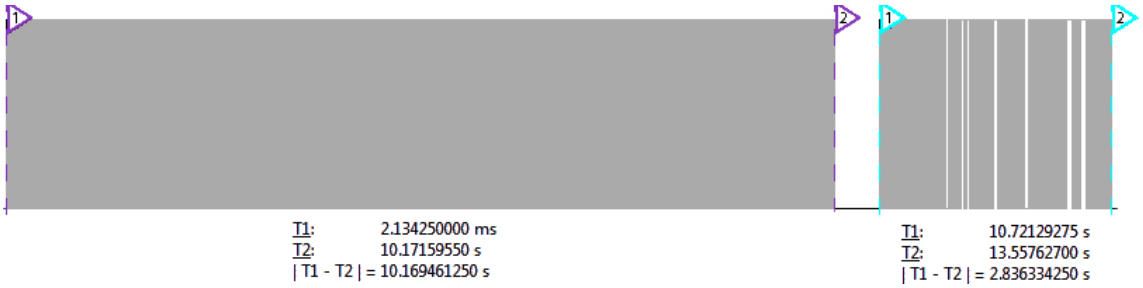

Figure 6.4: 10k Iterations using C# with optimizations

As it was done in the previous test a graph has been done comparing the different languages and compilation types.

Figure 6.5: Graph showing the elapsed time for the 10k iteration test. Blue is for C++ while orange is C#. On the left is represented the non-optimized compilations and on the right the optimized ones

At this number of iterations, the C++ performs much better than C# and also it is also possible to see an improvement between the debug and release versions of C++ being the optimized version 741 ms faster than the non-optimized one. And the difference between both languages shows that the C++ is 72% faster than C#.

6.3 Interrupts

It was observed that in HomeSense the response time in the sensor events was poor, probably because of the time that Mono takes between the interrupt detection and the response. To analyse the elapsed time between the trigger and the response the following test will be polling events from a pin, when an event occurs another pin will be used in output mode with an active high state.
Like the previous test, the C++ code has been generated using AlterNative so it also will be used to test some special functionalities like the delegates, the thread and the timer. The program will start and then create a delegate which will be the called function when an interrupt occurs.
In the figure 6.6 is shown the triggering of the interrupt (the top channel with the falling edge), after 1.326 ms the second channel is activated in state high which represents the interrupt response.

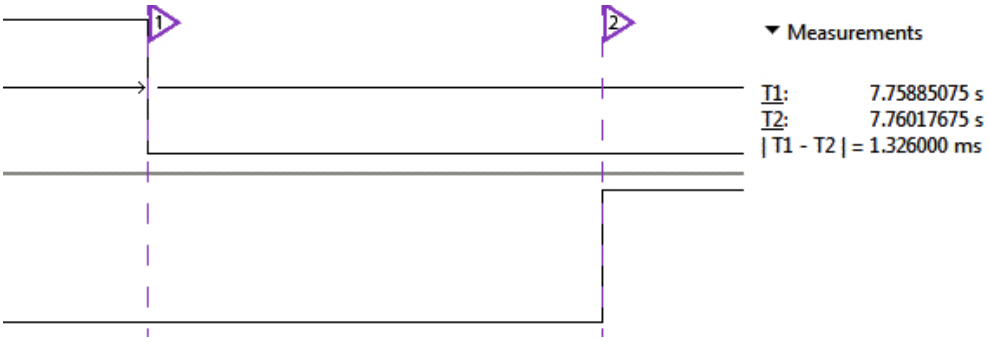

Figure 6.6: Response time of an interrupt in C# and Mono

Then the same test is performed using the C++ version. The result is pretty good because the interrupt is attended only 0.879 ms after the triggering. This implies that C++ is 447 μs faster than C#.

