Beginners Guide to
C#
and
.NET Micro Framework

August 11th 2010
Rev 1.04

Table of Contents

---

1. Change Log

Version 1.04

- Corrected typos

- Changed VS2008 to VS2010

- Added a note about IO40 board in expanding IO section

- Updated watchdog section, disable is not allowed

Version 1.03

- Corrected typos

- Revised all the font type in sample code boxes

- Revised a paragraph 16.1 Byte

- Rewrote "Missing Topics"

- Added Watchdog section

- Added Low Power Section

- Added important note to Garbage Collector

- Added a new Output messages subsection to Garbage Collector

- Added "other devices" in GHI offers

- Changed graphics section to point out the new SPI displays on www.TinyCLR.com

- The index now has two rows

Version 1.02

- Corrected typos

- Updated to use VS2010 instead of VS2008. Updated 4.0 to 4.1 and put new link to 4.1 download

- Added a note about using glitch filters with interrupt-capable pins

- Rewrote the Real Time Clock section

- Added new USB Client section

2. About the Book

2.1. Intended Audience

This book is for beginners wanting to learn .NET Micro Framework. No prior knowledge is necessary. The book covers .NET Micro Framework,Visual C# and even covers C#!

If you're a programmer, a hobbyist or an engineer, you will find a good deal of info in this book. This book makes no assumption about what you, the reader, knows so everything is explained extensively.

I have used my personal free time (if that is ever available!) to make this book. Expect a lot of typos and grammar errors, but please report them on the forum so I can enhance this book.

2.2. Translating the Book

This a book given to the community in an efforts to make NETMF easier for all users. If you think you can translate the book to other languages then we would love see your contribution. The original sources of this book and rules are on this page

http://www.microframeworkprojects.com/index.php?title=Free_eBook

3. Introduction

Have you ever thought of some great idea for a product but you couldn't bring it to life because technology wasn't on your side? Or maybe thought, "there's got to be an easier way!" Maybe you are a programmer that wanted to make a security system but then thought using PCs are too expensive to run a simple system? The answer is Microsoft's .NET Micro Framework!

Here is a scenario, you want to make a pocket-GPS-data-logger that saves positions, acceleration, and temperatures on a memory card and displays them on a small display. GPS devices can send position data over a serial port so you can easily write some code on a PC to read the GPS data and save it on a file. But a PC wouldn't fit in your pocket! Another problem is how would you measure temperature and acceleration on a PC? If you make this project using classic microcontrollers, like AVR, or PIC micro, all this can be done but then you need a compiler for the micro you choose (probably not free), a week to learn the processor, a week to write serial driver, a month or more to figure out the FAT file system and more time for memory cards...etc. Basically, it can be done in few weeks of work.

Another option is utilizing simpler methods (BASIC STAMP, PICAXE, Arduino, etc). All these products simplify a design but each one has its limitation. Only a couple of them have debugging capabilities. Finally, these devices are not normally suited for mass production.

3.1. Advantages

If you are using .NET Micro Framework then there are many advantages:

1. It runs on Microsoft's Visual C# Express, free and and high-end IDE.
2. .NET Micro Framework is open-source and free.
3. Your code will run on all these devices with almost on changes.
4. Full debugging capabilities. (Breakpoints, stepping in code, variables...etc.)
5. Has been tested in many commercial products so quality is assured.
6. Includes many bus drivers.(SPI, UART , I2C...etc.)
7. No need to use processors datasheets because of the standard framework.
8. If you are already a PC C# programmer then you are already an embedded system developer with NETMF!

Throughout this document, I may refer to .NET Micro Framework as NETMF.

4. Porting

There are two sides of working with NETMF, porting it and using it. For example, writing a JAVA game on a cell phone is much easier than placing the JAVA virtual machine (JVM) on a phone. The phone manufacture did all the work of porting JAVA to their phone and game programmers can use it with less effort. NETMF works the same way, porting is not easy but using it is very easy.

NETMF can be split in two major components, the core and HAL (Hardware Access Layer). The core libraries are made so they are hardware independent. Usually no modifications are needed on the core libraries. Developers need to make their own HAL to run NETMF on a device.

4.1. GHI Standard Offers

We work very close with our customers to make sure everything is flowing as needed. GHI offers many exclusive features that come standard and free with our products. (USB host, database, PPP, RLP, Wi-Fi, one wire, etc...)

Dedicated and free support is available through email, phone and forum.

5. Selecting a Device

GHI Electronics offers an extensive range from the very basic to the very advance.

- ChipworkX

- EMX

- USBizi

- FEZ Family

  - FEZ Domino and FEZ Mini

  - FEZ Rhino

  - FEZ Cobra

...and many more.

5.1. ChipworkX Module

If processing power and customization is needed then this is the right choice. ChipworkX runs a 200Mhz ARM processor with 64MB 32-bit SDRAM and 8MB for user applications. It also contains 256MB internal flash for file system storage. It includes all NETMF major features and adds GHI exclusive features like WiFi and USB host support.
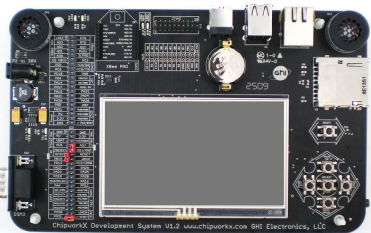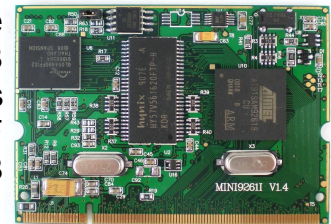
ChipworkX also adds SQLite database support and allows users to load their own native code (C/assembly) on the device using RLP (Runtime Loadable Procedures). RLP allows for advance processor intensive and real-time applications.

