Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

TITLE : IOSharp: MicroFramwork a Linux

MASTER DEGREE: Grau en Telemàtica

AUTHOR: Gerard Solé i Castellví

DIRECTOR: Juan López Rúbio

DATE: January 21, 2014

Títol : IOSharp: MicroFramwork a Linux

Autor: Gerard Solé i Castellví

Director: Juan López Rúbio

Data: 21 de gener de 2014

Resum

Aquest document conté les pautes del format de presentació del treball o projecte de final de carrera. En tot cas, cal tenir en compte el que estableix la “Normativa del treball de fi de carrera (TFC) i del projecte de fi de carrera (PFC)” aprovada per la Comissió Permanent de l’EPSC, especialment l’apartat “Requeriments del treball”.

Title : IOSharp: MicroFramework in Linux

Author: Gerard Solé i Castellví

Director: Juan López Rúbio

Date: January 21, 2014

Overview

This document contains guidelines for writing your TFC/PFC. However, you should also take into consideration the standards established in the document Normativa del treball de fi de carrera (TFC) i del projecte de fi de carrera (PFC), paying special attention to the section Requeriments del treball, as this document has been approved by the EPSC Standing Committee

Escriure aquí opcionalment la dedicatòria.

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 is also increased, from tens of Kilobytes to tens or hundreds of Megabytes. One of the reasons for this changes has been the price drop on production. This new embedded systems offers a power similar to the computers from the nineties and most of them implement operating systems which this 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, dynamic program loading. In example, one of the top notch devices nowadays is the Raspberry Pi which mounts an ARM6, 512MB of RAM and some I/O features such as GPIOs, SPI, UART. This hardware can be equivalent to a nineties computer similar to a Pentium II so they are really powerful to the task that they may do.
One of the operating systems 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 the GPIO ports, the SPI, UART and I^2C. There is no official implementation or port of Micro Framework that is capable to run on standard computers (a normal desktop), so any application written for this operating system will not work on Linux or Windows, and although exists a minimal port of this system to a Linux board called Eddy it do not offers all the hardware features that Micro Framework do, and it is only focused to the named board.
The aim of IOSharp is solve this lack and get Micro Framework applications run on any Linux machine which 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 IO 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 ones so this makes much easy to migrate between Micro Framework to .NET Framework.
The reason for this implementation is try to migrate the Micro Framework code that runs on a Netduino Mini which is the gateway of a Wireless Sensor Network (WSN). One of the problems that this platform has is that the it is getting out of system resources, so in order to keep the existing code different solutions are being researched, in this case IOSharp pretends to be one of that 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 expansion project which consists of extending a code translating 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# binary and then translate it to C++ trying to get better performance than C#. C++ theoretically performs better than C# applications, but normally are more platform oriented, so Windows binary will not work on a Linux sytem, but one of the interesting features of AlterNative is generate 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 more hardware oriented development, they are creating a 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.

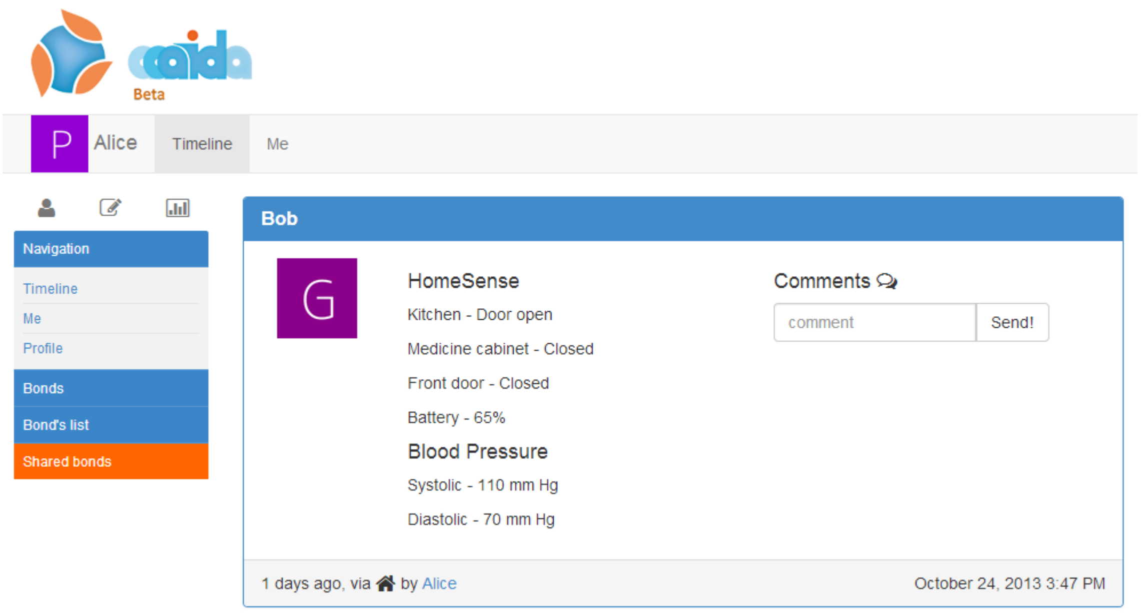
1.1 AlterAid products in depth

1.1.1 HomeSense

HomeSense, is a Wireless Healthcare Sensor Platform WSN created with the aim of control and care taking of the elderly and relatives, actually it uses a Netduino 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 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. This 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 mesh network with multi-hop transmissions which normally the nodes try to fetch the gateway because this is on charge of upload the information straight to the internet.
The gateway system has been entirely developed using .NET Micro Framework and deployed on a Netduino Mini. The mesh protocol has been defined internally on the company while it uses third-party hardware to create the physical links of the network.

1.1.2 Case of use

Alice is a young teenager whose grandfather, Bob, is ill and she wants to know if all is fine in Bob's home life. Alice will sign up in http://www.aaaida.com, there she will create a bond called Bob. A bond is a entity that represents a person, this entity can be configured with different measures. Then Alice will create 2 measures, the first one will be blood pressure while the other one will be bob's house. In blood pressure, Bob will use a simple elderly-oriented mobile application in order to upload his blood pressure every day, in this manner Alice can be aware of its health. Apart from this, Alice will buy a product from AlterAid, called HomeSense, which consists of a set of sensors that must be installed in doors, walls, drawers..., a small centralized system that must be plugged to the power and an application to configure HomeSense. This application will facilitate the linking between Alice account and Bob's bond, in addition, Alice will be able to setup a internet link to HomeSense using grandfather's Internet or a GSM connection to some mobile phone provider. Once the system is configured and working, Alice will be able to take care of Bob's home life, for example if he must take a pill at the morning, she can control at least if the medicine cabinet has been opened. Or if all doors are closed if Bob is out of home.



HomeSense
Kitchen - Door open
Medicine cabinet - Closed
Front door - Closed
Battery - 65%
Blood Pressure
Systolic - 110 mm Hg
Diastolic - 70 mm Hg

1 days ago, via by Alice October 24, 2013 3:47 PM

Figure 1.1: Case of use representation in aaaida

1.2 AlterNative

AlterNative is a language translating tool created by Alex Albalá and Juan López. It is capable of translated a compiled (in .NET) binary or library to standard C++ source, basically this program decompiles the 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, basically it starts writing plain text files with the C++ syntax. After that it links the necessary C++ libraries to work, ones are from boost library, and the other ones are self-written to look 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.

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 idea is to take the HomeSense code which runs on closed systems capable of execute a specific .NET framework called Micro Framework, the problem is that the gateway device that runs HomeSense 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, Interruptions 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 port must be done on the underlying code which executes Micro Framework code so the original source code should be used without major changes, minor changes such as port renaming and communication module name changing are acceptable because 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 use AlterNative to translate the IOSharp driver to C++ in order to increase and analyse the performance of IOSharp running on C++ instead of C#. To accomplish with this some C++ libraries will be need to be written in order to translate IOSharp.

CHAPTER 2. STATE OF THE ART

This chapter sketches out briefly the state of the art of the embedded operating systems and its capabilities. Then according to this thesis it will be explained what is the current operating system running on bottom of HomeSense and finally why has been chosen the RaspberryPi as the target device.

2.1 Embedded Systems

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

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 offers 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.

An OS is an important program 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 de ability to load programs to it self 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 OS 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 isn't recommended to use it.

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 isn't 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 after 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. 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 interruptions 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 build 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 constrained devices which cannot execute the full .NET stack. It is Virtual-Machine based operating system that 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 offers the advantages of .NET language using Visual Studio and it also offers real-time debugging directly on the device.

2.2 Micro Framework .NET

MicroFramwork, is also known as NETMF, had its roots in a project called **Smart Personal Objects Technology (SPOT)**. The first devices implementing the SPOT technology where smart-watches from Fossil and Suunto in 2004 and after them became kettles, weather stations and for traffic and 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.

2.2.1 Devices using Micro Framework

Since Microsoft released the Micro Framework source code, different 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 this of this two boards is the layout of the board, it is totally compatible with most of the arduino shields in the market. Secret Labs has another board called netduino go, which is similar to netduino plus but without storage, ethernet and it does not use the typical arduino layout, so a globus module is required to use arduino shields.
GHI Electronics is another hardware manufacturer that has designed and released different boards implementing Micro Framework or modules 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. One interesting thing of FEZ devices over netduino is the possibility of load native code (C/Assembly) for real-time requirements which it is really interesting. 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.

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.

Eddy is an ARM embedded device board which 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 I2C which are 3 interesting I/O protocols and ports for 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 RaspberryPi. In other words, people in GHI forums are asking for NETMF ports for RaspberryPi 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 NETMF 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 C and .NET ones are interesting to accomplish this thesis goal, 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 the there was a lack of an interesting library written in .NET, probably if this thesis succeeds a more useful .NET library will exist.

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 explaining which two options where at project definition, then the implementation is explained for the different IO ports and communication standards used in this project. Finally, it will be explained how is done the port mapping to work between different boards and devices.

3.1 Planning the development

At project start, the development was focused on a tiny Linux board called RaspberryPi. This device was designed by RaspberryPi Foundation in England taking in mind the kids around the world and helping them to provide cheap tools to be introduced in computer science. It is an interesting board for its features, offering basic IO using the provided GPIOs, it also provides a SPI module for peripheral communication, an 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 WiFi, Bluetooth, ZigBee or any stick for wireless transmissions, HDMI for graphics and user interface and Ethernet for network communications. Apart from this IO characteristics, it also mounts a decent ARMv6 (CPU) running at 700MHz on stock frequency and being overclocked to 1GHz without problems. Together with the CPU 512MB of RAM are provided which is enough for normal desktop usage (surfing, emailing and office) and for embedded projects.