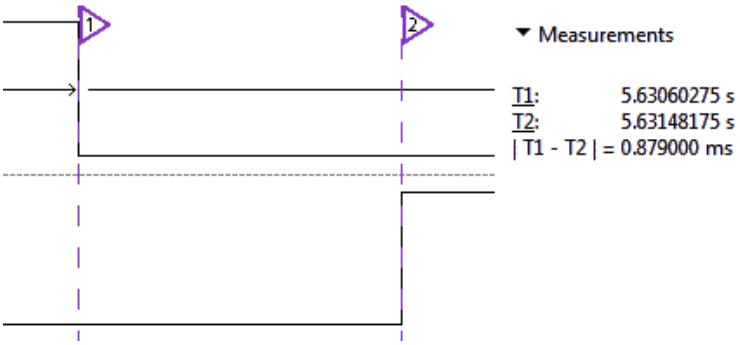

Figure 6.7: Response time of an interrupt in C++

It is important to remark that the interrupts in operating systems cannot be considered real-time interrupts, this implies that sometimes an event is attended at a certain time but in another moment, the time required can be much more different due to the non real-time kernel that usually Linux uses. If a hard real-time is needed it is recommended to use other platforms like FreeRTOS which is intended to do tasks that require controlled response times.

6.4 Conclusions

The iteration test is important because in embedded systems the highest speed that can achieve a board is related to the maximum rate that the board can interact with external devices. Whereas the interrupt test, which analyses the interrupt-response time, is useful to determine the response delay in front of an interrupt. Apart from this, these tests can certify that one of the use cases of AlterNative is real, because the performance is increased as the 10K iteration example shows being 75% faster than the program running in Mono.

CHAPTER 7. CONCLUSIONS

7.1 Project Conclusions

An implementation of Micro Framework has been done with the necessary features to make the existing software created for that platform work on it (HomeSense and the RFID card reader for example). Although it works well, the performance is not as good as the original, probably because the Micro Framework runs on dedicated hardware and all the consuming operations are implemented internally on the virtual machine in C. So this is probably one of the reasons why HomeSense works better on the Netduino than on the Raspberry Pi although the second one is much more powerful.

Another goals was to create a portable implementation. IOSharp is implemented using C# with some calls to Linux SYSFS or Kernel functions. By using this way the implementation is architecture independent, so it is capable of running either on a standard desktop or on an ARMv6, like the Raspberry Pi. The only requirement is having an underlying Linux with the Mono interpreter executing .NET code.

In order to try to improve the IOSharp performance, it was translated to C++ using AlterNative which is a code translating tool. Currently, it can partially transform the original code to C++ but it still needs some work on its core because many classes are not implemented yet. The parts that AlterNative could not translate where fixed by hand, however this the amount of time saved with this tool is bigger than the required time to implement the same code from scratch. After all, once IOSharp is translated to C++, the performance is increased as it is shown in the chapter 6. For example, in 10k iterations in the state of an Output GPIO the C# version needs around 10 seconds to finish the task while the translated one only takes around 3 seconds. This implies an increase of a 62% in performance.

And finally, the best of AlterNative is that the generated code is still cross platform because it uses standard C++ or libraries which are also available on different platforms, so once the source is generated it can be taken to an ARM Linux and then compile it to an executable assembly.

7.1.1 Achieved Objectives

On Chapter 1 (Project Overview) the objectives for this thesis where listed, above are listed the results in terms of objective achievements.

- **IOSharp:** all objectives had been achieved.
  - **GPIO**
  - **Interrupts**
  - **SPI**

  - **UART**
  - **HomeSense**

- **AlterNative:** most of the objectives where achieved.
  - **Cross-Platform:** now is possible to use this tool in any operating system capable of run .NET programs.
  - **Library:** the necessary methods for IOSharp translation are done.
  - **Tests:** each implemented method has its own functional test.
  - **Performance analysis:** shown in chapter 6.
  - **Translate HomeSense:** although AlterNative translates IOSharp, HomeSense is not completely translated.