5.2. EMX Module

This small module includes all NETMF major features and adds many GHI exclusive features. On the software side: File system, TCP/IP, SSL, Graphics, debugging and more NETMF features are included. GHI also adds: WiFi, PPP, USB host, USB device builder, CAN, Analog in/out, PWM and more. As for the hardware: It is 72Mhz ARM processor with 8MB SDRAM and 4.5MB FLASH.
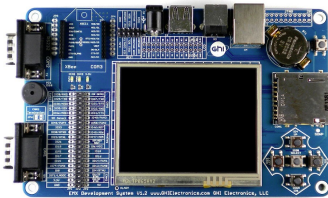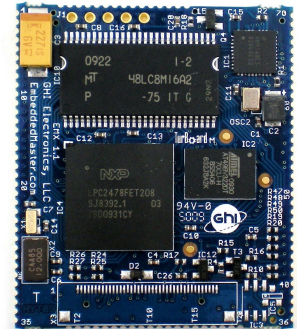
The processor on EMX contains Ethernet MAC built right in with DMA transfers, which gives it a large boost when compared with classical SPI-based Ethernet chipset used by others.

5.3. USBizi Chipset

USBizi is the smallest and only single-chip running NETMF in the world. The software running on it is a scaled down version of Embedded Master. It includes all features except networking (TCP/IP and PPP) and native graphics. Even though these features are missing, USBizi can be connected to a network using TCP/IP chipsets like WIZnet and can run simple displays. There are example projects already provided showing how USBizi can be networked and can display graphics.
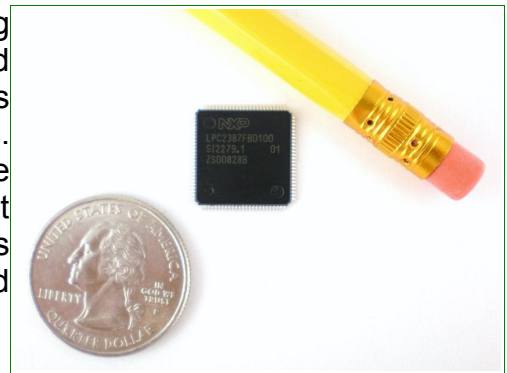
5.4. FEZ Family

FEZ Domino and FEZ Mini

FEZ Domino and FEZ Mini are very small (open source) boards targeted for beginners. They are based on the USBizi chipset. FEZ offers many peripherals, such as USB host and SD interface, not available with many hobbyist-targeted boards. Even though FEZ is targeted for beginners, it's also a low-cost starting point for professionals wanting to explore NETMF.
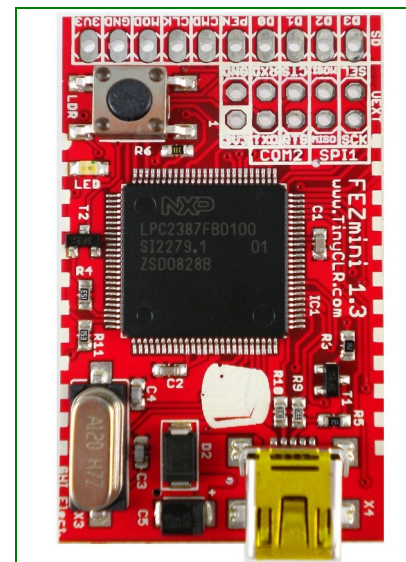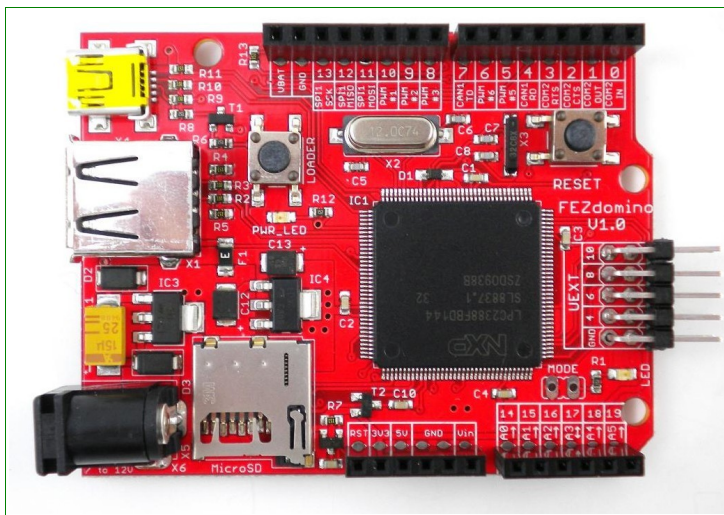
FEZ stands for "Freakin' Easy!"

FEZ offers many features not found in Arduino, BASIC STAMP and others:

- Based on Microsoft's .NET Micro Framework.

- Runs on a 72Mhz NXP ARM processors.

- Supports run time debugging (breakpoints, variable inspection, stepping, etc.)

- Uses Visual Studio 2008 C# Express Edition for development.

- Advanced capabilities like FAT, USB device and USB host.

- Easily upgrades to hardware such as Embedded Master.

- Open source hardware design files.

- Use existing shields and holder boards.

- Based on the USBizi chipset (ideal for commercial use).

- FEZ Mini pin-out compatible with BS2.

- FEZ Domino pin-out compatible with Arduino.