After choosing the target device, two implementing options where designed and analysed. Each one has its own benefits and problems that are going to be explained in the following sections. As an introduction to the options that were considered where the use of the specific tools and libraries for the RaspberryPi and its CPU, this should help on achieving high efficiency when deployed on the device. On the other hand, trying to develop the project as widely as possible to give the chance of running under any Linux (or similar operating systems like Unix or Android) avoiding the exclusion of other devices.

3.1.1 Focused on RaspberryPi

Initially IOSharp was started taking in mind the Raspberry Pi computer, a library written on native C was found. Using this library let native control over all the features provided by its CPU including a wide range of pins for the GPIO, different protocol communications like SPI and UART and other features like PWM in some pins.
This methodology is interesting when is important to achieve high performance on the execution of the programs that make use of its hardware, in this case the modules of the

CPU are used on a low-level way by changing its registers. This normally let the programs run faster, but in principle this is not required for HomeSense usage, although later in this thesis it will be explained that the performance of HomeSense has not been as it should be.
The library written for the bcm2835 is the one that should be used in case of this methodology is chosen. 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

It may also be interesting to get a wider public for IOSharp because more users implies more implementations, tests and in general feedback for the development and improvement of this project. The idea is to make it run on any devices capable of execute C# and C code and in general any device that runs Linux or any similar stack like Unix, Android, being able to use hardware features such us the GPIOs, SPI, serial port, ...
In case of this project, it has been chosen to develop using this method because Linux offers the appropriate tools like the SPI Kernel functions or the mapping of Ports through the user space. To use any of this features, the modules must be loaded to the Kernel in order to have access to this functions or devices.
Apart from this, the Micro Framework offers the possibility to configure the port mapping to the corresponding pins and devices of its underlying hardware. Apart from this, implementations of C will be used in order to exploit the functions provided by the Linux Kernel as it has been commented before.
In short, IOSharp will be capable of run in any platform that uses Linux such as the RaspberryPi, a Cubieboard or a standard desktop.

3.2 Implementation

Remember that as it was explained on the introduction, the goal of this thesis is deploy and run successfully the HomeSense platform on a RaspberryPi. Taking in mind this, some modules where required to develop in order to accomplish with this requirement. In this case the GPIO, Interrupts, SPI and UART need to work in order to use the different components of HomeSense.

3.2.1 GPIO

In order to implement the GPIO ports in NETMF it will be used the IOPorts.cs file which contain the structure for Input, Output, Tristate and Interrupt ports, the first three will be explained in this section whereas the interrupt port will have a dedicated one.
Below is explained the different options on implementing GPIOs in Linux, which has been chosen and how has been implemented in this project.

3.2.1.1 Implementation Options

GPIO acronym stands for General Purpose Input Output which are Ports on systems that are capable of generating an output or reading an input. Normally embedded systems work with ports at 3.3V, other devices had low power supply like 2V. And it is not strange that many of them are tolerant to 5V in input.
In case of the Netduino and RaspberryPi the ports on both devices run at 3.3V and also the different test hardware modules used in this thesis.

The GPIOs in Linux can be controlled in several ways, the most common and simple is use the userspace which stands for a set of directories with readable and writeable files representing the ports of the MCU or CPU. On the other hand the Linux kernel also provides a library and a module in order to control the different pins.
The decision must be done between this two systems, in order to use any of this solutions the GPIO must be activated in kernel, many desktop Linux distributions have GPIO disabled and that's why the kernel must be recompiled enabling this feature. Basically GPIOs and SYSFS must be activated on kernel configuration, after compiling and installing the new kernel both methods will be enabled.
In case of Linux operating systems designated for embedded devices, for example the RaspberryPi or CubieBoard, will have the Input/Output Ports enabled by default.
One of the most curious things that where found at studding this possibilities was that Android is capable to use this ports. Although it cannot seem an interesting feature nowadays android is everywhere and can run in many devices, so is another reason to try to fetch this this sector in future versions of this software.

3.2.1.2 Using GPIO from SYSFS

Since each solution can be used in this project and both are available in any Linux OS when the kernel is properly compiled the chosen option was control the GPIO through the SYSFS in order to simplify both the development and the testing.
Use the Input/Output ports using the userspace is really simple and that one of the biggest reasons, the easiness of testing and debugging the functionality. It is easier to read or write a file in order to test if the control of the GPIOs is successful rather than debugging a C function embedded in a library.

As it was said before, in userspace the control of the GPIOs is carried by several files and directories located under */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 will be used to disable it. After enabling a GPIO a new folder will be created representing the enabled port, for instance if port 2 is enabled, a folder called gpio2 will be created. Inside this new folder there will be several files, the direction file describes how port should work, if the desired function is as an input port an "in" must be written in the file whereas "out" must be write for an output port. After setting the port direction the value file comes in which will do the functions of reading the port in case of input ports, or write 0 or 1 through it if the port is described as an output. To do this, just read this file to read an incoming value, or write 1 for active-high or 0 for active-low.

3.2.1.3 *Implementing in NETMF*

Taking in mind that this implementation has to be done over the existing code extracted from the *IOPorts.cs* is important to design how to do it properly, in this case a **GPIOManager** has been created using a singleton pattern, in order to restrict one instantiation of this class among all the code, and simplify the use of this class.
This manager will be in charge of enabling, disabling and operating the different ports. Also it will control which ports are enabled in order to avoid problems, i.e instantiating the same port twice will make the GPIOManager throw an exception so it can prevent hardware damages.
For GPIO implementation Output, Input and Tristate ports will be required, below you can see how look the classes provided by NETMF. This classes will consume the GPIOManager explained above.

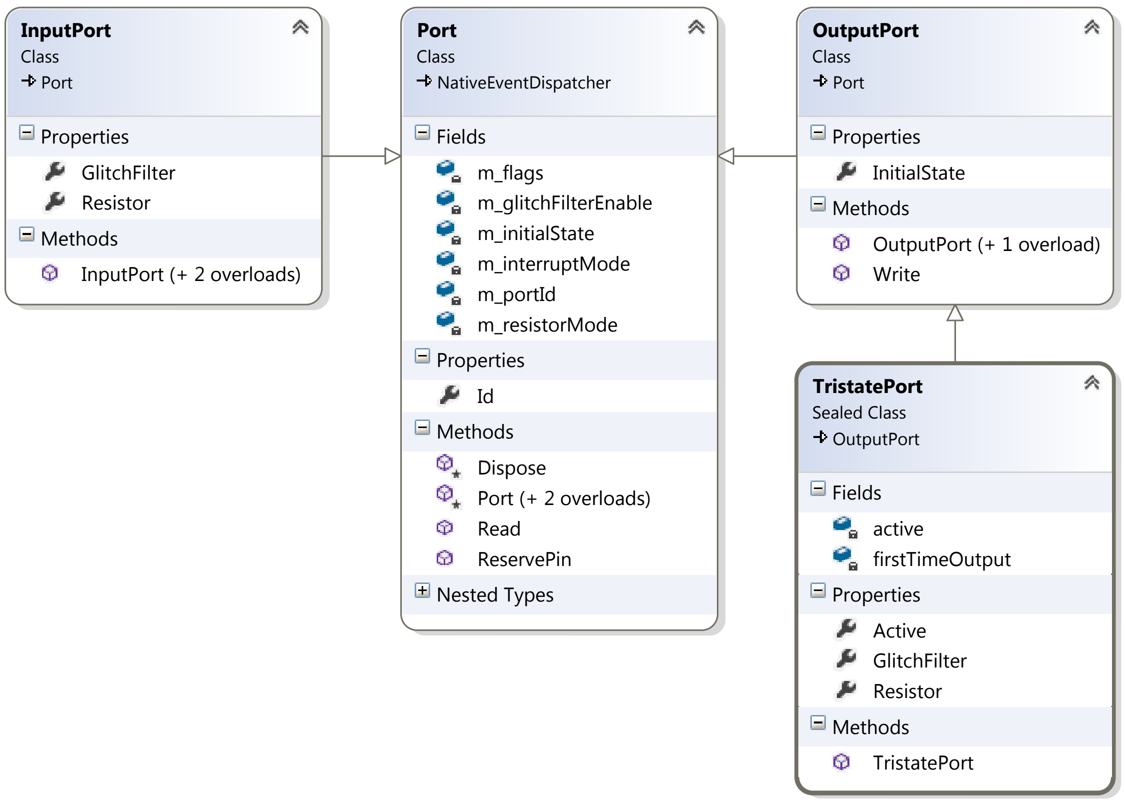


Figure 3.1: UML Diagram of NETMF Port and its inheritance

As is shown each Port type inherit from Port object which implements the methods to enable or disable a port (Port and Dispose methods), Read which is used to obtain the current state of the port i.e read an input value or know in which state is configured the output port. Finally it also has a ReservePin and as its name says, it is used to reserve a pin for future usage. Taking a look on the left box it can be seen the InputPort which inherits from Port. It does not have any special method and its constructor suppers to Port class. On the right side there is an OutputPort also inheriting from Port which implements a new method called Write that its used to write a state through the port, active high or active low. TristatePort also inherits from OutputPort, a NETMF TristatePort is a type of port capable of commuting between input or output port, i.e with a TristatePort is possible to control 2 leds which one of them is connected in pull-up and the other one in pull-down, so with this TristatePort is possible to light one, the other, if it is configured as an output port either on active high or low, or turning the TristatePort to an input port will avoid the lightning of any led.

3.2.2 Interrupt

After implementing the GPIO and taking in mind the requirement of HomeSense an interrupt system was needed for the future SPI code block. Although the BCM2835 supports native interruptions via IRQ at the time of this project the RaspberryPi did not support GPIO interruptions from the Linux Kernel using IRQ, but when this thesis was being written this interrupt types where implemented. It was decided to maintain the current implementation to avoid problems of the same kind in other platforms and devices.

3.2.2.1 Designing the Interruptions

The interruption system will be written in C because of the simplicity to detect an interruption on a GPIO port, after doing some research a solution was found which uses a C function called poll, this is commonly used by developers that want to intercept GPIO interruptions in Linux environments where IRQ interruptions from GPIO are not available. The poll function is configured to wait certain events on a File Descriptor obtained from the GPIO file enabled on the SYSFS in a similar way as it was explained in the 3.2.1 GPIO section. The poll is configured to wait the execution of the file until the POLLPRI event is detected. This event will be triggered by the OS when the file had urgent data to be read and the poll function will collect this event and then will continue executing code block which basically will read the state of the port (active high or active low). As it is really important to notify the running program of an interruption on the port, a delegate will be used, this delegate is configured by the developer on the `OnInterrupt` using NETMF and basically it contains three parameters which are the port, the state and the time of the event.
The interruptions can be disabled at any time and the port can also be disposed.

