Design and Development of a Software Operating Platform for the eMote based on .NET Microframework

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Kartik Natarajan

Graduate Program in Computer Science and Engineering

The Ohio State University

2011

Master's Examination Committee:

Dr. Rajiv Ramanath, Advisor

Dr. Jay Ramanathan
Copyright by

Kartik Natarajan

2011
Abstract

Once deployed wireless sensor networking devices need to operate over a long period of times unattended, thereby making low power consumption an important requirement. These devices typically have less memory and clock frequencies in order to achieve lesser power draw. Due to limited resources these devices pose constraints on running a traditional operating system. The .NET Micro Framework (MF) is an open source bootable runtime for such resource-constrained devices. The MF brings the security and reliability of managed code to these platforms. With a memory footprint of few hundred kilo-bytes MF makes an ideal wireless sensor operating platform. It also provides ability to run a high level language such as C# on these platforms. This provides developers with programming constructs that can significantly improve development times in an embedded application development space.

As a part of this thesis we evaluate the hypothesis that the .NET Micro Framework is adequate as an operating platform for wireless sensor networking devices in that it provides platform support, resource management (of power, memory, I/O etc.), a safe and secure runtime and services for application development, and can be ported to a new hardware platform in a short time (six weeks or less). We explore this question through a port of .NET MF v4.0 to an ST Micro’s development board (STM3210E) a state of the art 32-bit embedded architecture – powered by the ARM Cortex M3
microprocessor. We provide insight on the steps that can be taken as part of the porting process to incorporate open source compilers and the pitfalls that can be avoided while debugging and testing the various components during the port.
Dedication

This document is dedicated to my family and friends.
Acknowledgments

First, I would like to thank Dr. Anish Arora and Dr. Kenneth Parker for providing me an opportunity to be a part of The Samraksh Company and for their valuable inputs at various critical points in the project and mentoring me throughout the project. I am also very thankful to Dr. Rajiv Ramnath and Dr. Jay Ramanathan for giving me an opportunity to be a part of CETI.

I would also like to thank Nived Sivadas and Nick Knudson who collaborated with me at various stages of this project. Dr. Sandip Bapat has been really helpful in providing me with some great ideas and helped me comprehend some of the complex development and debugging issues. All of their contribution was important in performing this work.

Finally, I would like to thank my parents, Shwetha Mureli and my friends (Amit Modi, Rohit Patali, Ashwin Guru, Raghuram Onti and Sharada Patil) who have inspired me and provided me with unconditional support and motivation.
Vita

2007..........................................................B.E., Information Technology and Management, SASTRA University, Tamil Nadu, India

2009 ..........................................................Software Engineer, Wipro Technologies

2010..........................................................Graduate Research Fellow, CETI

Fields of Study

Major Field: Computer Science and Engineering
Table of Contents

Abstract .............................................................................................................................. ii
Dedication ......................................................................................................................... iv
Acknowledgments ............................................................................................................ v
Vita ..................................................................................................................................... vi
Table of Contents ............................................................................................................. vii
List of Figures .................................................................................................................. x
Chapter 1: Introduction .................................................................................................... 1
1.1. Introduction ............................................................................................................... 1
1.2. Motivation .................................................................................................................. 3
1.3. Problem Statement .................................................................................................... 4
1.4. Contribution ............................................................................................................... 4
1.5. Related Work ............................................................................................................ 5
Chapter 2: Operating System Concepts for a Sensor Node ............................................. 7
2.1. Fundamental Concepts ............................................................................................ 7
2.2. Tiny OS and the .NET Micro Framework – A comparison ...................................... 10
Chapter 3: The Hardware Platform

3.1. Introduction to the ARM Cortex-M3

3.2. STM 3210F103ZE The Microcontroller

3.3. The System Architecture

3.4. The Memory and Peripherals

Chapter 4: The Software Platform

4.1. Introduction

4.2. The Structure of the .NET Micro Framework

4.3. The Hardware Abstraction Layer (HAL)

4.4. The Platform Abstraction Layer (PAL)

4.5. The Common Language Runtime

4.6. The Class Libraries

4.7. The Interoperation

4.8. Visual Studio Integration

Chapter 5: The Port

5.1. The Environment

5.2. The Tools

5.2.1. The Tool Chain

5.2.2. Debugging Tools
5.2.3. Testing Tools ................................................................. 44

5.3. Structure of the Porting Kit .................................................. 46

5.4. A Step by Step walk through to Porting .................................. 48

5.4.1. HAL Drivers ................................................................. 52

5.4.2. HAL Extension (Interops) ............................................... 57

Chapter 6: Reflections, Conclusion and Future Work ........................... 59

6.1. Reflections ................................................................. 59

6.2. Conclusion ............................................................... 60

6.3. Future Work ............................................................. 61
List of Figures

Fig 1: The Cortex-M3 processor.................................................................16
Fig 2: The Device Memory Map...............................................................17
Fig 3: The Cortex-M3 Exception Table.....................................................19
Fig 4: The Cortex Execution Modes.........................................................20
Fig 5: The Cortex-M3 Three State Prefetch..............................................21
Fig 6: The Internal Bus Structure and the Peripherals of the STM32............24
Fig 7: The STM32 Memory Map.............................................................25
Fig 8: The Structure of the .NET Micro Framework....................................29
Fig 9: The Structure of the CLR runtime loop in .NET Micro Framework........33
Fig 10: The Interoperation Execution......................................................37
Fig 11: Visual Studio with Micro Framework SDK and the Device Emulator.....38
Fig 12: The OpenOCD in session............................................................43
Fig 13: The GNU Debugger in session....................................................44
Fig 14: The ANT16..............................................................................44
Chapter 1: Introduction

1.1. Introduction

Wireless Sensor Networks (WSNs) consist of a large group of miniature computers that are dispersed and networked. These miniature computers are low power, resource-constrained, embedded, computing devices capable of identifying themselves, and known as motes. These motes collaborate together to accomplish tasks such as tracking or sensing which cannot be accomplished by few powerful sensors or computers [1] effectively.

Motes are used in environmental and structural health monitoring, medical sensing applications [2], target tracking etc. When deployed, motes are desired to operate unattended for long periods of time. Platforms such as the TelosB [6] and MICAz [3] can be used for long-term deployments due to their low power consumption; however, their computing capabilities are highly limited. For example, a TelosB mote (which is powered by an MSP430x1xx) runs at 8Mhz and has only 10KB of RAM and 48KB for ROM. On the other hand there are platforms such as the iMote2 [5] which trade off higher computational power and memory sizes for energy efficiency and therefore cannot be used for long term deployments. These constraints
thereby restrict the use of any traditional embedded operating system on these devices.

The .NET Micro Framework (MF) offers a complete development and execution environment that brings the productivity of modern tools to this class of devices [7]. It includes a smaller version of the Common Language Runtime (CLR) and supports development in C# for these devices. This platform provides services of a traditional operating system in terms of process scheduling, power management, I/O etc. The MF can be ported and made to run directly “on the bare metal”; it supports common embedded peripherals and interconnects such as flash memory, GPIO, I2C, SPI, USB etc. The MF is optimized for energy efficiency thereby keeping the power consumption to a minimum. The port of the MF to a WSN platform can provide lower development times, OS-like facilities, and the reliability of managed code, all with lower power consumption.