When using FEZ, the possibilities are endless...
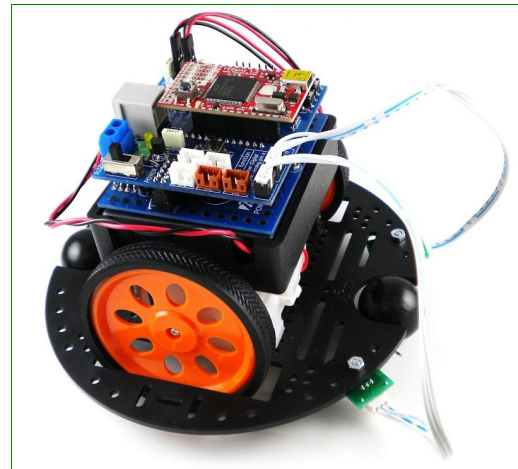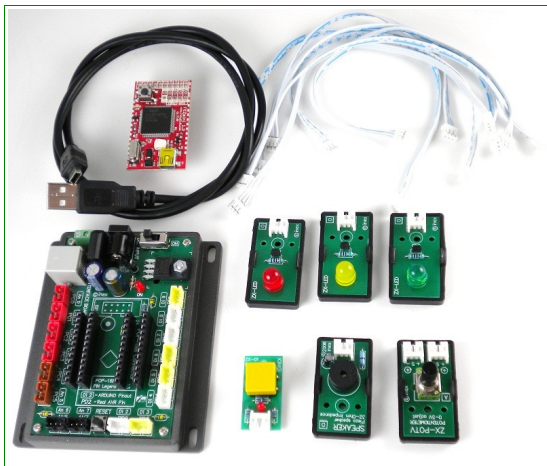
There is a numerous amount of sensors that are ready to plug directly into the kits. From LEDs and buttons to reflection and temperature sensors.

These book examples are made for FEZ devices. In general, the examples are still for .NET Micro Framework, so modifying it to run on any NETMF system should be an easy task.

FEZ Cobra

FEZ Cobra is an open source circuit board based on EMX module. With multiple optional displays, optional enclosure, native graphics support, native Ethernet support with SSL, and megabytes of memory, FEZ Cobra is ideal for those high-end projects. Not to forget that this is still FEZ so it is Freakin' Easy to use!
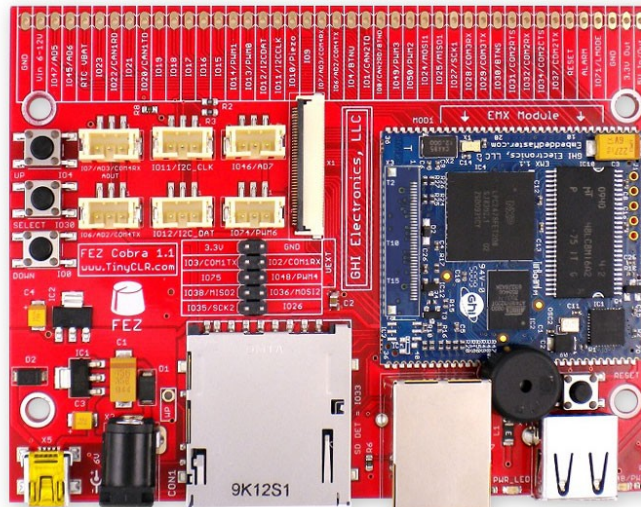
An LCD expansion board is available to hold either 3.5" (320x240) or 4.3" (480x272) displays.
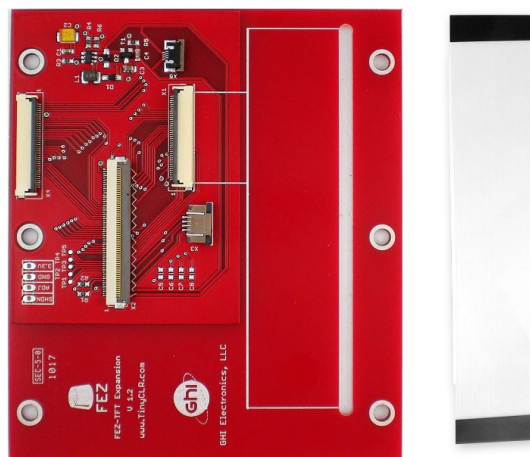
Other Devices

GHI strives to deliver the latest technologies dynamically to its customer needs. There is always something new at GHI so please visit www.GHIElectronics.com and www.TinyCLR.com for the latest offers.

6. Getting Started

Important note: **If you have just received your hardware or you are not sure what firmware is loaded on it, you MUST update the firmware. The documentation (manual or tutorial) of your device shows you how to update the firmware. This is a required step. Also, make sure you read the "firmware/assembly matching" section in this book.**

6.1. System Setup

Before we try anything, we want to make sure the PC is setup with needed software. First download and install **Visual C# express 2010** (VS2008 or older will not work)

http://www.microsoft.com/express/vcsharp/

Now, download and install .NET Micro Framework 4.1 SDK (not the porting kit).

http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=cff5a7b7-c21c-4127-ac65-5516384da3a0

If link above didn't work, search for ".NET Micro Framework 4.1 SDK"

Finally, install the GHI NETMF SDK. You can get the SDK from:

http://www.tinyclr.com/dl/

6.2. The Emulator

NETMF includes an emulator that allows the running of NETMF applications right on the PC. For our first project, we will use the emulator to run a very simple application.