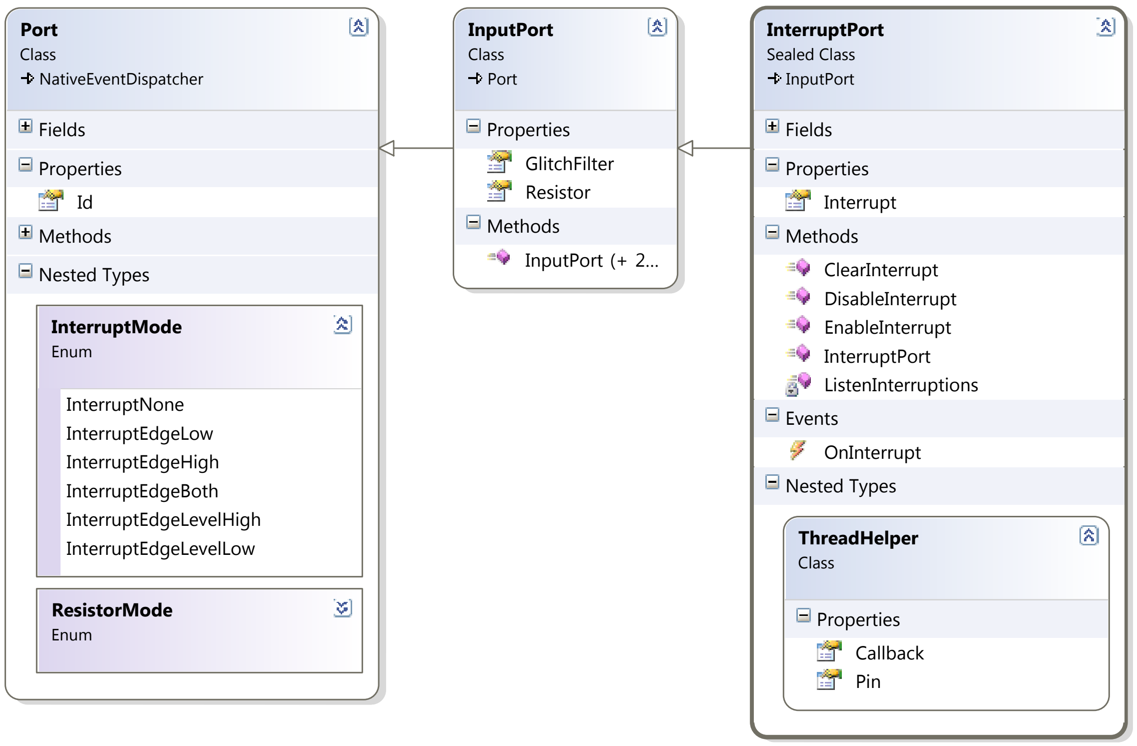


Figure 3.2: UML Diagram of NETMF Interrupt Port

3.2.2.2 Platform Invocation Services

In C# is possible to invoke external libraries which they do not have to be written in the same language, in this case the library has been written in C and then compiled into a shared library making it available to any program suitable for it, IOSharp in this case.
The Platform Invocation Services is the methodology that has been chosen to do this external calls to written libraries. This calls are known as P/Invokes and are used to call unmanaged code from managed code.

- **Static Library**
  This kind of library is that one that is imported while programming, and when the source code is the library is linked statically in the generated binary, this means that the compiler takes the library functions used along the program and then are embedded statically to the compiled binary, in this way, the functions are fiscally 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 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 way of static libraries, any function from the library will be embedded on the generated binary, in this case is important to check that the dependencies are well satisfied when the binary is used, because a missing dependency will break the execution.

In this case, 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/link 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, using static libraries the access to its functions by programs is much fast than dynamic ones. Using them also avoids dependency problems, because the dependencies are embedded instead of being located in the file system like dynamic ones.
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 have two bad effects into the generated 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.

P/Invokes in .NET makes use of dynamic loaded libraries in order to use the contained functions. The implementation difficulty of P/Invokes increases on how complex is the function to be called regarding its parameters, for basic type parameters such us `int`, `long`, `byte`, etc is really simple to make a P/Invoke call, but when passing object parameters things get much difficult because they must be passed to unmanaged code which sometimes can be impossible to do without using structs as interchange objects.

Below is shown the important parts of the implementation of the library and how P/Invoke is done in C#.

Listing 3.1: IOSharp.c - Polling function

```c
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;
}
```

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

```c
#ifndef IOSHARP_H_INCLUDED
#define IOSHARP_H_INCLUDED

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

#endif
```

Listing 3.3: GPIOManager.cs - P/Invoke section

```csharp
// 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

After defining and writing the library, and studding how to make calls to native functions it was time to code de final implementation making work together, the library and the IOSharp code. In this case, the program flow is shown on Figure 3.3. The Interrupt Port inherits from Input Port as Figure 3.3 shows.

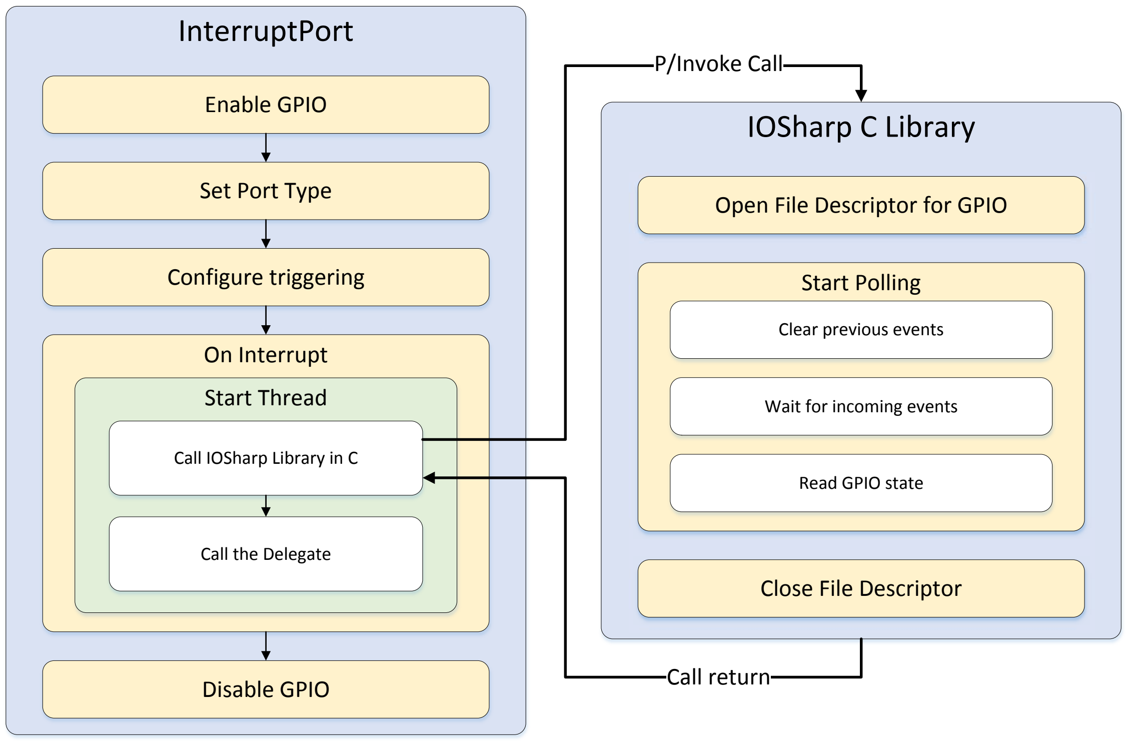


Figure 3.3: Representation of the Interrupt Port flow

The idea is to do the exact same steps like the other ports do, the GPIO enabling is done by the Port implementation which is the base class for Input Port, then the port type is set to be an Interrupt Port, after this is important to configure the triggering. NETMF boards usually supports four kinds of interruptions whereas Linux supports a few less, in the following table is shown which are supported. It is important to know that the Edges are the point where the port changes from one state to another one, and the Level is used for to trigger interruptions where there is a continuity on a certain state.

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 without counting on None interruptions. In order to configure EdgeLow interruptions the word *falling* must be written, in case of EdgeHigh will be used *rising* and finally *both* is used for EdgeBoth.
Once the triggering is configured IOSharp will wait for a delegate to be configured by the user in which will receive the interrupt events, once a delegate is passed to the OnInterrupt a thread will be assigned to the task of polling a GPIO using the library previously explained. This thread works as an infinite loop P/Invoking to C and waiting for the call return, when this occurs the delegated passed by the user is called and in this way the interruption event is passed straight to the program which uses IOSharp. Finally the interrupt port can be disabled as the other ones.

3.2.3 SPI

Potser seria interesant explicar mosi, miso, sclk, cs The SPI is one of the important features to be implemented on IOSharp. The SPI is a protocol used in embedded systems to communicate boards and components in a Master-Slave way. This protocol offers a full-duplex communication where the master and the slave can write and read at the same time on the channel. 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.

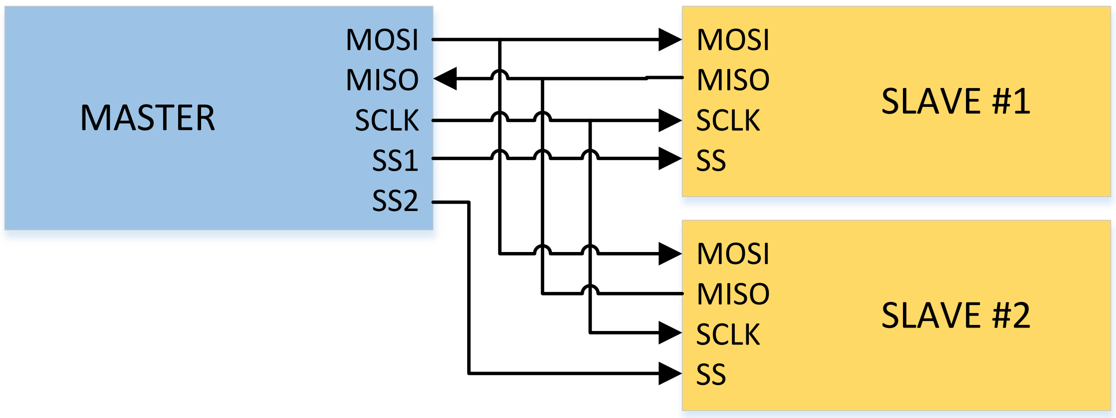


Figure 3.4: SPI bus setup with one master and to slaves

Both Netduino and RaspberryPi support the SPI communication protocol which makes easy the use of different modules. In case of this thesis, the SPI was required for HomeSense which make use of it to control the Nordic chip.

3.2.3.1 Designing the SPI

It was decided that the implementation of this component will be done in a similar way as it has been done in the interrupt port which has been explained on the previous section. The idea is to use the functions provided by the Linux Kernel by using the `<linux/spi/spidev.h>` library. This makes easy the access to the SPI device, make the required configurations and write or read the channel.
In Linux the SPI devices are mapped under the `/dev/` directory with a convention 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`, which is also an integer and it shows the Chip Enable, an SPI device can have multiple chip enablers, so it can has more than one slave.

Before doing a transaction via the SPI bus this must be configured, first of all defining the operational mode. SPI 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 de clock is in the edge low or in edge high according to the configured parameter, the second one, CPHA defines in which phase the sample must be done. This concept is much easy to understand using the figure 3.5 which shows the different CPHA and CPOL options with the equivalent SPI modes needed to configure the device in the C library.

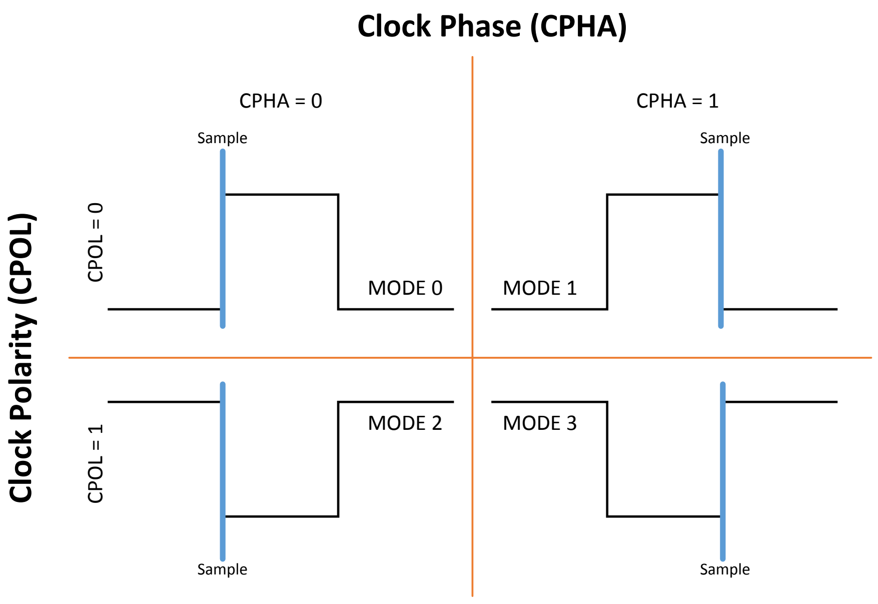


Figure 3.5: SPI modes are defined with the parameters "CPOL" and "CPHA" relatives to the data sampling acording to the Clock (SCLK) state.

This modes must be configured using the ioctl function passing the File Descriptor according to the SPI device and the desired Chip Enable, then the preprocessor macro, in this case the `SPI_IOC_WR_MODE` which is defined in the `spidev.h` and it is used to specify which parameter will be configured, in this case its the SPI mode that has been explained before, the last parameter corresponds to another macro and corresponds to the operational mode explained before, the figure 3.5 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, this struct is the type of `spi_ioc_transfer` specified in the `spidev.h` library. This struct contains different variables, the `tx_buf` and `rx_buf` which are configured with the write and read buffers, and are used as pointers???. Apart from the buffers, the transmission length is configured using the variable `len`. Then the delay is configured, this indicates how many microseconds the SPI driver must wait before starting the transmission, this is important because some slaves take awhile between they are selected and they are able to communicate, this is configured using the `delay_usecs` variable. Another parameter to configure is the clock by using `speed_hz`. In order to deselect a device before another transfer the parameter `cs_change` must be true. Finally, to configure or override the wordsize of the transmission `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,
  };
```