As a part of this thesis, we port the .NET MF v4.0 to an STM32 platform known as the STM3210E a hardware platform that is similar to the eMote. This platform is powered by a Cortex M3. The Cortex M3 is a 32-bit ARM processor based on the ARMv7-M architecture that is designed specifically for high system performance combined with low power consumption. The costs are low enough to challenge traditional 8 and 16-bit microcontrollers. We then evaluate the adequacy of the .NET MF as a platform for WSN and talk about various debugging and testing techniques involved in the porting.
1.2. Motivation

The problem of hardware resource constraints on the WSN platforms restricts the use of traditional operating platforms. There are other platforms such as the TinyOS which provide an event driven operating environment which fits well within the hardware specifications. It also uses a special language (nesC) using which composes a system of smaller components which are statically linked with the kernel to form the firmware. But there are various limitations to programming in nesC such as the programmability is hard, steep learning curve, static memory allocation model etc. In comparison porting the MF on the Cortex M3 provides advantages in terms of savings in costs and power consumption of the device (due to the design of the microprocessor) and also allows developers to write applications in C# for these platforms. As a result of which the developer is given an advantage of using a high level language and exploiting its rich set of features for faster application development.

Apart from providing ease of programmability and efficient use of hardware porting of the MF on a WSN frees the programmer of the burden to have a deep understanding of the hardware for which the application is being written.

1.3. Problem Statement
There are two issues that this thesis tries to address, First one is to develop a software operating platform for the eMote which is based on STM3210E by porting the .NET MF v4.0. This port would enable application developers to program and run applications in C# for the eMote. The port evaluates the adequacy of the .NET MF v4.0 as a platform for WSN. In places where this platform seems inadequate a new set of interfaces has been introduced at the HAL level and have been exposed to the C# as a set of user defined libraries. Secondly, the thesis tries to describe the issues that were faced while porting during development, debugging and testing. The claim is that if these issues are addressed at the right time using the right set of tools the porting time can be reduced.

1.4. Contributions

The thesis contributions are as follows,

- The port of the .NET Micro Framework v4.0 on a STM3210E development board (powered by ARM Cortex M3), which will form the software operating platform for the eMote.

- The ability to develop C# applications which can utilize various hardware peripherals of this platform.

- In this port we also provide the ability to extend the Hardware Abstraction Layer interfaces and expose them to C# using interoperation for this platform.
A description of pitfalls in terms of development, debugging and testing issues which if addressed at the right time can speed up the process of porting the MF. We also describe ways in which these pitfalls may be addressed.

1.5. Related Work

There have been other cases where the .NET MF has been ported on a hardware. Specifically for the purpose of WSN a port on the iMote2 has been provided by Crossbow. The only disadvantage of using the iMote2 is the higher power consumption which comes as a trade off for higher computational power and higher memory capacity [2]. There are other instances where the MF has been ported to a more general hardware platform such as the FEZ Panda [8] which is powered by a 32-bit ARM7 processor running at 72Mhz (NXP LPC2387) with 512KB of flash and 96KB of RAM and the Netduino [9] powered by 32-bit Atmel processor (48Mhz) with 128KB of flash and 60KB of RAM.

The rest of this thesis is organized as follows. In Chapter 2, we present the background of sensor node operating system concepts and provides a comparative overview of MF and the TinyOS. In Chapter 3, we introduce the ARM Cortex M3 and the STM3210E and talk about the peripherals and memory specification. In Chapter 4, we introduce the .NET Micro Framework, the structure of the software stack, the Common Language Runtime, the Platform Abstraction Layer and the Hardware
Abstraction Layer. In Chapter 5, we describe the process of porting in detail starting from the environment set up to the development tools, we then describe the various HAL modules that were implemented as a part of this port, we also talk about the various debugging and testing strategies and the pitfalls that were encountered while developing these modules. This chapter also describes the process of providing HAL extensions to access the peripherals that are not part of the standard HAL from MF from C#. In Chapter 6, we describe the results of running the standard tests executed on the port for validation. Finally in Chapter 7 we summarize the contributions and provide a discussion of our findings and present some ideas for future research.
Chapter 2: Operating System Concepts for a Sensor Node

This chapter describes the fundamental principles that govern the structure and working of an operating platform on a sensor node.

2.1. Fundamental Concepts

Process Management –

A process is essentially an active program. Process Management consists of the steps that orchestrate the lifecycle of a process by an operating system. It orchestrates the stages that a process needs to go through in order to finish execution. A typical OS scheduler switches task between Ready, Running and Blocked states. These stages are determined on a number of factors such as the CPU availability, I/O access, memory access etc. A traditional operating system provides services such as context switching and inter-process communication and administers the resource sharing and coordination amongst processes. When all the processes are well managed they provide accurate interrupt handling mechanisms, the handling of queues, and synchronization (to avoid data race conditions). In an environment (such as mote) where power management is crucial, the design of an efficient process management subsystem (a scheduler) is critical.
Storage Management –

Storage Management is an important design aspect of the sensor network operating platform. As these devices tend to have very less memory capacity the operating system design needs to take care of the size of the firmware (which includes the kernel and the peripheral drivers). These operating systems are mostly written in C because it gives various advantages such as faster execution and smaller memory footprint. Also, most of the modules of the firmware are linked statically i.e. once the modules are linked and compiled into a binary modifying the system is not possible. Apart from loading itself into the memory (ROM) the OS needs to provide higher layers with interfaces which can store data (in RAM and ROM) depending on the requirements. As these devices are used for the purpose of tracking and sensing events should be able to read/write data from/to memory.

Power Management –

Power Management is an important factor for mote like platforms because these motes once deployed run on batteries and the operating system must be able to provide services power management services such as sleep and wake up. Such interfaces allow the mote to sleep 99% of the times there by conserving the battery power on the mote and simultaneously provide adequate amount power supply to the mote.
Input/Output (I/O) System –

I/O subsystem in a WSN platform deals with providing interfaces to access the I/O pins on the hardware. These pins can be configured for various purposes such as interfacing with other devices, triggering active high on an input pin to indicate an event of interest, sending output signal to a peripheral etc.

Memory Management –

Memory management is the way of managing the system memory at run time. Most of these devices do not have a Memory Management Unit (MMU) at the hardware so the memory management module is responsible for providing abstractions to handle it at the software. Typically the operating system fetches the instructions from a fixed memory location processes it and stores the result back. The instructions in this case are binded to fixed addresses. The instructions and data are binded to addresses either at compile time or at runtime. In compile-time binding the addresses are known before hand and the instructions are binded to these addresses as a part of the build process. On the other hand, the runtime binding the actual binding is delayed until the program is ready to be loaded and executed. The OS typically deals with program memory and flash memory. It provides interfaces to allocate and free memory based on program requirements.

Security Management –

Security is an important aspect of the platform that supports running of WSN applications. These applications often provide functions and data that can be sabotaged
by malicious entities such as viruses, trojan horses etc. Even though providing a robust security architecture is hard to provide keeping in mind the computational abilities and the cost sensitivity of these devices [10], a layer of security is provided as part of these operating systems such as tamper resistance mechanisms and application signing. These services improve reliability of the system [11].

2.2. TinyOS and the .NET Micro Framework – A comparison

TinyOS is component based operating system that is designed around a static resource allocation model for resource constrained sensor network nodes [12]. TinyOS is designed for the WSNs which execute concurrent and reactive programs and operate under severe memory and power constraints [14, 15]. The system runs in an event driven way with the scheduler being a single thread of execution. An execution is triggered when an event causes tasks to be posted on the system queue. The scheduler sleeps most of the time unless an event causes it to wake up there by conserving on the energy. Apart from the scheduler, the complete system configuration consists of a graph of components. A component has a set of command handlers (non-blocking requests made to the lower level to perform service of execution of a task), set of event handlers (deal with the hardware events which signal completion of a service), a frame and a bundle of simple tasks. Tasks, commands and handlers execute in context of a frame. The frames are statically allocated and thus the memory requirements are known at compile time. In the
system the work is performed as tasks, which are atomic with respect to each other and preemptive. A task can call lower level commands and signal higher level events. The tasks share a single stack which is assigned to the currently executing task. This is essential considering the memory constraints on these devices [14].