Create a Project

Open Visual C# Express and, from the menu, select **file -> New Project.** The wizard now should have "Micro Framework" option in the left menu. Click on it, and from the templates, select "Console Application".
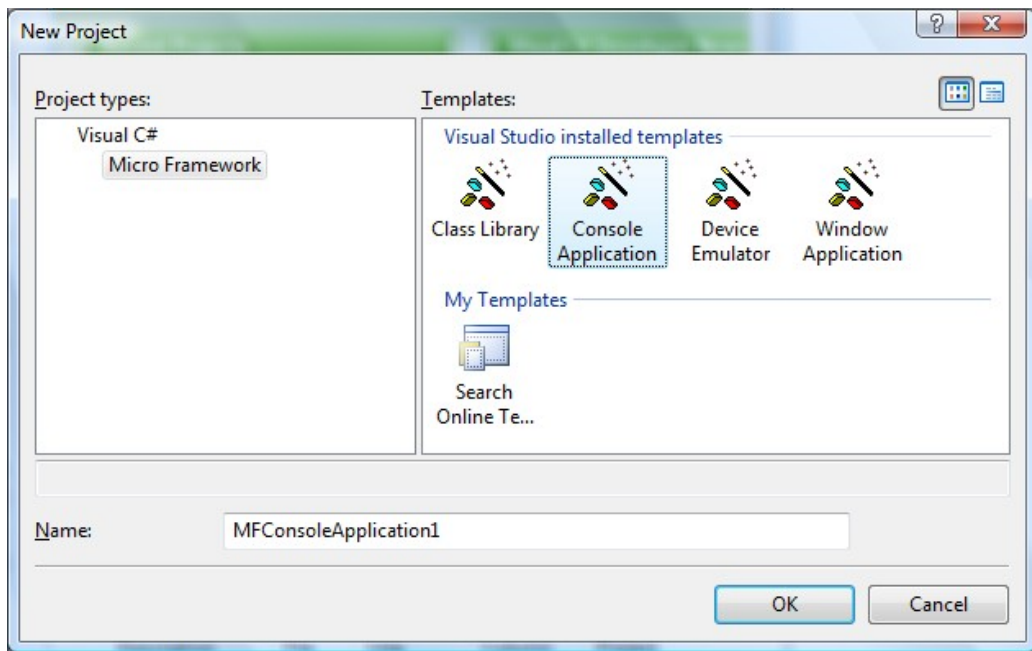
Click the "OK" button and you will have a new project that is ready to run. The project has only one C# file, called Program.cs, which contains very few lines of code. The file is shown in "Solution Explorer" window. If this window is not showing then you can open it by clicking "View->Solution Explorer" from the menu.

```csharp
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(
                Resources.GetString(Resources.StringResources.String1));
        }

    }
}
```

For simplicity change the code as it look like the listing below

```
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print("Amazing!");
        }

    }
}
```

Selecting Transport

Don't worry if you do not understand the code. I will explain it later. For now, we want to run it on the emulator. Let us make sure you have everything setup properly. Click on "Project->Properties" from the menu. In the new showing window, we want to make sure we select the emulator. On the left side tabs, select ".NET Micro Framework" and make sure the window looks like the image below.

Transport: Emulator
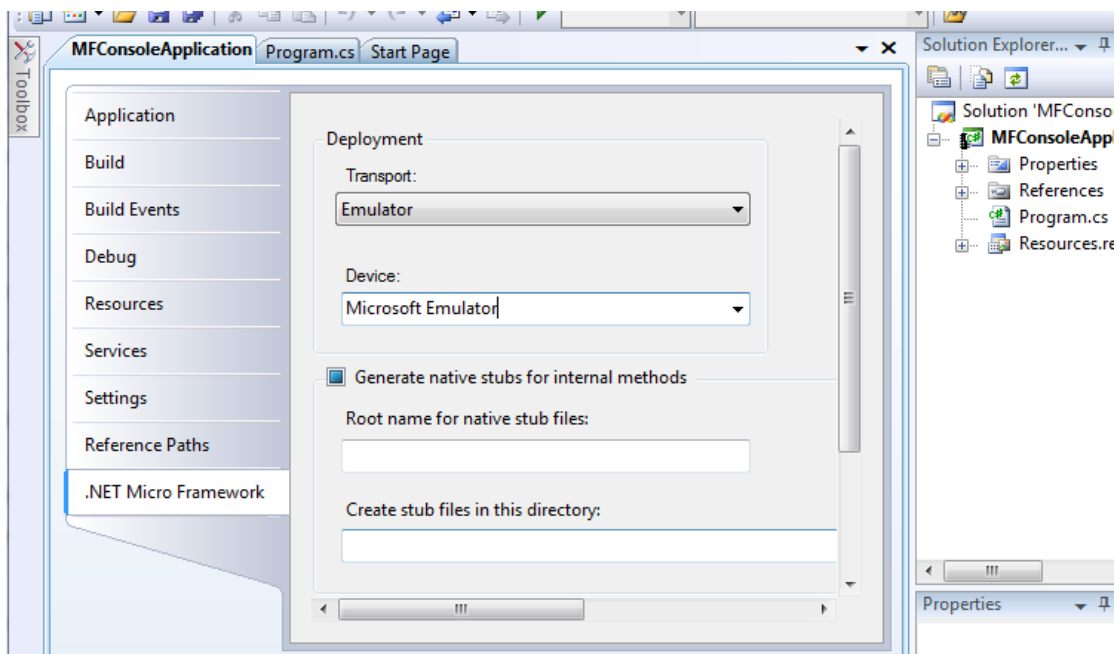
Device: Microsoft Emulator

Make sure the output window is visible, click on "View->Output"

Executing

Finally, we are ready to run our first application. Press F5 key on the computer. This is a very useful shortcut and you will be using it a lot to run your applications. After you press F5, the application will be compiled and loaded on the emulator and in couple seconds everything will stop! That is because our program had finished execution so fast that we didn't see much.

We want to "debug" the code now. Debugging means that you are able to step in the code and see what it is doing. This is one of the greatest values of NETMF.