After configuring this struct, it can be passed to another ioctl call which will make the transfer it self, 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 the ir only a transfer at a time, 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 is time to modify IOSharp to add the necessary calls to this library in order to make Micro Framework use the SPI in Linux. In this case the calls will be done in a similar way as it has been done in the Interrupt part (3.2.2. In this case 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 the same, do a P/Invoke from C# in order to call the functions in the 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 configurations 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. First of all 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 configuration the required pins for theMISO, MOSI, SCLK and CS are obtained and reserved, so the used can not create a port (for example an input or an output) over this pins.
When the SPI instance is created, the methods will be able to be used, basically there are different overloads of the `Write` and `WriteRead` methods, but all of them end calling the same internal function which will do the P/Invoke to the library.

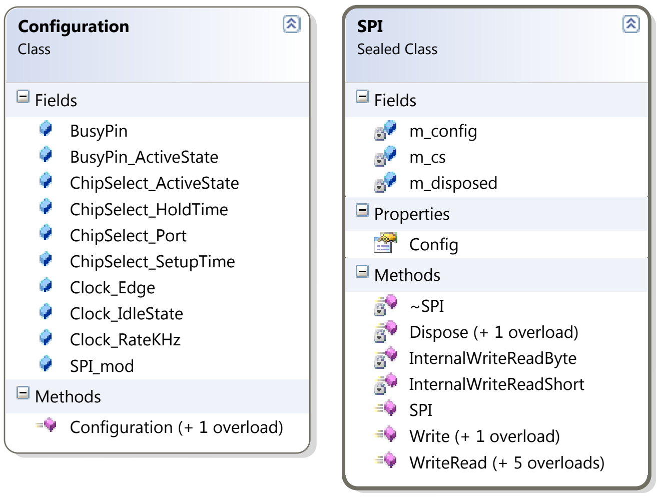


Figure 3.6: UML representation of the SPI Configuration Class (Left) and the SPI Port (Right)

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. This struct will be the one that is passed with the P/Invoke to the C implementation.
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;
```

In this case, this struct represents the C# implementation, it contains the same parameters as the C versions and also implements a constructor which simplifies the conversion between the Configuration class and this structure, the important thing in this case is the `[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]` which is the attribute of the struct which is used by the P/Invoke to know how to serialize the structure and also which encoding must be applied to it when is passed to the 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;
    }
}
```

Using this method the configuration is passed to the library so SPI can work just as the developer wants. It is also interesting to remark that the buffers for the incoming or the outgoing data are not returned from C to C# but the buffers have been read and written so C takes the pointer of that buffer, which is the same as the C# and then writes or reads the information from there.

3.2.4 UART

The UART is a really simple protocol using asynchronous serial communication between two devices. As figure 3.7 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.

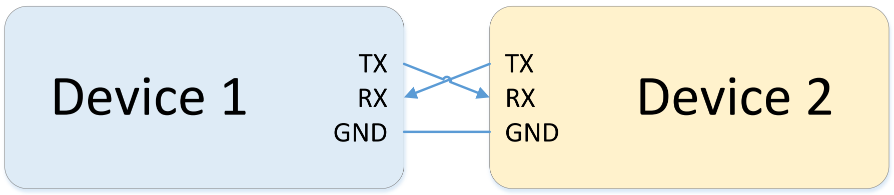


Figure 3.7: UART communication schema

Unlike the above functions which are not implemented on the standard .NET Framework the UART has a class on it with the same naming and namespacing, so this can be problematic in case of class reimplementation maintaining the original namespacing used in Micro Framework.
Before start writing a new implementation a comparison and evaluation was carried on for the implementation of .NET Framework and the Micro Framework version, after comparing them, they where so similar that a new implementation for IOSharp was not required, at least until the missing functions were required. This decision is due two reasons, the first one is that any reuse of code is better rather than writing again the same feature, and considering that the communications between devices must be really well done, a class on the standard framework will be much more stable and tested than a code written from scratch. The other reason on choosing the existing class that although it is not exactly as the NETMF class it has all the required functions that are needed for deploying HomeSense.

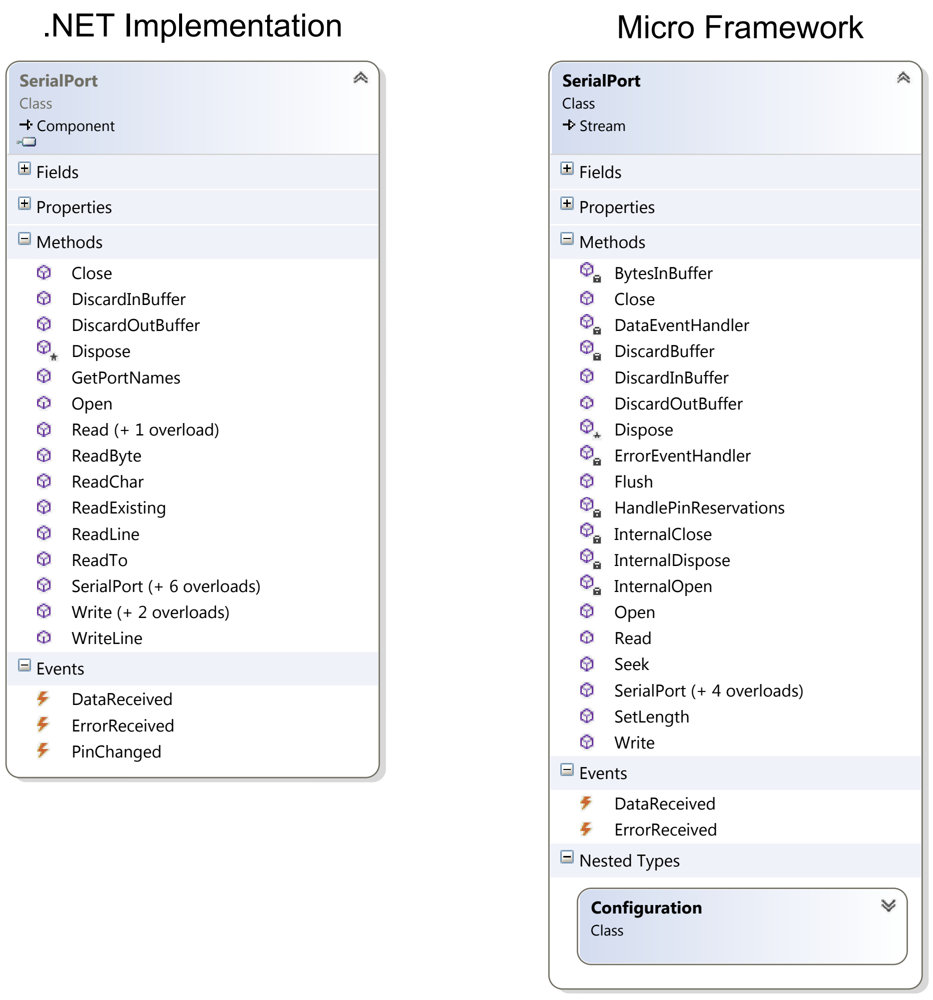


Figure 3.8: UML representation of the original .NET Framework SerialPort (Left) and the SerialPort in Micro Framework (Right)

Is important to emphasize that IOSharp in Linux runs using the Mono implementation of the standard Framework classes, and in this case the SerialPort class has some disadvantages on the Mono version, basically they do not support events such as `DataReceived` or `ErrorReceived` because the functions have not been implemented on its internal runtime, but as it was said, this feature is not used in HomeSense so it does not influence on the execution of the platform.