In TinyOS, the small amount of processing which are required to service the hardware event can be processed immediately whereas, long running running tasks are interrupted. The task scheduler executes tasks in a FIFO manner and allows the tasks to be preempted (in case of interrupts from hardware). Uninterrupted execution is possible with the help atomic section primitives which disable the interrupts before going into execution.

The .NET Micro Framework is highly optimized for small and resource constrained devices such as the WSN nodes. The port of the .NET MF makes it possible to use C# to program embedded systems. The MF uses a layered architecture which is extremely flexible and adaptable to new hardware platforms [16].

The Runtime Component Layer (RCL) includes three components [16], the Common Language Runtime (CLR) which is the runtime environment of the MF, the Platform Abstraction Layer (PAL), which comprises of the C++ interfaces which are called by the CLR and the Hardware Abstraction Layer (HAL) which comprises of functions that are purely hardware dependent.
Many drivers in the MF exist as pair, both the PAL and the HAL consist of a driver and these drivers are used together to accomplish a particular task. In such cases the CLR calls the PAL interfaces which in turn call the HAL interfaces for hardware specific requests [16].

The .NET MF includes a Class Library layer which is an object oriented collection of reusable types. These libraries can be used by the application developers to write applications in an embedded space. The MF makes it possible for the vendors who port the framework to provide additional class libraries which are supported by the hardware. Finally the top layer of the MF consists of the application layer which is a managed application created for the hardware.

The managed applications are executed by the CLR. The CLR is a single thread of execution and owns all the memory in the system. During normal operation the CLR iterates in an interpreter loop and schedules managed threads using a round robin scheduling policy. During the allocated time quota the interpreter executes the intermediate language that belongs to the stack of the managed thread. When the thread sleeps or waits for a synchronization primitive (such a monitor) or an event that is not signaled, the CLR moves the thread to the wait queue and tries to schedule another thread for execution. The CLR checks for any hardware events amidst scheduling of threads. If so, the CLR dispatches the event to the thread that requested it. If at any point there are no threads to be scheduled, the CLR sets a time out value and goes to sleep in order to conserve power. In case the CLR is woken up by an interrupt it performs the required
processing. If the CLR is woken up by the timer, the CLR executes the pending completions (tasks that occur at a time specified by the user) and checks again for interrupts, at this point if there are no interrupts to be processed the CLR goes back to sleep [16].

**NesC v/s C# for programming in the embedded space –**

TinyOS is written in and supports running of a language known as the NesC (a C based language) [13]. The major goals for NesC were to allow for strict checking at compile time and ease of programming of the TinyOS components. It uses macros to create command and event interfaces between components and to express the component graph itself. NesC is a programming model that supports reactivity, concurrency and communication [13] making it an ideal choice for sensor network platforms. There are various challenges that the language addresses such as the applications are not interactive or batch driven, they are driven by events that are caused by changes in the environment. It also needs to deal with challenges of concurrency (such as processing packet arrival and servicing reactive events) on a platform with limited resources [13]. One of the main advantages of NesC is that it’s a static language [13]. The call-graph is fully known at compile time and this makes analysis and optimization simpler and accurate (by providing inlining, which saves memory and eliminating unreachable code) [13]. NesC does not offer function pointer support and dynamic memory allocation. It is also limited in terms ease of programmability. The programming constructs provided are much less
intuitive than the C# counter parts. NesC also poses portability restrictions i.e. a similar application cannot run on two different hardware platforms running TinyOS.

The .NET MF on the other hand provides support for writing and executing applications in C#. Having C# as the choice of language simplifies development in terms of providing programming constructs which significantly reduce development times. C# provides memory protection, exception handling, native types, threading support, automatic garbage collection, type safety and extensive library support. The C# code is interpreted and not just-in-time compiled in the MF. This is because of the memory constraints posed by these devices. The interpreted flow execution goes through the following steps for execution, interpreted instruction is fetched and decoded, corresponding calls to compiled C/C++ functions are made and the function operates on the values on CLR’s stack or heap. As a result, (of interpretation) the execution tends to be much slower than executing the same code in native. Also, applications which are written in native code and compiled in machine code tend handle data in its most compact fashion so the memory usage and behavior is easier to comprehend. Whereas, for high level languages such as C# use complex data types for everything (objects) which takes up more memory. These factors make C# unsuitable for hard real time systems. But C# tends to be much more intuitive in for application developers, the programming constructs are more understandable than the NesC counterpart. The applications written in C# are portable (which do not user third party class libraries). Thus, making it a better choice for a platform which do not tend to be hard real time.
Chapter 3: The Hardware Platform

In this chapter we describe the hardware platform for the eMote. The port is originally performed on a STM32 evaluation board (STM3210E) which is powered by the ARM Cortex M3 platform. The hardware specifications of this development board are similar to the eMote and the port can be loaded and run on the eMote out of the box.

3.1. Introduction to the ARM Cortex M3

The Cortex-M3 is a RISC based processor (having load-store architecture) which is designed for high system performance combined with low power consumption. The cost of the Cortex M3 is comparable to 8-bit/16-bit microcontrollers. These factors make it an ideal choice for the sensor network platform. The Cortex M3 is based on Harvard architecture [17] and thus has multiple buses which allow operations in parallel.
It has a 4Gigabytes linear address space which is split into regions for code, SRAM, peripherals and system peripherals. The Cortex M3 allows unaligned data accesses which ensures efficient usage of the internal SRAM. The processor also supports bit-banding which supports setting and clearing of bits within two 1Mbyte regions of memory.
The first 1Gbyte of memory is split evenly between a code region and a SRAM region. The code space is optimized to be executed from the I-Code bus and the SRAM is reached in the D-code bus. The code can be run from the SRAM but is slower as it incurs an extra wait state. The code can be run faster from the on chip memory located in the code region. The port of the Micro Framework to the e-Mote presently runs from the in chip flash memory. The 0.5Gbyte of memory houses all the peripherals provided by the vendor. The next 2GByte of address space is allocated to external memory-mapped SRAM and peripherals. The final 0.5GByte is allocated to the Cortex processor registers.
The ARM& and ARM9 instruction sets are capable of accessing byte, half word and word signed and unsigned variables. This enables CPU to support integer variables without any software library support. Also, Cortex-M3 allows unaligned memory accesses which allow compiler tool chain better ability to order program data in SRAM.

The Cortex M3 provides a Nested Vector Interrupt Controller (NVIC). The NVIC is the standard structure for interrupt handling in a Cortex based platform. It provides dedicated interrupt vectors for up to 240 peripheral sources which can be individually prioritized. It thus enables fast interrupt handling (it takes about 12 CPU cycles between the occurrence of the interrupt and reaching the first line of code). In case of back to back interrupts the Cortex uses a tail chaining method that allows interrupts to be served only a 6 CPU cycles latency. Also, the high priority interrupt can pre-empt a low priority interrupt without incurring any extra CPU cycles. The interrupt structure is tightly coupled with the low power modes of the Cortex M3. There are ways to configure the CPU to automatically enter sleep mode after interrupt handling and stay asleep till the next interrupt. The NVIC is designed to efficiently handle multiple interrupts. It also allows pre-emption to allow higher priority interrupts to be processed faster. The NVIC configuration needs three things, the vector table needs to be configured for interrupt sources, the NVIC registers need to be enabled and the priorities have to be set and finally the peripheral should be configured with it interrupt support enabled. The Cortex-M3 exception vector table starts at 0x00000004 and looks like as follows,
The vector table entries are 4 bytes wide and hold the start address of each service routine associated with the interrupt. The first 15 entries are for exceptions that occur with Cortex. The peripheral interrupts start at 16 and will be linked to peripherals defined by the vendor. In the firmware the vector table is maintained in an assembly file called the VectorTrampolines.a.