7.2 Personal Conclusions

Beyond academic achievements, all the process involving this bachelor's thesis has been rewarding. This project has given me a real chance to start working with embedded strategies, protocols and it also has provided an introduction to C++ development.

This project has allowed me to get familiar with Micro Framework and Linux development, which represents the new paths for the embedded systems as it has been explained on the introduction of this thesis. Then, on the AlterNative part of the project, an introduction to AST patterns was done to achieve the capability to transform a representational AST in one language into another one. Also I was introduced to C++ development focused on multi-platform environments.

There are many skills acquired or consolidated during this time: from the initial touchdown on Micro Framework, platform invocation services (cross-language calls), delegation patterns, guidelines for application performance, etc.

I also have realized that the development of a project is a quite complex task and requires hard effort and dedication, but most of all a strict control of timings in order to accomplish with the established work plan.

7.3 Future Work

The results of this bachelor's thesis point to several interesting directions for future work. In case of IOSharp:

- **Addition of new protocols:** Currently IOSharp offers the simple GPIOs, UART and SPI but there are other common protocols or interfaces that could be developed to extend the features, such as the I^2C bus protocol. It could also be nice to implement the analogical ports or even PWM control in I/O ports.

- **Performance optimization:** The implementation in .NET is too much slow running on Mono, probably related to the use of the SYSFS. It could be interesting to do some performance tests using the SYSFS and the Kernel functions provided by Linux. Apart from this, the interrupts are not implemented using IRQ so using IRQ instead of polling interrupts could be a good improvement on performance although it could affect on the portability between boards, some distributions do not accept IRQ interrupts from the GPIOs.

In the case of AlterNative translating tool:

- **Garbage Collector:** Although AlterNative has a garbage collector implemented using the Boehm GC library, it does not get called periodically so programs with a big footprint in RAM can get out of memory easily. To solve this issue the Boehm GC could be called to release unused RAM and make it available again to the program or the system.
- **Continuous Integration:** Changes on the core of AlterNative are very susceptible to break some implemented functionalities, this is why regression tests were created to test if the different functionalities are working well. The problem is that this test takes a huge amount of time to finish so introducing continuous integration tools such as Jenkins or Hudson can provide an easy way to quickly test these changes on a server capable of detecting new commits to a git repository. After executing all the test an inform could be mailed to know the results of it.
- **Extend capabilities:** It could be interesting to make IOSharp work with MAREA2 which is a Middleware for distributed embedded systems in different areas such as: telecommunications, avionics, health-care, automotive, defense, etc. MAREA is a software specifically designed to fulfil Unmanned Aircraft Systems (UAS) communications and their application to the design of complex distributed UAS avionics.

7.4 Environmental Impact

At last but not least it is relevant to talk about the environmental impact of the work described in this document. It can be seen from the present document, this project consists in the design and development of a software application. This has not a direct environmental benefit, but IOSharp is an implementation on a high-level basis of .NET Micro Framework which is an operating system for embedded devices, so it makes easy to develop applications which helps control different kind of situations like home installations (i.e. lights, temperature or humidity) to applications capable of detect and analyse different parameters from the environment (i.e. weather and wind stations).
AlterNative also reduces the power consumption required to execute a program, by removing the .NET virtual machine in which non-translated programs relay on. The translated binary runs directly on the operating system avoiding all the overhead that a virtual machine implies.

GLOSSARY

BCM2835 ARMv6 CPU mounted on the RaspberryPi.

CS Chip Select.

File Descriptor file descriptor (FD) is an abstract indicator for accessing a file on POSIX systems.

GPIO General Purpose Input Output.

HAL Hardware Abstraction Layer.

I^{2}C Inter-Integrated Circuit is a multimaster serial single-ended computer bus.

ioctl Abbreviation of input/output control, system call used for device-specific input/output operations.

IRQ Interrupt Request.

ISM Industrial, Scientific and Medical.

MISO Master Input Slave Output.