This time use F11 instead of F5, this will "step" in the application instead of just running it. This will deploy the application on the emulator and stop at the very first line of the code. This is indicated by the yellow arrow.
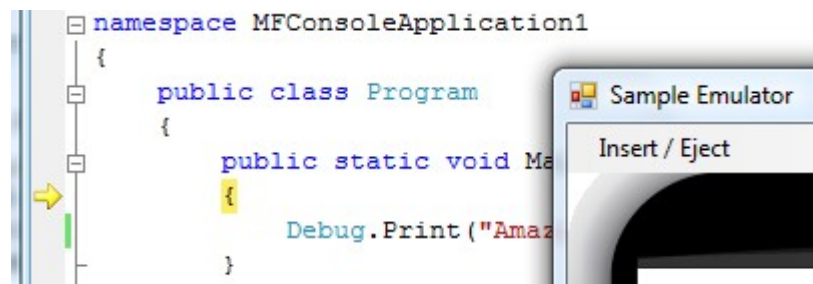
C# applications always start from a method called Main and this is where the arrow had stopped. Press F11 again and the debugger will run the next line of code, which is the line you changed before. You probably have guessed it right, this line will print "Amazing!" to the debug window. The debug window is the output window on Visual C# Express. Make sure Output window is visible like explained earlier and press F11 one more time. Once you step on that line, you will see the word "Amazing!", showing in the output window.

```
public static void Main()
{
        Debug.Print("Amazing!");
}
```

Output   ▼ ┄ ×

Show output from: Debug   ▼ | 🗗 | 🗐 | ↕

```
'Microsoft.SPOT.Emulator.Sample.SampleEmulator.exe
'Microsoft.SPOT.Emulator.Sample.SampleEmulator.exe
'Microsoft.SPOT.Emulator.Sample.SampleEmulator.exe
The thread 0x2 has exited with code 0 (0x0).
Amazing!
```

📇 Call Stack | 🖹 Immediate Window | 🗐 Output | 🔖 Error List
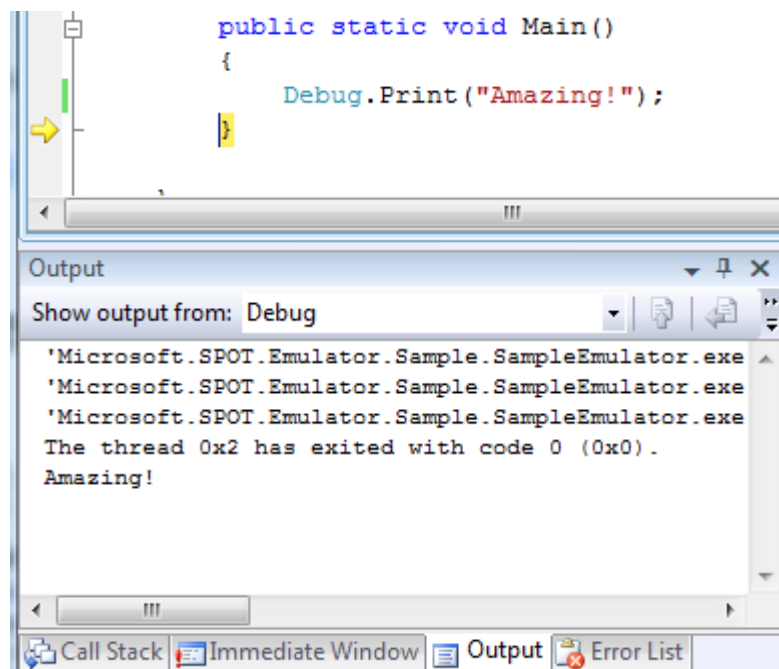
If you now press F11 again, the program will end and the emulator will exit.

Breakpoints

Breakpoints are another useful feature when debugging code. While the application is running, the debugger checks if execution has reached a breakpoint. If so, the execution will pause. Click the bar right to the left of the line that prints "Amazing!" This will show a red dot which is the breakpoint.

```
{
    public static void Main()
    {
        Debug.Print("Amazing!");
    }
```
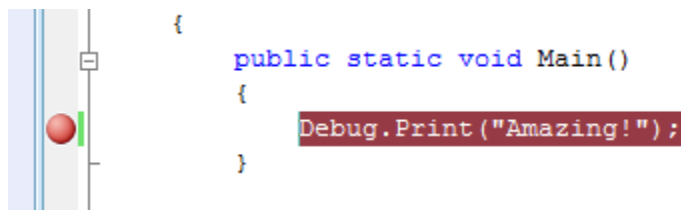
Now press F5 to run the software and when the application reaches the breakpoint the debugger will pause it as showing in the image below

```
    public static void Main()
    {
        Debug.Print("Amazing!");
    }
```
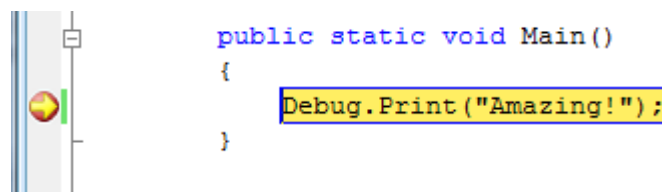
Now, you can step in the code using F11 or continue execution using F5.

6.3. Running on Hardware

Running NETMF applications on hardware is very simple. Instructions can be very slightly different on every hardware. This book uses FEZ for demonstration purposes but any other hardware will work similarly.

MFDeploy can Ping!

Before we use the hardware, let us make sure it is properly connected. The NETMF SDK comes with software from Microsoft called MFDeploy. There are many good uses for MFDeploy but for now we only need it to "ping" the device. Basically, "ping" meas MFDeploy will say "Hi" to the device and then checks if the device will respond with "Hi". This is good to make sure the device connected properly and transport with it has no issues.

Open MFDeploy and connect FEZ using the included USB cable to your PC. If this is the first time you plug in FEZ, Windows will ask for drivers. Supply the driver from the SDK folder and wait till windows is finished.