Even though the Cortex M3 is a low cost core it still has a 32-bit MCU and has support for two modes of operation, the Thread mode and the Handler mode. The CPU will run in Thread mode while it is executing in non-interrupt background mode and will switch to the Handler mode when it is executing exceptions. The Cortex core can execute code in privileged and non-privileged modes. In privileged mode the CPU has access to

---

**Fig 3: The Cortex-M3 Exception Table [17]**

<table>
<thead>
<tr>
<th>No.</th>
<th>Exception Type</th>
<th>Priority</th>
<th>Type of Priority</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reset</td>
<td>-3</td>
<td>fixed</td>
<td>Reset</td>
</tr>
<tr>
<td>2</td>
<td>NMI</td>
<td>-2</td>
<td>fixed</td>
<td>Non-Maskable Interrupt</td>
</tr>
<tr>
<td>3</td>
<td>Hard Fault</td>
<td>-1</td>
<td>fixed</td>
<td>Default fault if other handler not implemented</td>
</tr>
<tr>
<td>4</td>
<td>MemManage Fault</td>
<td>0</td>
<td>settable</td>
<td>MPU violation or access to illegal locations</td>
</tr>
<tr>
<td>5</td>
<td>Bus Fault</td>
<td>1</td>
<td>settable</td>
<td>Fault if AHB interface receives error</td>
</tr>
<tr>
<td>6</td>
<td>Usage Fault</td>
<td>2</td>
<td>settable</td>
<td>Exceptions due to program errors</td>
</tr>
<tr>
<td>7-10</td>
<td>Reserved</td>
<td>N.A.</td>
<td>N.A.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>SVCall</td>
<td>3</td>
<td>settable</td>
<td>System Service call</td>
</tr>
<tr>
<td>12</td>
<td>Debug Monitor</td>
<td>4</td>
<td>settable</td>
<td>Break points, watch points, external debug</td>
</tr>
<tr>
<td>13</td>
<td>Reserved</td>
<td>N.A.</td>
<td>N.A.</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>PendSV</td>
<td>5</td>
<td>settable</td>
<td>Pendable request for System Device</td>
</tr>
<tr>
<td>15</td>
<td>SYSTICK</td>
<td>6</td>
<td>settable</td>
<td>System Tick Timer</td>
</tr>
<tr>
<td>10</td>
<td>Interrupt #0</td>
<td>7</td>
<td>settable</td>
<td>External Interrupt #0</td>
</tr>
<tr>
<td>256</td>
<td>Interrupt #240</td>
<td>247</td>
<td>settable</td>
<td>External Interrupt #240</td>
</tr>
</tbody>
</table>

---


the full instruction set where as in the non-privileged mode access to certain instructions is disabled (such as MRS and MSR instructions). The following table summarizes the two modes of the CPU core.

![Fig 4: The Cortex-M3 Execution Modes [17]](image)

When pressed the reset the CPU core both thread and handler mode execute in privileged mode and there are no restrictions on access to any processor resources and both the modes use the main stack. The core needs the reset vector and the start address of the stack to start execution. This aides sophisticated software design and implementation on these processors. This is because system code and the application code can be partitioned and errors in the application code will not cause the OS to crash (if OS is configured to run in Handler mode with privilege mode).
The Cortex M3 also offers support for executing Thumb-2 instruction set; this provides higher code density by blending 16 and 32 bit instructions. The Thumb-2 instructions have improved multiply instruction which can execute in single cycle and a hardware divide takes about 2-7 cycles. The instruction set is a rich instruction set that is designed as a target for C/C++ compilers. The Cortex core can execute most instructions in a single cycle which is achieved by three stage pipelining. While one instruction is being executed another one is being decoded and the third one is being fetched from the memory.

![3 Stage Prefetch](image)

Fig 5: The Cortex-M3 Three State Prefetch [17]

The Cortex core also supports branch prediction; this means that unlike ARM7 and ARM9 cores which flush and refill code in case of a conditional branch, the Cortex core performs a speculative fetch so that both destinations of a conditional instruction are available for execution without a performance penalty. The Cortex M3’s register file has
sixteen 32-bit registers. R0 to R12 are simple registers, R13 is a stack pointer, R14 is the link register and R15 is the program counter.

The Cortex core also has integrated sleep modes this makes it ideal for WSN platforms because 90% of the times the WSN motes are in a deep sleep mode. The sleep mode places the core in its low power mode and halts execution of instructions within the CPU. A part of interrupt controller is kept awake so that interrupts are generated from the peripherals can wake the core up. The sleep mode can be entered by executing the Wait For Interrupt (WFI) or the Wait For Event (WFE) instruction. The core in case of WFI resumes execution and serves the interrupt and based on value of the SLEEPON EXIT bit either resumes normal code execution or goes back to sleep. In case of a WFE the core resumes execution from the point it was places into the sleep mode.

3.2. STM3210F103ZE – The Microcontroller

This section explains the hardware development board and the STM32 chip on the board on which the port was performed. The STM3210E-EVAL board comprises of STM3210F103ZE chip (with 144 pins). This requires a single power supply of range 2.0V to 3.6V. The internal regulator is used to generate a 1.8V supply for the core. The board has two other optional power supplies. The Real Time Clock (RTC) and a set of registers are located on the battery domain (they can preserve data when core is in a deep sleep mode). The second power supply is used to power the Analog to Digital Convertor.
The chip also contains an internal reset circuit that holds the device in reset as long as VDD is below 2.0V.

The chip can be started in three different boot modes. These are selected by two external pins (BOOT0 and BOOT1). Based on the selection the uC will alias different areas of the memory map to the bottom of the memory there by giving capabilities to execute code from user FLASH, internal SRAM or system memory. If booted from the system memory the chip executes a factory programmed boot loader which allows user flash to be reprogrammed. Further details about the booting and the memory map as explained in the further sections.

3.3. The System Architecture

The chip is composed of a Cortex core which is connected to the Flash memory by the dedicated Instruction bus. The Cortex Data and System busses are connected to the Advanced High Speed Bus (AHB). The internal SRAM and Direct Memory Access (DMA) are directly connected to the AHB bus matrix. The other peripherals are located on the two Advanced Peripheral Buses (APB1 and APB2) and the APB is bridged on to the AHB. The AHB runs are the same clock speed as the Cortex core. It also has prescalers which enable clocking at lower speeds in order to conserve power. The speed of the APB1 is limited to 36 MHz where as the APB2 can run at full 72MHz. The Cortex core and the DMA can arbitrate the bus. The internal bus structure looks as follows,
3.4. The Memory and Peripherals

As mentioned earlier the chip offers a linear 4Gbyter address space despite having many internal busses. The program memory of the chip starts at 0x00000000 (the map has to conform to the standard cortex layout). The SRAM (on chip) starts are 0x200000000 and all the internal SRAM is located in the initial bit band region. The user peripherals are memory mapped starting from 0x400000000. Finally the Cortex registers are at their standard locations (from 0xE0000000). The STM32 memory maps is shown below,
The Flash memory region is composed of three sections, the user flash (starting at 0x00000000), system memory (4K of the Flash memory) which is programmed by the factory boot loader and the final section from 0x1FFFF800 which has optional bytes that allow to configure system settings. The memory section can be selected based on the values of the BOOT0 and BOOT1 pins.
The chip also has internal and external oscillators. The internal oscillators are capable of supplying clock to the internal PLL. This allows the uC to run at its maximum frequency at 72 Mhz. The low speed internal runs as 32.768KHz and the high speed internal runs at 8Mhz. The external oscillators are of low speed (32.768KHz, used as a source for RTC and Windowed Watchdog) and high speed (of range 4-16Mhz with PLL multiplier to derive 72Mhz) as well. The external oscillators are used to derive the Cortex processor and peripheral clocks. The core can be clocked either by the Internal or External oscillators or from an internal Phase Locked Loop (PLL). The only downside of using the internal oscillator is that it is not so accurate and stable (the 8Mhz clock source). The oscillator, PLL and the bus configuration registers are located in the Reset and Clock Control group (RCC). After the initial reset the CPU derives its clock from the high speed internal oscillator by default which can be changed by the software after that core has been started up. The high speed external oscillator is switched on and the system waits until is stabilizes. Once the oscillator is stable it can be selected as the input to the PLL. The output frequency of the PLL is defined by selecting an integer multiply value. Once the PLL is stable the CPU clock source can be switched to PLL.