MOSI Master Output Slave Input.

PAL Platform Abstraction Layer.

PWM Pulse-width modulation.

RPL IPv6 Routing Protocol for Low-Power and Lossy Networks.

RX Reception Channel.

SCLK Serial Clock.

SoC System on a Chip.

SPI Serial Peripheral Interface.

SYSFS Virtual file system provided by Linux. SYSFS exports information about devices and drivers from the kernel device model to user space.

TX Transmission Channel.

UART Universal Asynchronous Receiver/Transmitter.

WSN Wireless Sensor Network.

REFERENCES

[1] Weinberg, B., *Uniting Mobile Linux Application Platforms*, LinuxPundit.com, 2008. Also available at http://www.linuxpundit.com/cv/docs/Platforms_WP_LP.pdf

[2] Gómez, C., Paradells, J. and E. Caballero, J, "Operating Systems", *Sensors Everywhere. Wireless Network Technologies and Solutions*, p.273-278, Fundación Vodafone España, 2010. Also available at http://fundacion.vodafone.es/static/fichero/pre_ucm_mgmt_002618.pdf

[3] Kühner J., *EXPERT .NET MICRO FRAMEWORK*, Apress, United States of America, 2010.

[4] Kiely, D., "Delegates Tutorial", *The Microsoft Developer Network (MSDN)*. Available at http://msdn.microsoft.com/en-us/library/aa288459(v=vs.71).aspx

[5] ILSpy homepage, http://ilspy.net/, 2013

[6] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers. Principles, Techniques, and Tools*, Bell Telephone Laboratories, United States of America, 1986.

[7] D. Grune, H. E. Bal, C. J.H. Jacobs, K. G. Langendoen, *Modern Compiler Design*, Vrije Universiteit, Amsterdam and Delf University, John Wiley & Sons, Ltd, England, 2000

Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

APPENDICES

TITLE : IOSharp: .NET Micro Framework on Linux

DEGREE: Bachelor in Telematics Engineering

AUTHOR: Gerard Solé i Castellví

DIRECTOR: Juan López Rúbio

DATE: February 7, 2014

APPENDIX A. TECHNICAL INFORMATION. LIBRARIES AND DATASHEETS

A.1 Library Types

Below there are explained the different types of libraries, its characteristics and the different pros and cons of them.

- **Static Library**
  This kind of library is the one that is imported while compiling, and when the source code in the library is linked statically in the generated binary, the compiler takes the library functions used along the program and then they are embedded statically to the compiled binary. In this way, the functions are physically located with the program.
  For example, imagine that you create a library with 3 functions, called `get_password()`, `generate_token()` and `hash(char[] plain())`, but in the current project you only use two of this functions, which are `get_password()` and `get_token()`. When the compiler builds the project, it code will only take the functions from the library that are currently used in the program and then insert them into the binary. At this point, deleting that library won't affect the generated binary, and it will be able to use it without any dependency problems related to missing libraries (dependencies).
  The typical files for this type is .lib for Windows and .a for UNIX systems.

- **Dynamic Library**
  In order to avoid the replication of libraries that occurs in static ones, the dynamic libraries were created. This type of libraries is normally used along the operating systems to let applications use the offered functions and APIs written from the OS. In this case, and instead of the functional behaviour of static libraries. It is important to check that the dependencies are well satisfied when the binary is used, because a missing dependency will break the execution.

  The usual extension files are .dll for Windows and .so for UNIX. But .so files are also used in Windows, especially in web browsers, which use this types of library to load browser plugins such us Flash.

  There are two subtypes of dynamic libraries which are explained below:

  - **Dynamically Linked**
    These libraries must be available at compiling/linking phase, because the compiler will verify that the function exists and that it is used properly. The libraries will be loaded at start time of the program. In this case, all the functions are mapped into the code.
  - **Dynamically Loaded**
    Instead of the previous library, the dynamic loading is used by programs to load or unload libraries and use its functions at run time. When the program needs to use a function it loads the library, then it uses the required functions and finally the library is unloaded again.