In the drop-down menu, select USB. You should see USBizi showing in the device list. You will see USBizi because FEZ is based on USBizi chipset. Select USBizi and click the "Ping" button. You should now see back TinyCLR.
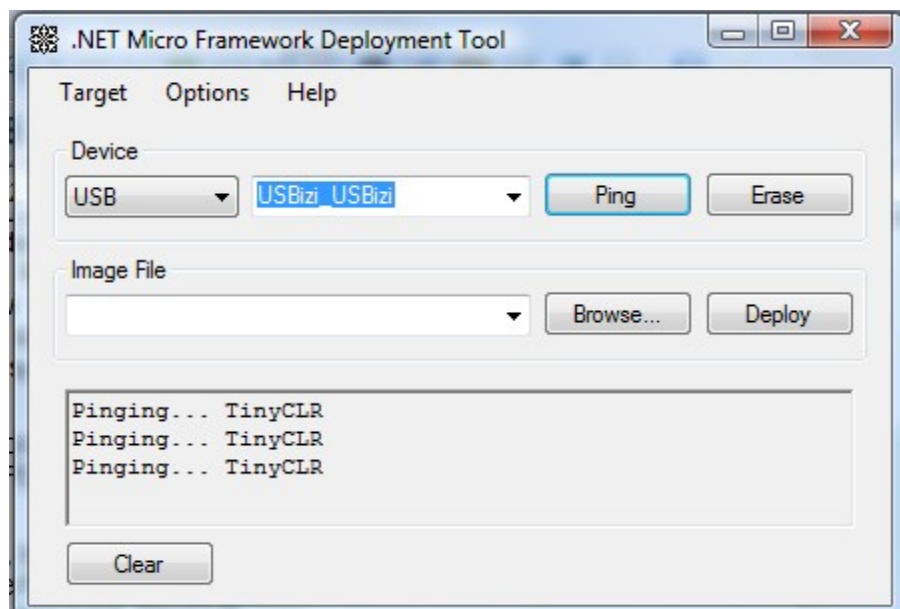
Deploying to Hardware

Now that we checked that the hardware is connected using MFDeploy, we need to go back to Visual C# express. From the project properties, select USB for transport and USBizi for the device. Make sure your setup looks similar to the image below.
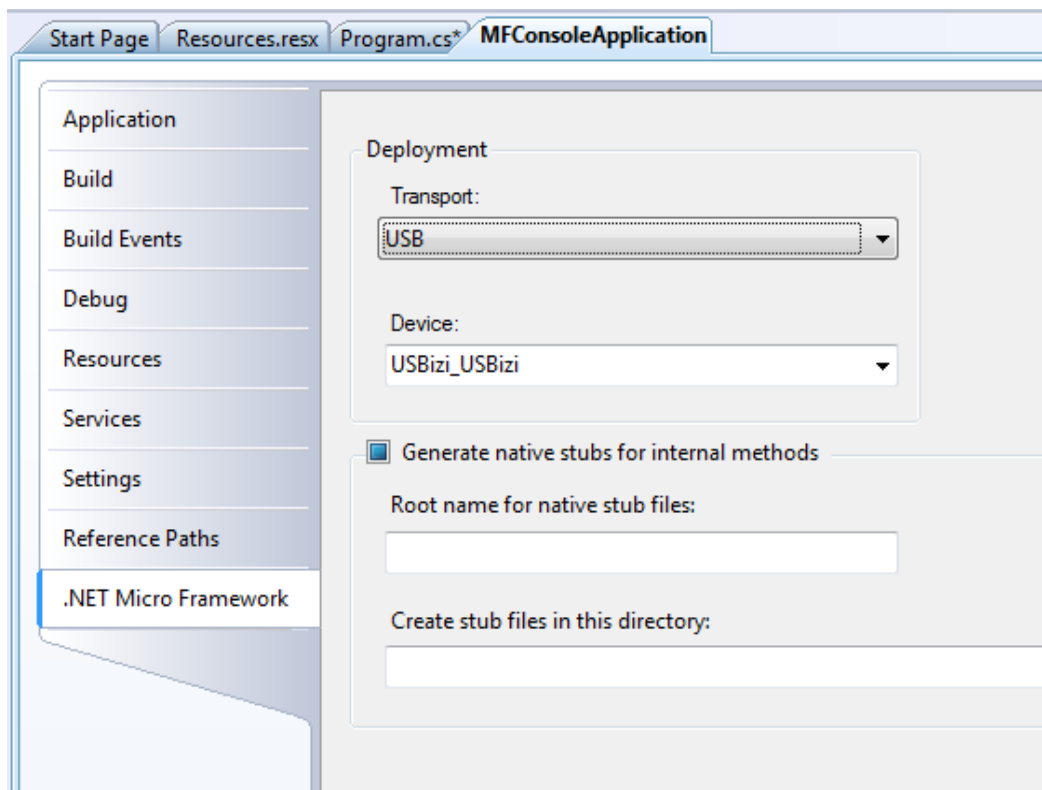
Pressing F5 will now send our simple application to FEZ and it will run right inside the real hardware. Switching from emulator to real hardware is that simple!

Try the steps we did with the emulator, like setting breakpoints and using F11 to step in the code. Note that "Debug.Print" will still forward the debug messages from the hardware to the output window on Visual C# Express.

7. Component Drivers

FEZ components (LEDs, buttons, temp-sensor, relays, servo-driver...etc.) and FEZ shields (Ethernet, LCD, motor-driver...etc.) come with example drivers. Those drivers assume you know nothing about hardware. For example, to blink a LED, you simply command the driver to do so. It doesn't talk about processor pins and how to change the pin state...etc. On the other hand, this book teaches the basics. Use the component drivers to get started and then use this book to understand what the driver is actually doing.

8. C# Level 1

This book is not meant to teach C# but we will cover most of basics to help you get started.

So learning C# is not boring, I will divide it into different levels so we will go on to do more fun things with NETMF then come back to C# when necessary.