After the PLL has been selected as the system clock the CPU can be run at 72Mhz. To get the rest of the chip running at optimum speed the AHB and APB buses are configured.

The STM32 chip also supports various peripherals such as the General Purpose Input Output (GPIO) lines, Timers, Analog to Digital Convertor, Digital to Analog
Convertor, External Interrupt line, Universal Serial Bus, Serial peripheral Interface, I2C etc. Each of these peripherals are discussed at a greater detail in the following chapters.
Chapter 4: The Software Platform

In this chapter we discuss about the software platform for the eMote. This platform is based on the Microsoft’s .NET Micro Framework (v4.0). The .NET MF is a lightweight OS abstraction running on bare metal hardware. The .NET MF provides various capabilities to its users, it runs based on the Common Language Runtime which enables users to write applications in C# for WSN platform. The runtime being a VM provides portability of managed code across platforms for such devices. Further sections in this chapter talk in detail about this platform.

4.1. Introduction

The .NET MF is a minimal operating system which is lightweight and has a memory footprint of about 500KB. The CLR when ported on to the hardware runs as a bootable module and provides services such as managed thread scheduling, type checking, garbage collection, execution of managed code etc. The MF can run on hardware without Memory Management Units (MMU). The CLR supports minimal .NET class library which is tailored for embedded applications including GUI classes etc. It has specific libraries at the C# in the form of Dynamic Linked Libraries (DLL) for common hardware and interconnects such as memory, GPIO, I2C, USART, SPI etc. The
.NET MF also provides a managed driver model which enables user to write drivers from C#. It also provides integration with Visual Studio for development, debugging, code deployment capabilities and an emulator. So the primary strengths of MF include, lower hardware costs than other managed platforms, lower development costs than other embedded applications and lower power consumption [18].

4.1. Structure of the .NET Micro Framework

The following figure shows the components that make up the .NET Micro Framework

![Diagram of .NET Micro Framework](image)

Fig 8: The Structure of the .NET Micro Framework [16]
The bootable runtime services are provided by the Hardware Abstraction Layer (HAL) and the Platform Abstraction Layer (PAL). These provide services such as environment initialization, interrupt handling, threading and process management, heap management and other support which are necessary to run application programs. These services enable the .NET MF to run directly on the hardware.

4.2. Hardware Abstraction Layer

The HAL forms the foundation layer of the MF which is typically about 20-30KB. This is the only part of the MF which is tightly coupled with the hardware. This layer provides capabilities which enable the applications to access hardware peripherals on the device through queued I/O there by shielding details such as Interrupt Service Routines and queue locking details from it. It also provides infrastructure for bootstrapping the device by providing primitives for starting up CPU clocks, setting up interrupt vector tables, starting up the peripherals on the device etc. The HAL has interfaces to I/O, IRQ’s, Timers etc but does not perform any process scheduling. The process scheduling is performed by the higher layers by using these services. This keeps the design and development of the HAL simple. Due to the simple design of the HAL the power consumption of the device is kept to the minimum. The HAL has two modes of execution, one is the single threaded application mode and the other one is the interrupt service routine mode. The single threaded mode runs the CLR which houses the
execution of managed applications. The interrupt mode is used by the CLR for purpose of multitasking of threads and I/O.

**4.3. Platform Abstraction Layer**

The PAL exposes abstractions such as logical timers (over a single hardware timer), memory blocks, asynchronous communications and a set of data structures (which are used by the CLR for various purposes such as buffered I/O) to the CLR. The PAL comprises of features that are not specific to any hardware. Various modules such as communication buffer API’s, the C runtime, logical timer API’s, memory abstractions, file system abstractions, network service abstractions constitute the PAL of the MF.

**4.4. The Common Language Runtime (CLR)**

The MF CLR implements features such as the Execution engine, Type system and Memory management (allocation, defragmentation, garbage collection etc), Library support (such as numeric types, arrays, delegates, events, references, weak references etc), Reflection, Serialization, Exception Handling, Thread Management. Most of these features are similar to the ones provided by the .NET CLR. Certain features such as limit on call durations posed by execution constraints, string are represented as UTF-8 internally and are exposed as Unicode, value types are emulated by runtime for garbage collection optimization and lower RAM usage, global string tables for well known type
values, no virtual table support, specialized garbage collection, weak reference persistence in flash memory [18].

The CLR provides a virtual machine abstraction for code execution, the application programs are compiled to Intermediate Language (IL) which is then executed by the execution engine at run time. This offers flexibility in terms of running applications on various different hardware platforms. The CLR in the MF provides only interpreting capabilities and not Just In Time (JIT) capabilities for the code execution because of memory and power constraints on these devices. As a result of this interpretation the code has smaller memory footprint and is safer (increased determinism as native code does not have the control) [18]. The major disadvantage of the code interpretation is slower than native execution. The CLR manages the heap which is allocated as part of the RAM memory for object allocation. It keeps track of the various parameters which speeds up allocation and minimizes fragmentation. The CLR also manages the list of logical timers in the system (which are based on one hardware timer known as the system timer). It takes care of updating and scheduling the timer triggers. The other important responsibility of the CLR includes the thread management, the CLR runtime loop is described in the following figure,
As specified earlier, the CLR has only a single thread of execution (and it also owns the memory of the system). While operating normally the CLR iterates in the interpreter loop and schedules managed threads. It uses a round-robin scheduling algorithm [16] and the threads are scheduled based on priority. Each managed thread gets a 20ms time quantum during which the runtime executes the IL that belongs to the stack of that particular managed thread. When the quantum of time expires the thread is put to sleep, if the thread (while executing) waits for a synchronization primitive the thread is moved to the wait queue and the CLR schedules the next thread in the queue. The CLR also checks for any hardware events (e.g., an I/O event) that might have been caused by the HAL. In case
any events need servicing the CLR dispatches the thread that made the request. If no events are caused the CLR processes the next thread. In case there are no threads that need to be processed by the CLR it sets out a timer and goes to sleep in order to conserve power [16].

The MF handles thread priority with its own strategy [16] to avoid starvation and race conditions. The CPU times are assigned to the threads based on their priority using a base-2 logarithmic scale. Thus, the thread at a given priority level gets twice as much CPU than the thread one priority level lower.

The CLR handles interrupts for a managed application (such as I/O) by registering the subscription and the delegate associated. For a synchronous call once the call has been (say a read or a write call) made the CLR blocks the thread and invokes the PAL API to service the request. So the I/O happens asynchronously while other threads are been processed by the CLR. Once the request is processed the CLR is notified by setting system flags known as the Event Flags. The CLR can check the values of these flags and based on the values of these flags the managed thread is dispatched which then continues with the execution in the order specified earlier.