3.3 Port Mapping

In order to make IOSharp portable between hardware platforms a port mapping must be done. The idea is to map the underlying hardware to variables in order to be used on the upper-layer taking into account its GPIO ports, the SPI devices and its corresponding ports and the Serial Pins.

3.3.1 HardwareProvider

In fact, the original NETMF supports a hardware descriptor called HardwareProvider which do the mentioned things. The Raspberry Pi was the target platform for this project so a hardware descriptor was written, below is represented the pins of the Raspberry Pi, on the left the revision 1.0 and on the right the revision 2.0. In this pins are located the GPIO, two SPI devices, two UART ports and an I^{2}C bus.

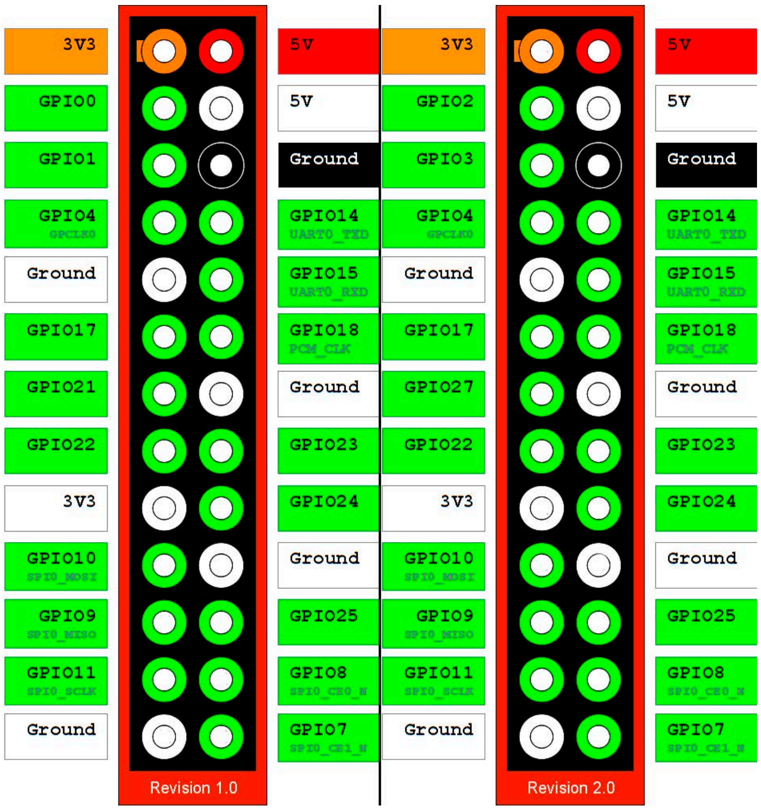


Figure 3.9: Raspberry Pi available ports

The mapping has been divided on two classes, the first one maps the Pins so it will contain the references to the platform pins, in case of this thesis, both versions are mapped. The second file is used to pass the specific pins to the program, for example it configures the pins for the SPI or the UART. It is interesting to mention that in Linux the SPI and UART are known as devices so they are located under `/dev/` directory. For the SPI the device is called spidev0.0 and spidev0.1 in case of the Raspberry Pi, this shows basically that there are two SPI devices which shares the MISO, MOSI and SCLK pins it has two different chip selects so it can connect to two different slaves. In case of the UART it is also represented as a device and will have a name like `/dev/ttyAMA0` in case of Raspberry Pi.

This is not the unique case, for example the Netduino provides the same classes for the pins for the CPU.

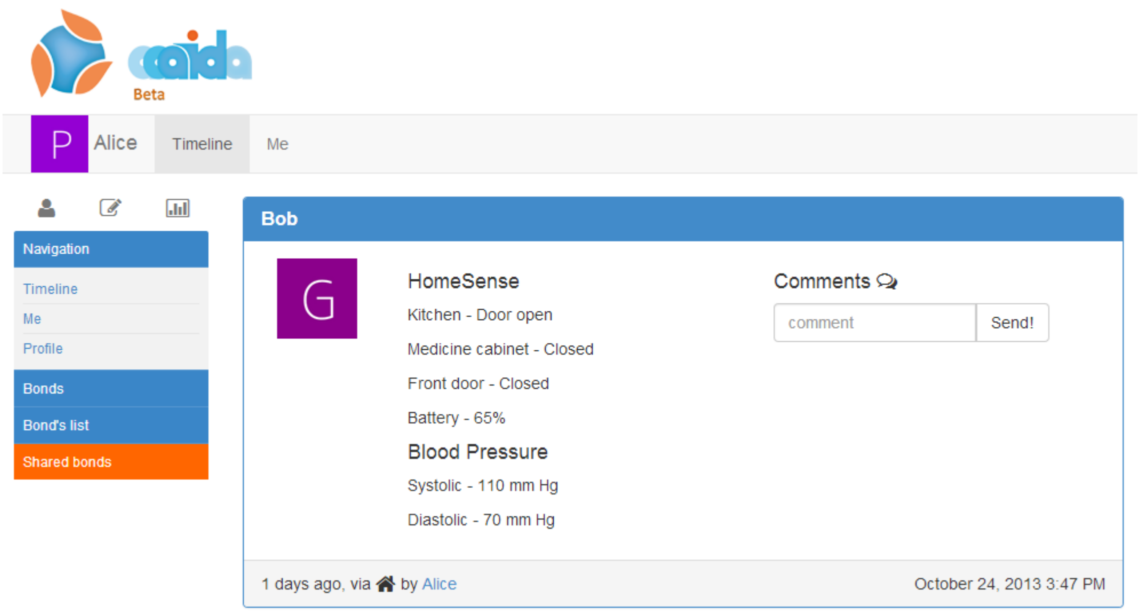
CHAPTER 4. TESTS OF IOSHARP

Simultaneously with the development of IOSharp some small tests were created so the developed module could be tested to verify the properly operation. There are three major tests without taking into account the simple GPIO. First of all an Arduino and a RaspberryPi running IOSharp where connected via a Serial Port so one could send a message and the other forward to the first again. Secondly the SPI was tested using a RFID reader. Finally HomeSense running on the RaspberryPi using IOSharp, and it will be compared with the original version of it, using the Netudino Mini.

4.1 UART. RaspberryPi and Netduino

This example which shows that is possible to use the .NET Framework SerialPort implementation as a replacement of the original Micro Framework one. In this case a Raspberry Pi and a Netduino is used, the first one will send a message through the UART port, when the Netduino receives that message will send it back to the Raspberry Pi.

Include here a Logic capture showing the bytes exachanges



HomeSense
Kitchen - Door open
Medicine cabinet - Closed
Front door - Closed
Battery - 65%

Blood Pressure
Systolic - 110 mm Hg
Diastolic - 70 mm Hg

1 days ago, via by Alice
October 24, 2013 3:47 PM

Figure 4.1: Bytes exchanged between a Raspberry Pi and a Netduino using the UART

As figure 4.1 shows the bytes sended through the TX channel of the Raspberry Pi are the same as the RX connected to the Netduino.

4.2 SPI. RFID and IOSharp

This was the first test to verify the SPI and the Interruptions 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, for example a gym. The project goal was grant users access to somewhere, for example a gym, 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 and have interesting features such as network connectivity. The card reader used along this project implemented a MFRC522 chip from NXP which can use different communication protocols like SPI, UART and I^{2}C. But that chip is mounted on a MF522-AN board from Mifare which only offers a SPI interface. And this is the reason why this proof of concept was used to test the SPI feature of IOSharp.
To communicate the Netduino with the card reader an API was written taking into account the MFRC522 datasheet (see on Appendix A the section A.2.1 MFRC522 Datasheet) and the existing Arduino implementation which uses Wiring, the Arduino programming language. The program creates an instance of this API, then starts the SPI configuration and then creates a Timer which will call a function repeatedly at certain time. This called function uses the API to communicate to the card reader and retrieve the card ID if any is present, then communicates again to obtain the Serial Number located in the card. Finally it prints this data through the console.

4.2.1 Micro Framework version

The original example uses the standard 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.

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 on the Netduino board.
The rest of the project can be viewed at `http://google.com`.

4.2.2 Migrating to Linux

To use IOSharp instead of Micro Framework there is not big requirements, basically the project must be converted to .NET Framework and then reference the IOSharp project which include the implementation of the Micro Framework classes, beside this library the mapping classes according to the deployment platform must also be referenced, in this case the Raspberry Pi pin library. The next step is change the pin for the CS to the according one. Normally in Linux each SPI device have one or more designed 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 by using this test can also be proven 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`) for this project which each one calls for two other project files (`*.csproj`) make possible to have one solution with the Micro Framework classes for the Netduino while the other one will contain 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 (the real typed by the developer) and create a new .NET Framework project with that code. With this changes, it was tried to show that is possible to maintain the same code for different platform deployments.
The next step was make easy the pin configuration because both boards have different pin naming and layout, so by using conditional compiling it was possible to instantiate the necessary port for 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 the Raspberry Pi don't. Taking a look in the above code, for the Netduino the used pin is the `Pins.GPIO_PIN_D9` and in the Raspberry Pi is `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 program working. As the figure posar la referencia shows the exchange of the data between the boards and the RFID reader is done by SPI

4.3 HomeSense over IOSharp

4.4 Results overview

After testing the different parts of IOSharp and deploying HomeSense on the Raspberry Pi using Mono it was seen that the performance was not as good as it was expected. HomeSense, need more time to send the network events and this is probably related to the time needed to attend the interruptions of the Nordic chip. In this case the IOSharp with Mono needs two transmission two send the same information that is send with only one transmission using the Netduino.
To try to solve this 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++ maintaining a similar C# syntax.