8.1. What is .NET?

Microsoft developed .NET Framework to standardize programming. (Note how I am talking about the full .NET Framework and not the **Micro** Framework.) There are a set of libraries that developers can use from many programming languages. The .NET Framework run on PCs and not on smaller devices, because it is a very large framework. Also, the full framework has many things that wouldn't be very useful on smaller devices. This is how .NET Compact Framework was born. The compact framework removed unneeded libraries to shrink down the size of the framework. This smaller version runs on Windows CE and smart phones. The compact framework is smaller than the full framework but it is still too large for mini devices because of its size and because it require an operating system to run.

.NET Micro Framework is the smallest version of those frameworks. It removed more libraries and it became an OS independent. Because of the similarity among these three frameworks, almost same code can now run on PCs and small devices, with little or no modifications.

For example, using the serial port on a PC, WinCE device or FEZ (USBizi) works the same way, when using .NET.

8.2. What is C#?

C and C++ are the most popular programming languages. C# is an updated and modern version of C and C++. It includes everything you would expect in a modern language, like garbage collector and run-time validation. It is also object-oriented which makes programs more portable and easier to debug and port. Although C# puts a lot of rules on programming to shrink down the bug-possibilities, it still offers most of the powerful features C/C++ have.

"Main" is the Starting Point

Like we seen before, programs always start at a method called Main. A method is a little chunk of code that does a certain task. Methods start and finish with open/close curly bracket. In our first program, we only had one line of code in between the curly brackets.

The line was Debug.Print("Amazing!");

You can see how the line ends with a semicolon. All lines must end the same way.

This line calls the Print method that exists in the Debug object. It calls it while passing the string "Amazing!"

Confused? Let's try to clear it out a bit. Let's say you are an object. You also have multiple methods to control you, the object. One method can be "Sit" and another can be "Run". Now what if I want you to "Say" amazing? I will be calling your speak method with the sentence (string) "Amazing!". So the code will look like

You.Say("Amazing!");

Now why do we need the quotes before and after the word Amazing? That is because C# doesn't know if the text you are writing is actually command or it is actually text (strings). You can see how it is colored in red when you add quotes, which makes reading code easier for us, humans.

Comments

What if you want to add comments/notes/warnings in your code? Those comments will help you, and others, understand what the code means. C# completely ignores these comments. There are 2 ways to create comments, line comments and block comments. Comments (Ignored text) are shown in green.

To comment a line, or part of a line, add // before the comment text. The color of the text will change to green indicating that the text is now comment and is ignored by C#.

```csharp
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            // This is a comment
            Debug.Print("Amazing!");//this is a comment too!
        }

    }
}
```

You can also comment a whole block. Start the comment with /* and then end it with */ symbols

```csharp
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            /* This is a comment
             it it still a comment
             the block will end now */
            Debug.Print("Amazing!");
        }

    }
}
```

while-loop

It is time for our first keyword, "while". The while-loop start and end with curly brackets to contain some code. Everything inside will continuously run while a statement is true. For example, I can ask you to keep reading this book "while" you are awake!

So, let's make the program that continuously print "Amazing!" endlessly. This endless loop has no ending so it will always be "true".

```csharp
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            while(true)
            {
                Debug.Print("Amazing!");
            }
        }

    }
```

```
}
```

In the code above, execution will start at "Main" method as usual and then it will go to the next line which is the while-loop. The while-loop is telling the run time to execute the code inside its brackets while the statement is "true". Actually, we do not have a statement there but we have "true" instead which means this loop will always run.

Do not hit F5 to run the program or you will flood the output window with the word "Amazing!". Instead, hit F11 and step in the code to understand how the loop works. Note that this program will never end so you will need to force stop using shift+F5.

Note: You can reach all these debug shortcuts from the menu under Debug.

Variables

Variables are places in memory reserved for your use. The amount of memory reserved for you depends on the type of the variable. I will not cover every single type here but any C# book will explain this in details.

We will be using int variable. This type of variable is used to hold integer numbers.

Simply saying

int MyVar;

will tell the system that you want some memory to be reserved for you. This memory will be referenced to as MyVar. You can give it any name you like as long as the name doesn't contain spaces. Now you can put any integer number in this memory.

MyVar = 1234;

You can also use mathematical operations to calculate numbers

MyVar = 123 + 456;

or you can increment the number by one

MyVar++;

or decrement it by one

MyVar- -;

With all that, can we make a program that prints the word 'Amazing!' three times? Here is the code

```
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
```

```csharp
    public class Program
    {
        public static void Main()
        {
            int MyVar;
            MyVar = 3;
            while(MyVar>0)
            {
                MyVar--;
                Debug.Print("Amazing!");
            }
        }
    }
}
```

Note: how the while-loop statement is not always "true" anymore but it is MyVar>0. This is saying, keep looping as long as MyVar value is more than 0.

In the very first loop MyVar is 3. Inside every loop, we decrement MyVar by one. This will result in the loop running exactly three times and therefore printing "Amazing!" three times.

Let's make things more interesting. I want to print numbers 1 to 10. OK, we know how to make a variable and we know how to increment it but how to print a number on the debug output window? Simply giving MyVar to Debug.Print will give you error and it won't work. This is because Debug.Print will only accept strings, not integers. How do we convert integer variable "ToString"? It is very simple, call MyVar.ToString(). That was easy!

```csharp
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            int MyVar;
            MyVar = 0;
            while(MyVar<10)
            {
                MyVar++;
                Debug.Print(MyVar.ToString());
            }
        }
    }
}
```

Last thing to add is that we want to make the program print

Count: 1

Count: 2

...

...

Count: 9

Count:10

This can be easily done by adding strings. Strings are added using the + symbol just like how you would add any numbers.