The CLR uses the Asynchronous Procedure Calls (APC) to make all the driver calls look asynchronous. It uses completions and continuations to do this. The completions and continuations are timed tasks that happen at the time specified by the user. The completions are interrupt based and the continuations are executed while the processor is idle. The CLR owns the system timer services implemented at the HAL and utilizes the Time API’s to accomplish this.
4.5. The Class Libraries

The MF includes implementations of the following class libraries,

- System
- System.Collections
- System.Diagnostics
- System.IO
- System.Net
- System.Net.Sockets
- System.Reflection
- System.Resources
- System.Runtime.CompilerServices
- System.Runtime.Remoting
- System.Text
- System.Threading

Apart from these libraries there are various hardware specific libraries provided under the SPOT namespace. The MF also provides facilities to expose hardware functionality which is present in the hardware but does not for form a part of the predefined HAL or PAL by using Interoperation. These Interoperation calls can be defined under a user defined name spaces as well and can form a part of the class libraries available at C#. The following section describes the Interoperation in detail.
4.6. The Interoperation

The interoperation (interops) capabilities in MF allow the user to expose capabilities present in the hardware to C# programs in the form of providing managed code libraries. This is done by defining the HAL functions (in native C++) for a particular hardware (not available as a part of default MF HAL) and linking them with the managed code. This enables the users to add extra functionalities to the standard MF. For the eMote access to hardware peripherals such ADC and Radio are provided by means of interops. The interops provide both synchronous and asynchronous calling capabilities. In case of asynchronous calls the Native Event Dispatcher class sets up and processes the event. This capability adds a lot of flexibility into the MF by letting users extend the basic MF HAL and implement their own drivers for memory mapped and I/O mapped devices (which in some cases are not accessible from the SPI or I2C buses).

The interops requires a user to create a C# class that can be used as the API’s at the managed code level by compiling it as a library, this is passed to a tool called the Metadataprocessor (which is provided as part of the .NET MF and is an important component in the build chain). The Metadataprocessor then creates C++ stubs for these C# API’s which can be linked with the HAL implementation of the driver. The peripheral can be then accessed via the C# calls from the managed code. The following figure depicts the way an interop call is made in MF,


4.7. Visual Studio Integration

The MF comes with the support for Visual Studio. Apart from editing and building code the visual studio provides an emulator for the hardware which lets you debug the program written for the device. The emulator does not emulate the actual CPU but is an Intel-native build of the CLR [18]. This runs on top of the Windows operating system and maps all the HAL calls to Windows APIs. The emulator can be customized to add ports, buttons etc. This is done by simply changing values in an XML file [18]. The
emulator provides facilities for logging, heap analysis and profiling of the device. The below figure shows a screen shot of the VS running along with the emulator,

![Visual Studio with Micro Framework SDK and the Device Emulator](image)

**Fig 11:** Visual Studio with Micro Framework SDK and the Device Emulator [18]

The VS also enables building and deploying .NET assemblies to the device and allows debugging the code running on actual hardware. It allows application download over a Serial (USART or USB) or over a network. The VS also uses the Serial or network to provide source level debugging capabilities. The MF includes a RPC style subsystem that enables debugging (with breakpoints and variable inspection at runtime).
Chapter 5: The Porting

This chapter describes the porting of the .NET Micro Framework to the eMote. Sections in this chapter talk about the steps that are needed to port the MF on a new hardware, the porting kit environment setup, tool chains and other utilities that are need to build and deploy the firmware on the hardware, development and debugging techniques of certain HAL drivers.

5.1. The Environment

In order to start the porting a development environment needs to be set up. This environment has all the software and necessary tools to perform the port. The following tools are necessary for writing and building code for the porting kit,

- The Micro Framework Porting Kit (v4.0)
- Visual C# 2008
- .NET Micro Framework SDK
5.2. The Tools

These below section describes certain third party tools that are necessary to compile, link, debug and test the modules that are designed and developed as part of the MF port to eMote.

5.2.1 The Tool chain

The MF uses an open source tool chain provided by codesourcery. This package contains the,

- GNU Binary Utilities (Binutils)
- GNU C Compiler (GCC)
- GNU C++ Compiler (G++)
- GNU C++ Runtime Library (Libstdc++)
- GNU Debugger (GDB)
- Newlib C Library

The Binutils contains various binaries of various important tools such as the GNU linker, loader, assembler, disassembler etc. All these tools are very useful programming tools and provide facilities to manipulate the object file [19]. In MF for purpose of building the project we use a tool called the MSBuild, this tool comes as a part of the MF SDK installation. The build and the linking sequence is transparent to the user. The MSBuild utility takes as input the path to the tool chain and internally uses the compiler, linker and
the assembler to produce the binary object file. The system uses an XML file called a targets file which contains all the compiling options along with the appropriate build flags. This file initially contained only the flags to build for ARM and Thumb instruction set. The flags were modified for the compiler to build for the Thumb2 instruction set. The build also internally uses a tool called the Metadataprocessor, the purpose of this tool is to save space on the device by collating all the managed assembly files (.pe) to a single database file (.dat).

5.2.2 Debugging Tools

Various debugging techniques were employed as part of the development of the HAL in the MF. The most commonly used debugging methodology was using the Light Emitting Diodes (LED) on the development board. For this purpose a LED driver was implemented. This driver had APIs to turn on, turn off and toggle different LEDs on the board. These APIs could be called from anywhere to check of the execution was reaching that point in the code.

Another useful method for debugging was using the print statements. The standard print statement in the C language was remapped to print on the serial port of the development board. The serial port would then use the Universal Synchronous and Asynchronous Receiver and Transmitter (USART) driver to print. The USART driver would have APIs such as send() which would send information packets from the micro-
controller on the development board to the host. The messages can then be printed on the hyper terminal and the results can be observed.

Apart for these techniques the GNU Debugger (GDB) was comprehensively used to debug and step though the code, set break points during execution of the code, examining the value of the registers and runtime variables etc. The GDB for an embedded platform is used along with a tool known as the open On Chip Debugger (OCD) and a Joint Test Action Group (JTAG). The open OCD is a in-system programming and boundary-scan testing for embedded devices [20]. It works along with a debug adapter (such as JTAG) and uses a transport protocol which involves different electrical signals (and a messaging protocol on top of that signaling to send and receive) [20]. The board contains Test Access Port (TAP) and JTAG enables communication with these ports. Thus, the open OCD is able to use the JTAG adapter and is able to debug the instructions running on the board. The figure below shows an Open OCD session in progress.

Fig 12: The OpenOCD in session
This figure shows a GDB session working over the Open OCD session. This session shown is loading code on the board using the JTAG adapter and the open OCD.

### 5.2.3 Testing Tools

There are two main tools that were used to verify the functionality and test the correctness of the tools were the Oscilloscope and the Logic Analyzer. Oscilloscope and the analyzer would be hooked on to the GPIO lines coming out on the development
board and the toggling rate can be measured from the wave forms being generated by these devices.

Fig 14: The ANT16

The figure depicts the start up screen of the logic analyzer. Once logic probes are connected to the board the channel would show the graphs for the toggle duration of the particular GPIO line.
5.3 Structure of the Porting Kit

The porting kit is broadly divided into HAL, PAL and the CLR. Each of these components comes with a code base that is provided by Microsoft. This code base is made available to the user as part of the Porting Kit installation. Once you have the porting kit installed on your machine a folder with name, “MicroFrameworkPK_v4_0” is created. This folder contains a directory structure which is the MF source tree. The following is a brief explanation of folder that are important for the porting of the MF to a new hardware,

- **Application** – This folder contains code for boot vector tables and entry points for the hardware.

- **BuildOutput** – This folder contains all the files that are created as part of the porting kit build. It contains the binaries for TinyCLR, disassembly files, database files, PE files, various managed dlls etc.

- **CLR** – This folder contains the source for the Common Language Runtime, these files are used to interface the HAL and PAL with the CLR. There are sub folders within this such as the **Core** which provides runtime memory configuration, the **Debugger**, which provides debugging facilities, **Diagnostics**, for diagnostic messages, **Graphics**, for graphics support, **Include**, which contains files needed to link HAL and PAL to the CLR, **Libraries**, which contains interoperability libraries, **Messaging** provides debug messaging support, **StartupLib** contains the bootstrap support and **Tools** contains code for various run time tools.
- **Crypto** – This contains the source for all the cryptographic services provided by the MF.