Pros and cons

The main problem of using static libraries is that the compiled binary takes much more memory and the library is embedded in every program that needs some functions from that library. But, on the other hand, by using static libraries, the access to its functions by programs is much faster than dynamic ones. Using them also avoids dependency problems, because the dependencies are embedded instead of being located in the file system as dynamic libraries do.

Regarding the dynamic libraries, they help to avoid replications and memory consumption, and also help to maintain the library updated in all programs that use them. Although this can seem pretty good, it can generate two bad effects into the compiled program. First of all, if the library is missing in the system, the program will not run or will crash in execution time. Secondly, if the library is updated but some methods are changed, the program will crash because the non-existing function, and it will be necessary to readjust the code again, recompile and redistribute it.

A.2 spidev.h

Source code of the library spidev.h used along this project. The different macros needed to configure the ioctl calls.

Listing A.1: linux/spi/spidev.h

```
/*
 * include/linux/spi/spidev.h
 *
 * Copyright (C) 2006 SWAPP
 *      Andrea Paterniani <a.paterniani@swapp-eng.it>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
  */

#ifndef SPIDEV_H
#define SPIDEV_H

#include <linux/types.h>

/* User space versions of kernel symbols for SPI clocking modes,
 * matching <linux/spi/spi.h>
 */

#define SPI_CPHA                0x01
#define SPI_CPOL                0x02

#define SPI_MODE_0              (0|0)
#define SPI_MODE_1              (0|SPI_CPHA)
#define SPI_MODE_2              (SPI_CPOL|0)
```

```
#define SPI_MODE_3		(SPI_CPOL|SPI_CPHA)

#define SPI_CS_HIGH		0x04
#define SPI_LSB_FIRST		0x08
#define SPI_3WIRE		0x10
#define SPI_LOOP		0x20
#define SPI_NO_CS		0x40
#define SPI_READY		0x80

/*---------------------------------------------------------------------------*/

/* IOCTL commands */

#define SPI_IOC_MAGIC			'k'

struct spi_ioc_transfer {
	__u64		tx_buf;
	__u64		rx_buf;

	__u32		len;
	__u32		speed_hz;

	__u16		delay_usecs;
	__u8		bits_per_word;
	__u8		cs_change;
	__u32		pad;

	/* If the contents of 'struct spi_ioc_transfer' ever change
	 * incompatibly, then the ioctl number (currently 0) must change;
	 * ioctls with constant size fields get a bit more in the way of
	 * error checking than ones (like this) where that field varies.
	 *
	 * NOTE: struct layout is the same in 64bit and 32bit userspace.
	 */
};

/* not all platforms use <asm-generic/ioctl.h> or _IOC_TYPECHECK() ... */
#define SPI_MSGSIZE(N) \
	((((N)*(sizeof (struct spi_ioc_transfer))) < (1 << _IOC_SIZEBITS)) \
		? ((N)*(sizeof (struct spi_ioc_transfer))) : 0)
#define SPI_IOC_MESSAGE(N) _IOW(SPI_IOC_MAGIC, 0, char[SPI_MSGSIZE(N)])


/* Read / Write of SPI mode (SPI_MODE_0..SPI_MODE_3) */
#define SPI_IOC_RD_MODE			_IOR(SPI_IOC_MAGIC, 1, __u8)
#define SPI_IOC_WR_MODE			_IOW(SPI_IOC_MAGIC, 1, __u8)

/* Read / Write SPI bit justification */
#define SPI_IOC_RD_LSB_FIRST		_IOR(SPI_IOC_MAGIC, 2, __u8)
#define SPI_IOC_WR_LSB_FIRST		_IOW(SPI_IOC_MAGIC, 2, __u8)

/* Read / Write SPI device word length (1..N) */
#define SPI_IOC_RD_BITS_PER_WORD	_IOR(SPI_IOC_MAGIC, 3, __u8)
#define SPI_IOC_WR_BITS_PER_WORD	_IOW(SPI_IOC_MAGIC, 3, __u8)

/* Read / Write SPI device default max speed hz */
#define SPI_IOC_RD_MAX_SPEED_HZ		_IOR(SPI_IOC_MAGIC, 4, __u32)
#define SPI_IOC_WR_MAX_SPEED_HZ		_IOW(SPI_IOC_MAGIC, 4, __u32)



#endif /* SPIDEV_H */
```