CHAPTER 5. ALTERNATIVE

This chapter is a walk-through AlterNative explaining its concept, how translates the code, then an example using IOSharp will be used along 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 possibility to get native code capable of work 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 original code in native language and does not depend on any virtual machine.

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 recompilation to a native assembly.

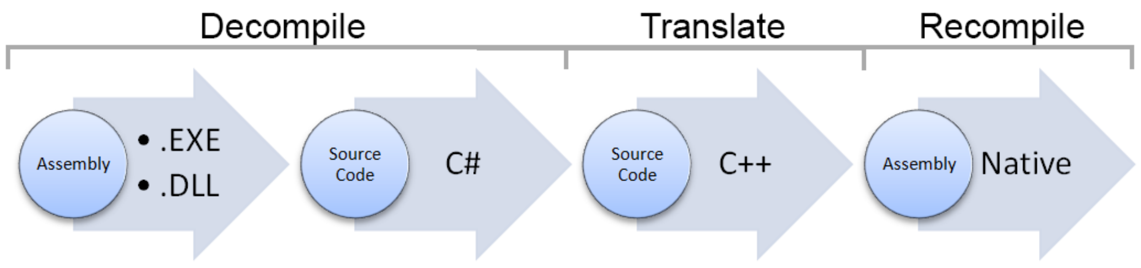


Figure 5.1: AlterNative process

5.2.1 Decompilation

First of all the Assembly (compiled program/binary) is passed through a decompiler in order to extract the source code. In this case the code extracted is C# and to decompile it is used the ILSpy which is an open-source .NET assembly decompiler. The code is not extracted in text format, but in 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 from the top-level (Assembly) until de low-level (instructions, types and constants).

The figure 5.2 shows how the `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 would return the method) which is `void` in this case, the identifier which shows the method name (`Main`) and finally the parameter declaration which describes the parameters that the function takes. In the method shown in the example 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 by a string and an array specifier, the second node is the identifier name of the parameter, in this case, `args`.

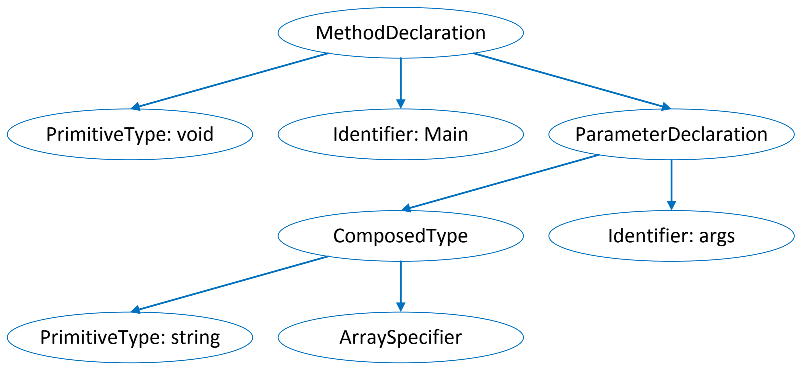


Figure 5.2: AST representation of the main method in C#

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 the how is typed the resulting language, then in order to achieve that some modifications need to be applied to the AST, by doing this a second AST 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. When printing the new code the headers and the code files must be printed.

Following the example explained in the previous section now the AST will be transformed to the correspond with the C++ one. In C# the method was `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 to achieve an AST corresponding to the C++ method which looks like `void Main(string args[])`. 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 moved from the original composed type to the composed identifier. As there is no composed type now the node is deleted and the parameter declaration is directly linked to the primitive type string.

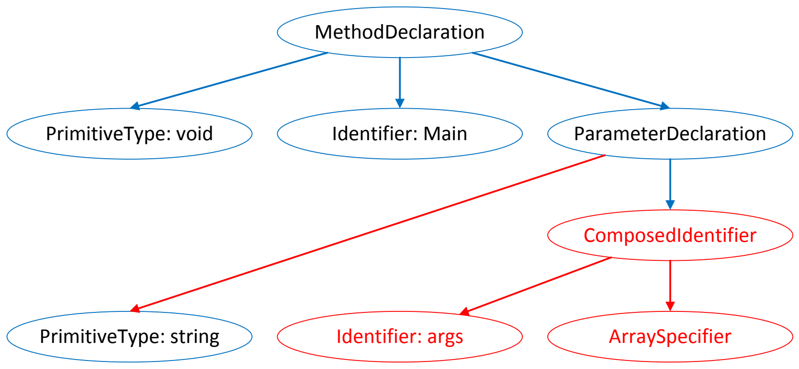


Figure 5.3: AST representation of the main method in C++

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 but 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. With all of this passed through AlterNative the program will run much faster than the original one because it is running in native code.
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 the threading and datetime library 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++ binaries.

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 generate cross-platform programs for embedded systems.

5.3.1 Performance

AlterNative gets the major advantage when 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 is the target, the more benefit that can be obtained. Taking into account that the performance seen on IOSharp over Mono is not as good as it should be compared to the native Micro Framework with a Netduino. Is supposed that passing a program created with IOSharp (which runs on top of Mono) through the translator will generate a faster binary because it does not depend on a virtual machine , and is well known that Mono is not the fastest implementation of the .NET specification.

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 fits with the implementation of IOSharp because passing the library through AlterNative a new library 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 an ARM-Linux assembly so the unique requirement to execute that code will be an underlying Linux running on the machine. The developer will get all the benefits of Micro Framework with the speed of C++, the code will be written using IOSharp or native Micro Framework which is really easy to develop embedded applications with this software and then translate the program with AlterNative.

5.4 Work done in 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 the C++ libraries that externally look like the C# ones but the methods are implemented using standard C++ functions with the boost library or other libraries like the Boehm GC.

At the beginning AlterNative only worked on Windows because some classes from the ILSpy had mixed some functions for the view and some functions from the decompiler so it was unable to run in Linux because the part of the view could not compile. In order to solve that an AlterNative.Core was created which could work in any system that could run C# code. 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 much the translation was near to be successful. There where several major features that AlterNative was not able to translate like the threading, the delegates used in the interruptions, the P/Invokes or some functions like DateTime, Timer and file access to read and write. Many of these features need to be implemented on the proprietary library because

this is linked to the translated program as C# does with its own libraries.
To implement this features the C++ boost library has been used. This library uses standard C++ functions so it can work in any device or operating system capable of execute binaries compiled from C++.
The classes currently implemented either totally or partially are shown in the Appendix in the section A.3 AlterNative System Library.

When IOSharp is translated through this program it will generate the source code necessary to run and execute the original functions. In the figure 5.4 is shown the differences between the original MicroFramework, the first implementation in C# 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 implement the functionalities of C# like the layer named AlterNative Library which is the implementation of the C# classes.

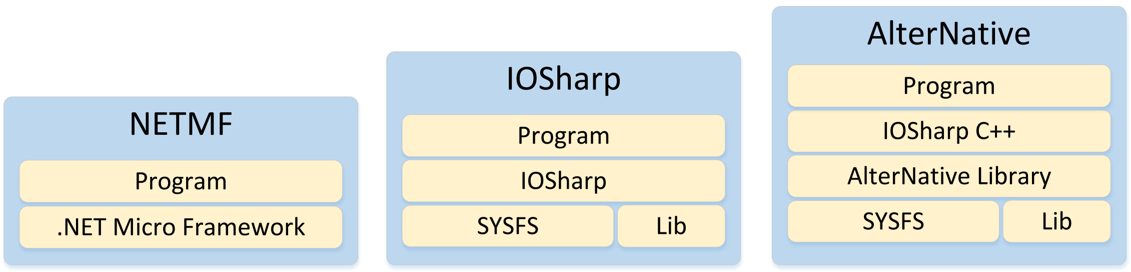


Figure 5.4: Stack in the original .NET Micro Framework, the IOSharp implementation and the AlterNative translation

5.5 Example

The best example to show in this section will be some of the parts of IOSharp translated to C++. In the moment of writing this thesis the only component working well in the translation was the GPIO ports which basically they use the SYSFS of Linux and the interrupt port which uses calls to C is also working.

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

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
    {
        public InputPort(Cpu.Pin portId, bool glitchFilter, ResistorMode resistor)
            : base(portId, glitchFilter, resistor, InterruptMode.InterruptNone)
        {
            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)
        {
            //throw new NotImplementedException();
            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);
        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 {
      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 is faster the translation of IOSharp 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 the results for a GPIO performance tests are shown. This tests will be done using the Raspberry Pi and two implementations of IOSharp, the original written in C# which has been explained on the first part of this thesis, the second implementation tested will be the C++ one translated from the original by AlterNative which has been explained in the second part of this thesis. Theoretically, C++ has a major performance compared to C# but this tests are used to determine how much the C++ implementation gains over the C# ones.
To make the measures in this tests the Logic16 has been used. This is a channel analyzer used to record, view, and measure digital signals. It also currently has 17 different protocol analyzers including SPI, serial, I2C and many more.

6.1 Compilation types

When a compiler generates a binary from a source code normally tries to do some changes to optimize the 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, also some optimizations can be used to make a program consume less power and this is interesting because nowadays smartphones and tablets have a small battery so reaching low power consumption is great to have long battery life. All of this optimizations are carried out by a sequence of optimizing transformations and algorithms high help create programs that uses fewer resources.
Normally compilers can generate programs using optimizations or without using them. If the compiler uses optimizations the compile time will growth but the program will be much more optimized than if the compiler does not apply them.

- **Non-optimized or debug:** this compile 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). For example the .pdb files from Visual Studio are created by the compiler and have the information to debug the created binary. But normally the debug mode will not allow some optimizations because are incompatible to the properly function of the debugger.

- **Optimized or release:** this compile mode is used to generate an optimized binary to execute 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 the debug mode are:

  - **Loop unrolling:** the compiler replaces the code inside a loop (for example inside a for). This helps avoiding the maintenance of the loop variables.
  - **Inlining:** the compiler places the method on the place of the call avoiding the overhead of the calling stack to the method.