- **Device Code** – The Device code contains the source files for the HAL and the PAL. The HAL folder contains the code for all the hardware platforms on which the MF has been ported. The PAL folder contains the source that is provided by Microsoft. These are interface implementation for the services that are required by the CLR from the lower layers in order to operate. The PAL can also be modified and the implementation of the interfaces can be changed as long as the interface signature remains intact. This folder has various other sub folders such as the **Core**, this contains the architecture specific source for the microprocessor, the **Drivers**, contains the source code for various peripheral drivers, it also contains source code for various software services, **Include**, contains the header files needed by these drivers, **Initialization**, comprises of the master boot strapping code, **PAL**, contains source code for the platform layer components and the **Targets**, contains source code for drivers for specific hardware platform.

- **Framework** – Contains source code specific to the object model of the .NET Micro Framework.

- **Product** – This contains the user application code that is hardware independent.

- **Sample Key** – Comprises of a sample cryptographic key.

- **Solutions** – This folder contains various vendor specific sub folders. They hold the source for the HAL and PAL implementations that are specific to the individual platforms. This folder contains the TinyBooher and TinyCLR versions
from each of the vendors in their respective folders. This also contains build files for each of these platforms. The MF build is initialized for a particular platform from here.

- **Support** – This contains source for specific support features such as CRC support, wire protocol for debugging etc.

- **Test** – This folder contains all the managed code tests. These tests can be built, deployed or run from visual studio or they can be built and loaded manually.

- **Tools** – This folder comprises of build and test tools which are useful during porting

- **USB_Drivers** – This folder contains the reference USB driver. While doing the port of the USB driver this folder is that place where the host side windows driver needs to be built.

Most of the development starts off by creating a hardware specific folder in the Solutions directory. Most of the development work takes place in Device Code. Rarely (for debugging purposes) the CLR code needs to be accessed and modified.

### 5.4. A step-by-step walk through porting

This section describes the steps that were taken to perform a custom port of the MF to the eMote.

Before beginning with creation of the code base and building process we need to verify that the build system is set up and working. As it was mentioned earlier the MF
The porting kit uses MSBuild (which is installed as part of the MF SDK) to build the project. This works along with various compilers such as RVDS, GCC etc. For this port we have used GCC as the compiler. We use the setenv_gcc.cmd script provided to us as part of the MF to set up the necessary environment variables for using a GCC compiler. Once this is set up we can take the following steps to build a project,

- Start a command window
- cd into the folder for which you need to invoke the build for (it is usually located at, `%SPOCLIENT%\Solutions\"YourHardware\")
- Issue the msbuild dotNetMF.proj command. There are various switches for this command such as the flavor of the build (debug or release), media type (Flash or RAM). These options are available at msbuild dotNetMF.proj /t:help

The porting kit comes with a tool called the, “Solution Wizard”, this tool generates the code base for porting the MF to your hardware platform. This tool works on the set of inputs that are provided by the user. This input takes in the processor type, name of the code base and all the features that are supported by the hardware (peripherals etc). Based on these inputs the solution wizard generates the complete code base for the specified hardware. This code base contains stubbed code which contains all the interfaces as required by the higher layers and later on the functionality can be filled into these stubs. The solution wizard also generates all the necessary build files that are used by msbuild in order to build the project. As the functionality is implemented in the stubbed code base the references should be made to point to the version of the code
which has an implementation and compiled and built along with the framework. The solution wizard also has option of choosing an existing code base for your port.

The next step in the porting is defining the configuration information for the hardware platform. As a part of this we need to specify the memory map of the hardware which will contain addresses of RAM and Flash of the device.

The memory map for a port to an existing hardware can remain unchanged where as to perform a port to the new hardware the XML files that specify the memory layout need to be modified. The memory map defines the RAM and the ROM sections of the flash. The organization of the RAM defines the stack and the heap regions of memory. Also, the layout of the RAM optimizes code execution (by holding some runtime code), data access and power consumption of the device. The ROM region of the memory hosts the runtime code as well as the managed assemblies. The regions on the flash memory are divided up into sections. These sections are based on a configuration table that is specified as part of the flash (block storage) driver. These sections are mostly divided into sections for code, application assemblies (managed code), application data and deployment regions. The CLR’s type system loads these assemblies at runtime from code and deployment regions.

The memory map of the CLR image is created by the linker in the build system. For laying out the memory correctly the linker needs to know the grouping of output sections and regions. This grouping is specified to the linker by scatter files. The scatter file provides information about the sections of the code (load regions and execution
regions) and it also provides placement information by describing the addresses of regions located in the memory map.

In MF the scatter files are defined in an XML format where the regions of the code are specified along with the addresses of the RAM and the ROM. This XML file is then fed to the linker as a part of the build process and the CLR image is laid out based on the configuration specified in these XML files.

The configuration information also specifies the Serial and USB information and the system clock specifications for the device. This information is then used by the system to specify the messaging and debugging ports. The clock specifications are used to configure and set the system time and timer modules.

The next step in the porting of the MF is to go through the FirstyEntry.s this assembly file contains the entry point for the linker and the system boot straps start from this file. After the entry point this file branches the execution to the Boot Strap code. The boot strap code starts up the system by initializing the system clocks, the PLL and other CPU blocks and calls the Boot Entry function. This function initializes all the peripherals in the device by calling the initialize method which are implemented in the device driver for that particular hardware. The second important file in the initialization process is the VectorTrampolines.s. This file along with the scatter files sets up the reset vector. The device will not get initialize and start executing the application code unless the reset vector is set at the correct memory location. A useful tool to verify the memory layout use the GNU object disassemble (objdump). This tool dumps out the layout of the firmware
in a human readable form. For eMote the flash region starts at 0x8000000 and it is necessary that the reset vector start from that memory location.

After these initial assembly files are implemented correctly we need to build this modified version of the MF. This version still builds and links the stubbed source file and thus will build correctly. On completion of the build process we need to verify that FirstEntry.obj and the reset vectors appear at the correct addresses of the memory map. This is important because unless we have the FirstEntry.obj and reset vector correctly we cannot jump and execute the application program.

The next step in the porting process is to implement and build the basic drivers on which the port depends. The following section describes the various drivers that need to be implemented as part of the porting,

**5.4.1. HAL Drivers**

There are various HAL drivers that are provided within the MF. These modules are stubbed out as part of the code base created by the solution wizard. For running any application a driver implementation has to be provided. The following drivers form the core set of drivers for the MF port, these drivers should be one of the first to be implemented. Having an implementation of these drivers would provide the ability to run the CLR on the hardware (also, we only need to implement a driver in software if it is available in the underlying hardware),
- **General Purpose Input Output (GPIO) Driver** – The CLR supports communication over the GPIO lines. This driver provides all the methods to initialize, access, modify and uninitialized the GPIO lines. There are various uses of this driver such as user input, external device communication, debugging etc.

- **CPU Interrupt Controller** – This driver provides APIs to control the CPU interrupt requests. There are APIs to activate and deactivate interrupt lines associated with hardware peripherals. It also provides mechanism to create and register call back functions. This function would then be executed from the interrupt handler for a interrupt from a particular peripheral.

- **Time** – This driver comprises of the system time module. This provides the CLR with a reference of system time. This driver provides APIs to initialize the system time, conversion of system time to ticks and ticks to system time, function to get the value of current system time and ability to set and compare a time value with the system time. In the event when a compare value matches the system time value it generates an interrupt and notifies the CLR. The CLR depends on this module for thread scheduling, executing asynchronous function calls, executing completions and continuations etc. The timer driver implementation uses a hardware timer underneath. The access to this hardware timer a driver is provided which has the APIs to control the hardware timer and its interrupts.

- **Direct Memory Access (DMA)** – The DMA is supported by MF if its available in the hardware. The DMA driver provides APIs for initializing and uninitialized the DMA peripheral. It has APIs such as DMA implementation of the memcopy
which copies the specified number of bytes from source to destination. The DMA
driver memcpy API provides ability to copy asynchronously based on the value
of a flag.