A.3 SPI Test. RFID Reader

A.3.1 MFRC522 Datasheet

The datasheet of this card card reader can be find at the NXP website or in the following link `http://www.nxp.com/documents/data_sheet/MFRC522.pdf`.

A.3.2 RFID Reader program

Listing A.2: SPIExample.cs - RFID Reading interval

```
using System;
using System.Runtime.CompilerServices;
using System.Threading;
using IOSharp.Utils;
using System.Net;

namespace IOSharp.Exmples
{
    public class SPIExample
    {
        private MFRC522.SPIApi mfrc522 = new MFRC522.SPIApi();
        private bool onUpdate = false;
        private bool activated = false;
        private Timer cardReader = null;

        public static void Main()
        {
            new SPIExample().Run();
        }

        private void Run()
        {
            mfrc522.ConfigureSPI();
            StringUtils.PrintConsole("MF522-AN Version: "+StringUtils.ToHexString(mfrc522.↩
                ReadReg_MFRC522(mfrc522.VersionReg)));
            ConfigureTimer(!activated);
            Thread.Sleep(-1);
        }

        private void ConfigureTimer(bool activate)
        {
            if (activate)
            {
                Utils.StringUtils.PrintConsole("****Card reader started****");
                onUpdate = false;
                mfrc522.MFRC522Init();
                cardReader = new Timer(StartMFRC522, this, 0, 500);
                activated = true;

            }
            else
            {
                Utils.StringUtils.PrintConsole("****Card reader stoped****");
                cardReader.Dispose();
                mfrc522.MFRC522Stop();
                activated = false;
            }
        }
```

```
        private void StartMFRC522(Object timerInput)
        {
            if (!onUpdate)
            {
                onUpdate = true;
                String cardType = mfrc522.ReadTagTypeString(mfrc522.PICC_REQALL);
                if (!cardType.Equals("*"))
                {
                    CardDetected(cardType, mfrc522.ReadSerialNumberString());
                }
                onUpdate = false;
            }
        }

        private void CardDetected(String cardType, String serialNumber)
        {
            /**Card type
            *       0x4400 = Mifare_UltraLight
            *       0x0400 = Mifare_One(S50)
            *       0x0200 = Mifare_One(S70)
            *       0x0800 = Mifare_Pro(X)
            *       0x4403 = Mifare_DESFire
            */

            cardType = cardType.Trim();
            switch (cardType)
            {
                case "44 00":
                    cardType = "Mifare_UltraLight (" + cardType + ") ";
                    break;
                case "04 00":
                    cardType = "Mifare_One(S50) (" + cardType + ") ";
                    break;
                case "02 00":
                    cardType = "Mifare_One(S70) (" + cardType + ") ";
                    break;
                case "08 00":
                    cardType = "Mifare_Pro(X) (" + cardType + ") ";
                    break;
                case "44 03":
                    cardType = "Mifare_DESFire (" + cardType + ") ";
                    break;
            }
            StringUtils.PrintConsole("Card detected: " + cardType + "- Serial: " + serialNumber)↩
                ;
        }
    }
}
```