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(100,1010) << 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 (100,1010): " << (a>b)? a : b << endl;
   return 0;
}
```

6.2 GPIO

The 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 Logic to start sniffing the second channel which will be the one that performs the output between the two states.
The code used in this test is shown below, both the original version and the translated one using AlterNative.

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");
        }
    }
}
```

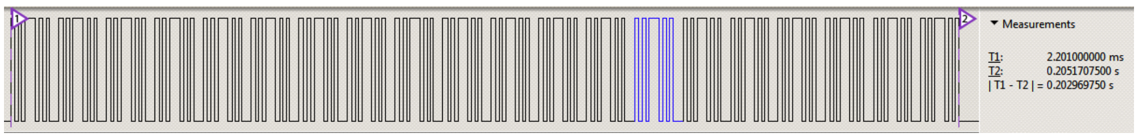
Listing 6.3: GPIO Performance translated to C++

```
#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 will be executed varying the number of iterations between 200 and 10000. Each iteration will swap the port between high and low. Apart from the iteration increase, two type of compilations will be done changing the optimization type between quick compilation without optimization (Debug) and with optimizations (Release).

6.2.1 200 Iterations

The result produced by Mono and the binary compiled in release mode (optimized) shows an irregular pattern consisting of two small pulses a wide pulse and then two small pulses followed by 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 or the thread round-roving that Mono does.
The small pulses are around 1 ms and the wide gaps/pulses are 2 ms. Each block is repeated every 10 ms.

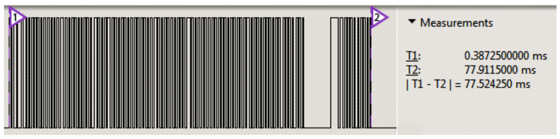


```
T1:             2.201000000 ms
T2:             0.2051707500 s
| T1 - T2 | = 0.202969750 s
```

Figure 6.1: 200 Iterations using C# with optimizations

The figure 6.1 represents 200 changes between state high and low in an OutputPort in the Raspberry Pi under Mono. 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 generated code by AlterNative compiled also in release mode. The resulting test is shown and explained below.



```
T1:             0.3872500000 ms
T2:             77.9115000 ms
| T1 - T2 | = 77.524250 ms
```

Figure 6.2: 200 Iterations using C++ with optimizations

This image is in the same scale that the C# one, so the magnitude of the elapsed time can be compared. In 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 interesting 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 in Mono using both compile types was the practically the same, it takes 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 the compile is done in release mode the time decreases to 78 ms.
From the figure 6.3 it can be concluded that the C++ version is a 62% faster than the Mono version.

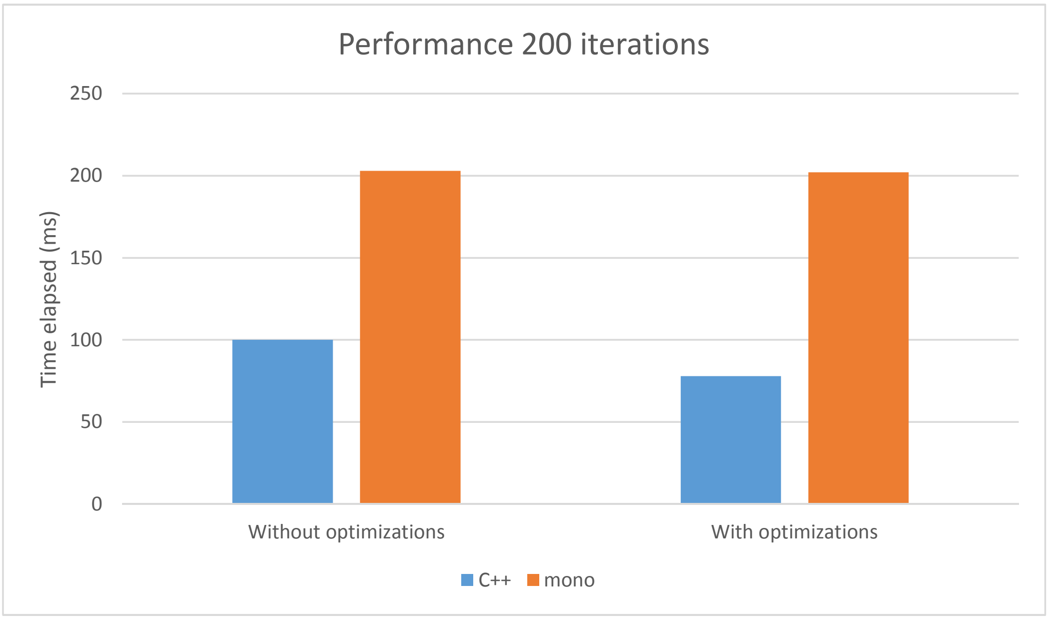
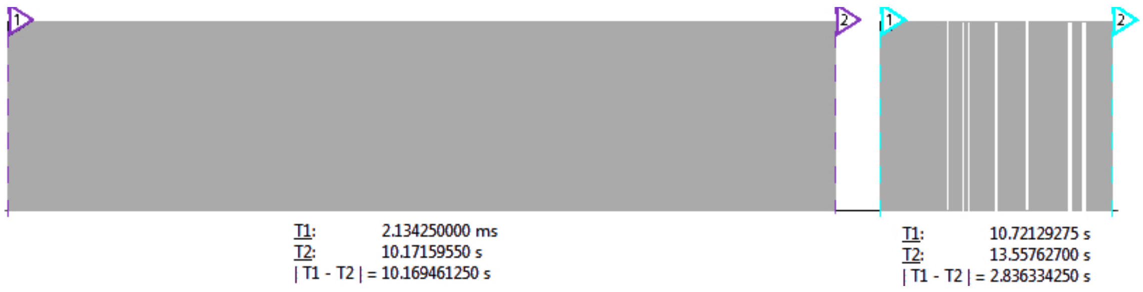


Figure 6.3: Graph showing the elapsed time for the 200 iteration test. Blue is for C++ while orange is C#. On the left is represented the non-optimized compilations and on the right the optimized ones

6.2.2 10K Iterations

After the first test using 200 iterations another one was done, but this time increasing to 10000 iterations or a factor of fifty, in this magnitude the compiler optimizations should be visible enough to observe some kind of improvement on the different compilation 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 10k iterations are done in approximately 3 seconds being three times faster.



```
T1:       2.134250000 ms
T2:       10.17159550 s
| T1 - T2 | = 10.169461250 s
```