- **Power** – The power driver implements the power management APIs. The CPU
  power can be initialized from this function and it provides APIs for power usage
  management such as setting the power level to a specific value, resetting the CPU,
  and putting it to sleep.

- **Clock Management** – This driver provides APIs for managing the system clock.
  It provides APIs to divide the system clock by a given factor and enabling and
disabling of peripheral clock.

- **Watchdog Timer** – The CLR uses a watch dog timer to automatically reset the
  system in case it becomes unresponsive. The watch dog timer driver provides
  APIs for enabling and disabling the device, setting and getting the value of the
  watch dog timer, functions to reset the timer and the CPU etc.

- **Block Storage** – The block storage driver specifies the memory layout
  configuration of the device to the CPU. The CLR uses the APIs provided by this
  driver to indentify regions of memory in the flash.

- **Global Lock** – This global lock provides synchronization primitives for the CLR.
  This uses a smart pointer implementation of a C++ object which when placed on
  the stack disables the interrupts for the processor. The object performs locking of
  interrupts in its constructor and unlocking in the destructor.
• **Flash** – This driver provides APIs to read and write data from and to the flash memory. This provides a persistent storage mechanism in eMote.

The following section explains the core communication drivers that are support within the MF,

• **I2C** – The I2C communication is supported by the CLR in MF. The I2C is a bus with a clock and a data line. The bus has two roles for the components which are connected to it. The master issues the clock and addresses slaves and the slave receives the clock and address. The driver provides the send and receive functions along with the structures that are used to hold the data and control signals.

• **Serial Peripheral Interface (SPI)** – The SPI driver implements the SPI communication protocol. The SPI is a master-slave bidirectional protocol which has four lines, the master in slave out, master out slave in, the clock and the chip select line. The driver provides APIs to access the device and send and receive information. It also provides the structure for holding data and control signals for the communication

• **Universal Synchronous and Asynchronous Receiver and Transmitter (USART)** – The USART driver is the main debugging interface for porting the MF to a new hardware and it is one of the first drivers to be implemented. The CLR works at the 115200 symbols per second rate (USART baud rate). This driver comprises of APIs to initialize and uninitialize the USART and to send and receive data over it. The USART can be easily connected to the host and messages can be printed on the Hyper Terminal for debugging. The USART can
be integrated to visual studio and MF deploy for the purpose of deploying and running code on the hardware.

- **Universal Serial Bus (USB)** – The USB driver is also supported by the MF for communication. The USB can be integrated along with the visual studio and MF deploy to deploy and run applications on the device. The USB driver implemented at the HAL layer coordinates with the USB driver implementation at the PAL layer. The HAL driver provides services such as initializing and uninitializing the USB core in the hardware, it deals with the interrupt handling and the PAL functions that are registered with the HAL are called for servicing the interrupts such as device enumeration. The other APIs at the HAL layer include sending out a data buffer issued by the PAL over the USB data lines and receiving data from the hardware and placing it in a buffer provided by the PAL layer.

These are only the list of drivers that are core to performing a basic port of the MF to a new hardware. Most of these drivers are supported by the eMote. There are various other HAL drivers such as the cache management, MMU management etc that are not supported by the eMote but are available within the frame work. There are also various system level functions such as private_malloc and private_free to access the systems heap, Application Entry Point which defines the entry point for the application. The initialization of the CLR is called from the this function, DelayLoop function, which provides a delay of the specified number of instructions, SystemState_Query, SystemState_Set and SystemState_Clear function, which query, set and clear the system
state respectively. It also contains functions that retrieve the state of the stack and handle stack overflow and various functions which checks the bound of a given number to be in a data type range.

5.4.2. HAL Extensions (Interops)

The MF as described provides the ability to expose hardware functionality to the CLR and managed applications through Interops. The MF port to the eMote exposes the following drivers by implementing the HAL drivers as a HAL extension and using the interops capabilities to expose them at the managed code,

- **Analog to Digital Convertor (ADC)** – This ADC is used by the eMote for the purposes of interfacing it with external modules such as the radar. The eMote collects the samples from the ADC interface and performs calculations on those values. The ADC driver provides interfaces for enabling and disabling ADC and reading the values over the ADC lines.

- **Radio** – The Radio driver is implemented with the eMote HAL. The Radio driver provides 802.15 communication capabilities in the eMote. The Radio driver is implemented on top of the SPI driver. The SPI bus is used for sending and receiving data from and to the eMote. The Radio driver provides functions for enabling and disabling the radio and sending and receiving of information packets.
• **Security** – The security in the eMote is implemented as part of the HAL extension and the standard security APIs of the HAL are left as stubs. This is because the eMote uses an AES encryption algorithm implemented in native language and linked with the MF HAL. This encryption APIs are exposed as interop calls to the C# user under a use defined security namespace. This HAL module provides APIs to encrypt and decrypt data using the AES algorithm.
Chapter 6: Reflections Conclusion and Future work

This chapter discusses the Reflections, Conclusion and the Future work of this thesis.

6.1. Reflections

The time it took to port the Micro Framework to the new hardware could have been improved if I had some prior domain exposure. Using of the right set of tools for the purpose of debugging was the most important exercise in the development cycle of a module. The amount of time that is required to design, develop and release a module can be improved if the right set of development and debugging tools are used. The amount of time for testing of a module can also be significantly shortened and made reliable if it can be automated. The testing in this space requires very frequent human intervention. Even though the test cases are to be written by the developer certain steps such as loading, running, execution of the tests can be automated. The automated testing module can send a report of the test execution to the developer there by saving developer time and make the testing process more robust and reliable.
6.2. Conclusions

As a part of this work we have performed a basic port of the .NET Micro Framework to the STM32 Cortex M3 platform. The Micro Framework has a lot of hardware support at the hardware abstraction layer which makes it adequate to be used as a platform for the Wireless Sensor Networking devices. For the certain features it does not have as a part of the platform, it provides mechanisms such as Interoperation for easy extensibility. The low resource consumption and memory requirements, the size of the firmware and the nature of operation of the Common Language Runtime make the Micro Framework a useful platform for the wireless sensing platform.

Also as a part of this thesis we have described steps where we can use open source tool chains for the purpose of porting the Micro Framework to a new hardware. We have described the debugging techniques that were employed during development stages. These techniques will useful for the people who are not familiar Micro Framework and do not possess domain knowledge in the porting space.

We have also established that as a single developer starting very little knowledge of the embedded domain and the Micro Framework it is not possible to perform the port in the specified time frame of six weeks. This can be attributed to the fact that as Micro Framework is considerably a new open source platform the help in terms of documentation and online resources is limited.
6.3. Future Work

This work has performed only a basic port of the Micro Framework to the STM32. There are various other HAL modules such as the DMA, I2C etc which are still be implemented and tested. The PAL can be extended by providing a file system service built over the Flash driver, a Medium Access Control layer can be implemented as part of the PAL over the Radio driver to further improve the capability of communicating over the Radio. The CLR in the current system performs an interpretation of the managed code at run time. The hardware for this device has enough power and memory to support Just-In-Time compilation. The CLR can be extended to provide JIT services to improve the speed of execution of the managed code.
References


2. Egs: A Cortex-M3 based mote platform, by Jeonggil Ko, Qiang Wang, Thomas Schmid, Wanja Hofer, Prabal Dutta, Andreas Terzis


7. Introducing the .NET Micro Framework, Product Positioning and Technology Whitepaper

8. GHI Electronics. FEZ Panda II. Available at: http://www.ghielectronics.com/catalog/product/256


10. Security in embedded systems: Design challenges, by Srivaths Ravi and Anand Raghunathan, NEC Laboratories America; Paul Kocher, Cryptography Research; Sunil Hattangady, Texas Instruments Inc.


16. .NET Micro Framework Porting Kit help document


18. Introducing the .NET Micro Framework, Product Positioning and Technology Whitepaper