A.4 AlterNative System Library

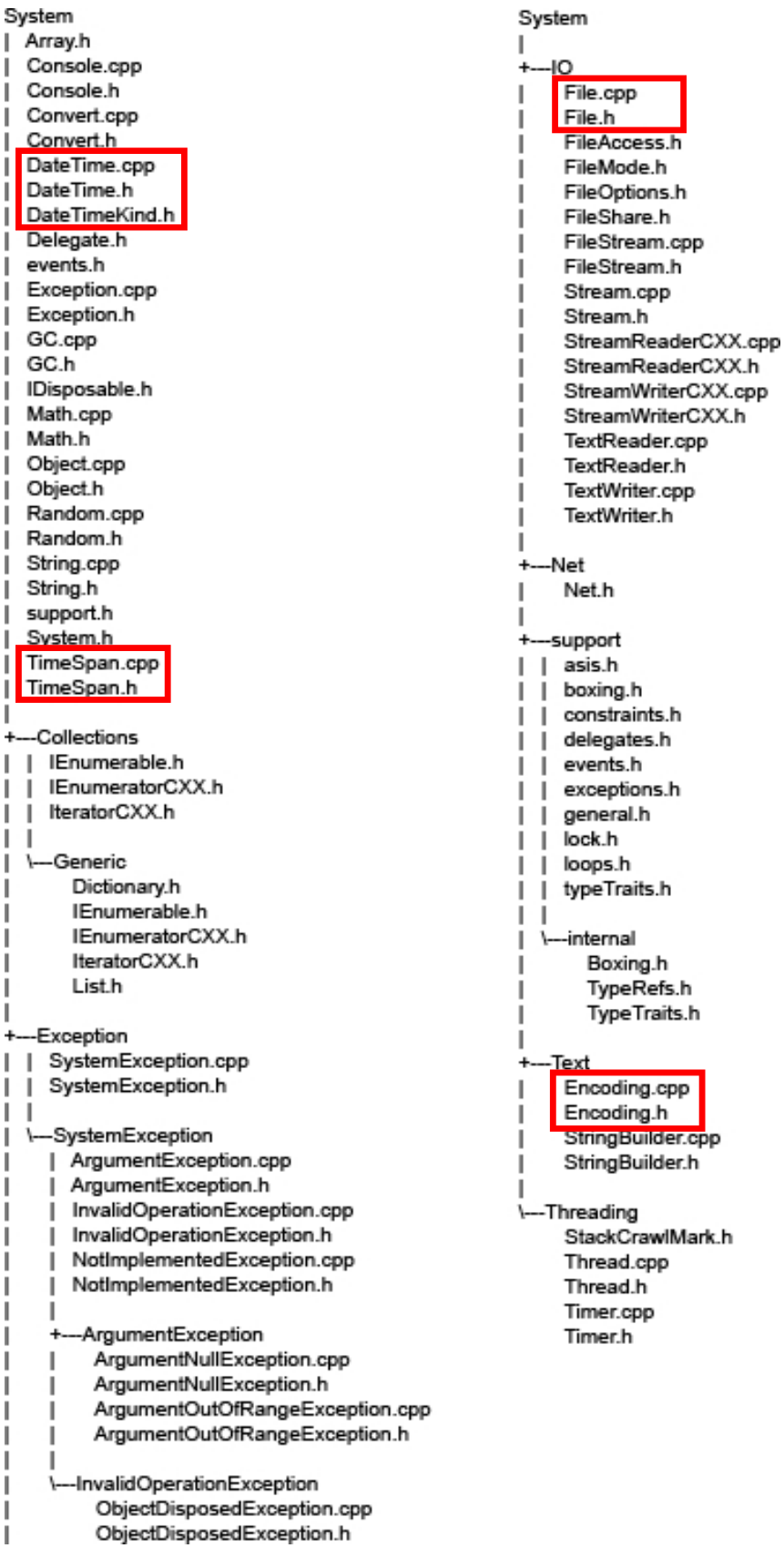

Figure A.1: Tree dump of the C++ libraries of AlterNative currently implemented