```
T1:       10.72129275 s
T2:       13.55762700 s
| T1 - T2 | = 2.836334250 s
```

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.

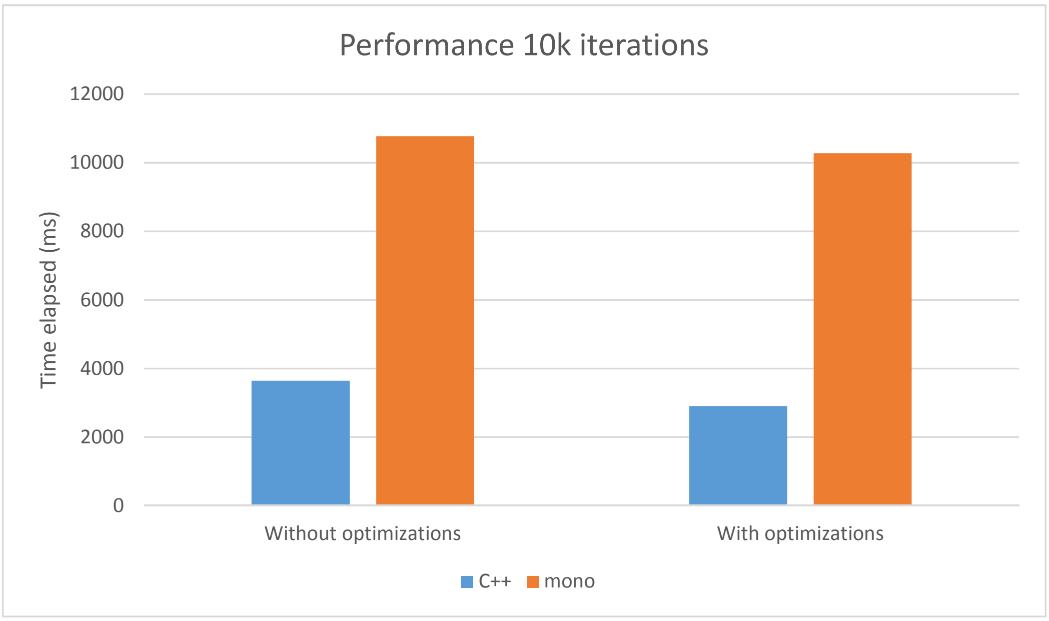
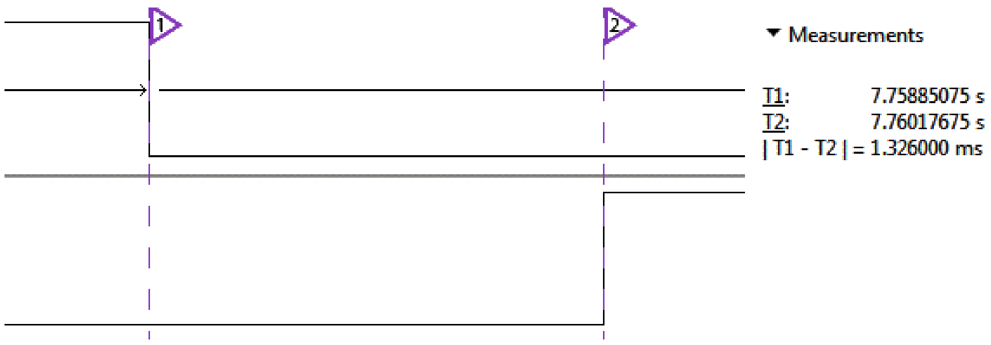


Figure 6.5: Graph showing the elapsed time for the 10k iteration test. Blue is for C++ while orange is C#. On the left is represented the non-optimized compilations and on the right the optimized ones

At this number of iterations the C++ performs much better than C# and also is possible to see an improvement between the debug and release versions of C++ achieving being the optimized version 741 ms faster than the non-optimized.

6.3 Interruptions

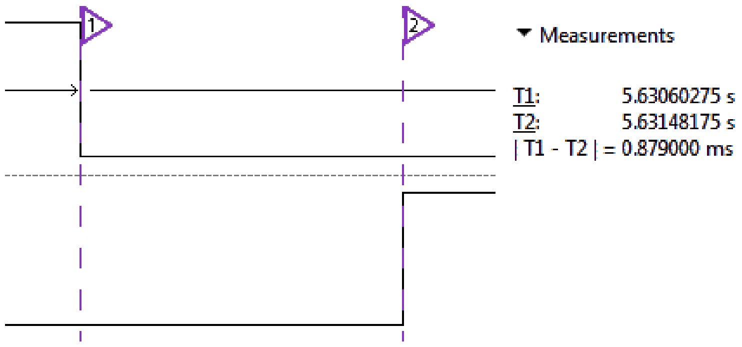
It was observed that in HomeSense the response time between sensor events was poor, probably because of the time that Mono takes between the interruption detection and the response to this interruption. To analyze the elapsed time between the trigger event and the response the example will be polling events from a pin, when an event is detected it will use another pin in output mode sending 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 or the timer. The program will start and create a delegate which will be the called function when an interrupt occurs. After the port instantiations the main thread will be slept and the interrupt will call the delegate if an interruption is present.
In the figure 6.6 is shown the triggering of the interruption (the top channel with the falling edge), after 1.326 ms the second channel is active in state high.



T1: 7.75885075 s
T2: 7.76017675 s
| T1 - T2 | = 1.326000 ms

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

Then the same test is performed but using the C++ version. The result is pretty good because the interruption is attended only 0.879 ms after the triggering. This implies that C++ is 447 Î¼s faster than the Mono version.



T1: 5.63060275 s
T2: 5.63148175 s
| T1 - T2 | = 0.879000 ms

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

It is important to take into account that the interruptions in operating systems are not real time, 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 runs usually Linux. If 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.

CHAPTER 7. CONCLUSIONS

7.1 Project Conclusions

7.2 Future Work

7.3 Environmental Impact

At last but not least it is necessary to talk about the environmental impact of the work described in this document. As 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 was 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 from home installations (i.e. lights, temperature or humidity) to applications capable of detect and analyze different parameters from the environment (i.e. weather stations).

GLOSSARY

BCM2835 ARMv6 CPU mounted on the RaspberryPi. 15

CS Chip Select. 23, 30, 31

File Descriptor file descriptor (FD) is an abstract indicator for accessing a file on POSIX systems. 15, 21, 22

GPIO General Purpose Input Output. 1

I^{2}C I^{2}C. 1

ioctl Abbreviation of input/output control, system call used for device-specific input/output operations. 21

IRQ Interrupt Request. 15

ISM Industrial, Scientific and Medical. 3

MISO Master Input Slave Output. 23, 30

MOSI Master Output Slave Input. 23, 30

POSIX Portable Operating System Interface. X stands for Unix. 51

PWM Pulse-width modulation. 11

RX Reception Channel. 24, 29

SCLK Serial Clock. 23, 30

SoC System-on-Chip. 3

SPI Serial Peripheral Interface. 1, 11, 15, 20

SYSFS Virtual file system provided by Linux. SYSFS exports information about devices and drivers from the kernel device model to user space. 37

TX Transmission Channel. 24, 29

UART Universal Asynchronous Receiver/Transmitter. 1, 11

WSN Wireless Sensor Network. 1, 3

REFERENCES

[1] 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`

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

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

[4] 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

APÈNDIXS

TÍTOL DEL PFC : IOSharp: MicroFramwork a Linux

TITULACIÓ: Grau en Telemàtica

AUTOR: Gerard Solé i Castellví

DIRECTOR: Juan López Rúbio

DATA: January 21, 2014

APPENDIX A. TECHNICAL INFORMATION. LIBRARIES AND DATASHEETS

A.1 spidev.h

A.2 SPI Test. RFID Reader

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.2.1 MFRC522 Datasheet

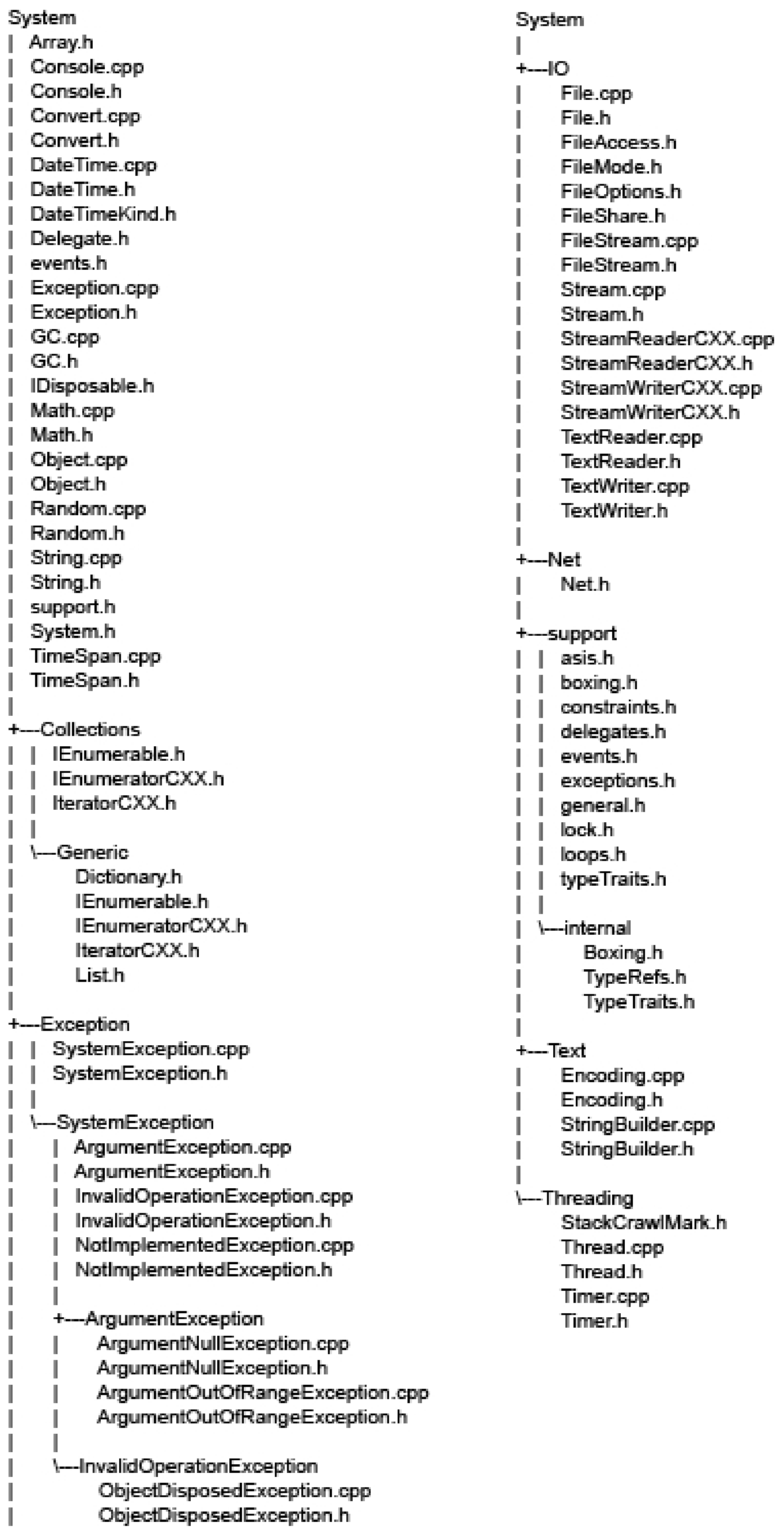
The datasheet of this card card reader can be find at the NXP website or in the following link `http://www.nxp.com/documents/data_sheet/MFRC522.pdf`.

A.2.2 RFID Reader

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

```csharp
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.3 AlterNative System Library



```
System
|   Array.h
|   Console.cpp
|   Console.h
|   Convert.cpp
|   Convert.h
|   DateTime.cpp
|   DateTime.h
|   DateTimeKind.h
|   Delegate.h
|   events.h
|   Exception.cpp
|   Exception.h
|   GC.cpp
|   GC.h
|   IDisposable.h
|   Math.cpp
|   Math.h
|   Object.cpp
|   Object.h
|   Random.cpp
|   Random.h
|   String.cpp
|   String.h
|   support.h
|   System.h
|   TimeSpan.cpp
|   TimeSpan.h
|
+---Collections
|   |   IEnumerable.h
|   |   IEnumeratorCXX.h
|   |   IteratorCXX.h
|   |
|   \---Generic
|           Dictionary.h
|           IEnumerable.h
|           IEnumeratorCXX.h
|           IteratorCXX.h
|           List.h
|
+---Exception
|   |   SystemException.cpp
|   |   SystemException.h
|   |
|   \---SystemException
|       |   ArgumentException.cpp
|       |   ArgumentException.h
|       |   InvalidOperationException.cpp
|       |   InvalidOperationException.h
|       |   NotImplementedException.cpp
|       |   NotImplementedException.h
|       |
|       +---ArgumentException
|       |       ArgumentNullException.cpp
|       |       ArgumentNullException.h
|       |       ArgumentOutOfRangeException.cpp
|       |       ArgumentOutOfRangeException.h
|       |
|       \---InvalidOperationException
|               ObjectDisposedException.cpp
|               ObjectDisposedException.h
```

```
System
|
+---IO
|       File.cpp
|       File.h
|       FileAccess.h
|       FileMode.h
|       FileOptions.h
|       FileShare.h
|       FileStream.cpp
|       FileStream.h
|       Stream.cpp
|       Stream.h
|       StreamReaderCXX.cpp
|       StreamReaderCXX.h
|       StreamWriterCXX.cpp
|       StreamWriterCXX.h
|       TextReader.cpp
|       TextReader.h
|       TextWriter.cpp
|       TextWriter.h
|
+---Net
|       Net.h
|
+---support
|   |   asis.h
|   |   boxing.h
|   |   constraints.h
|   |   delegates.h
|   |   events.h
|   |   exceptions.h
|   |   general.h
|   |   lock.h
|   |   loops.h
|   |   typeTraits.h
|   |
|   \---internal
|           Boxing.h
|           TypeRefs.h
|           TypeTraits.h
|
+---Text
|       Encoding.cpp
|       Encoding.h
|       StringBuilder.cpp
|       StringBuilder.h
|
\---Threading
        StackCrawlMark.h
        Thread.cpp
        Thread.h
        Timer.cpp
        Timer.h
```

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