Try the following code

```csharp
using System;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            int MyVar;
            MyVar = 0;
            while(MyVar<10)
            {
                MyVar++;
                Debug.Print("Count: " + MyVar.ToString());
            }
        }

    }
}
```

Assemblies

Assemblies are files containing compiled (assembled) code. This allows developer to use the code but they don't have access to the source code. We had already used Debug.Print before. Who made the Debug class/object and who made the Print method that is in it? Those calls are made by NETMF team at Microsoft. They compile the code and give you an assembly to use it. This way, users are not messing with the internal code but they can use it.

At the top of the code used before, we see using Microsoft.SPOT;

This tells C# that you want to use the "namespace" Microsoft.SPOT. Okay, then what is a

namespace? Programs are split into regions "spaces". This is very important when programs are very large. Every chunk of code or library is assigned a "name" for its "space". Programs with the same "namespace" see each other but if the name space is different then we can optionally tell C# to "use" the other name space.

The "name" for our program's "space" is namespace MFConsoleApplication1

To "use" other name space like "Microspft.SPOT" you need to add using Microsoft.SPOT;

What is SPOT anyways? Here is a short story! A few years ago, Microsoft privately started a project called SPOT. They realized that this project was a good idea and wanted to offer it to developers. They decided to change the product name to .NET Micro Framework but they kept the code the same way for backward compatibility. In short SPOT is NETMF!

Back to coding, now try to remove or comment out using Microsoft.SPOT; and your code will not work anymore

Here is the error message showing after I commented out using Microsoft.SPOT;
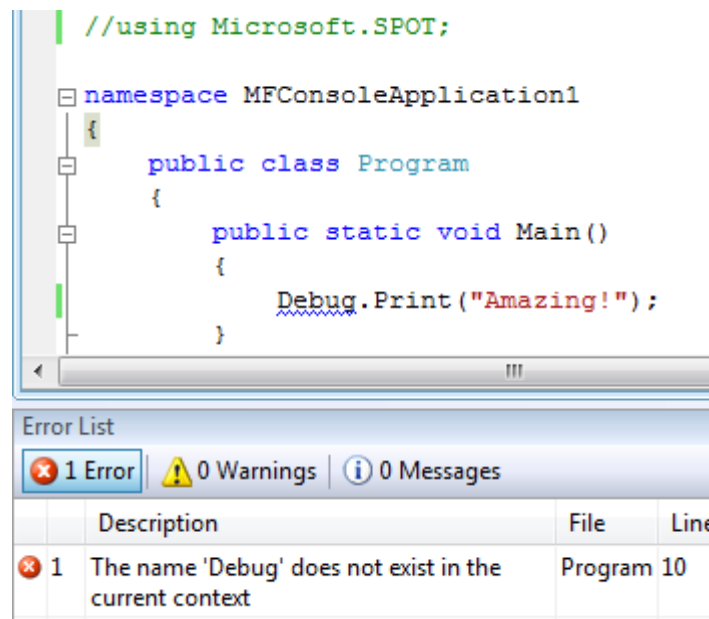
We used the assemblies but where are they added?

Uncomment the code and make sure it still works. Now take a look at "Solution Explorer" window. Click the little + sign by the word "References" and you should see two assemblies.
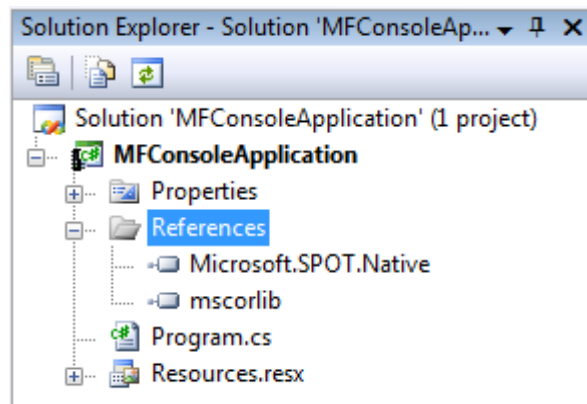
Now, right-click on "Microsoft.SPOT.Native" then click "Remove"
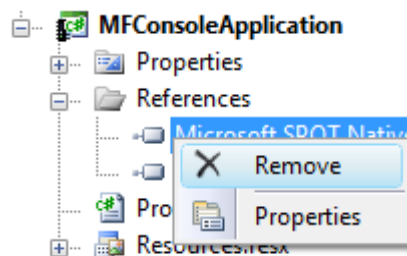
Our program still be exactly the same as before but now it is missing a very important assembly. Try to run it and you will see something like this
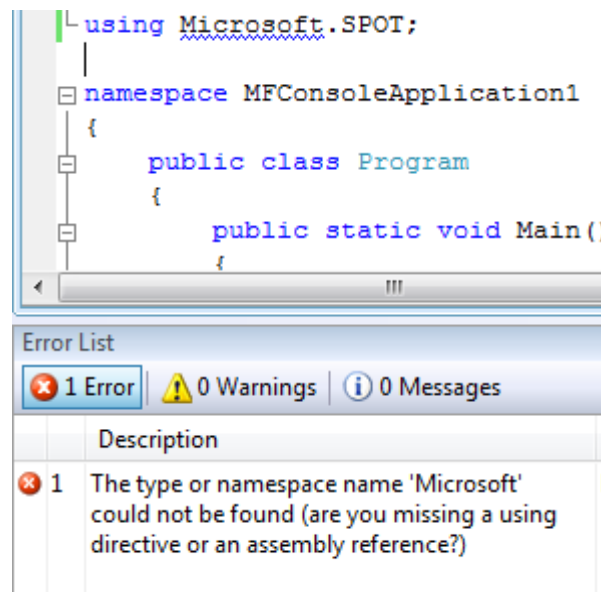
Let's add it back and make sure our program still runs. Right click on the work "References" and select "Add Reference..."
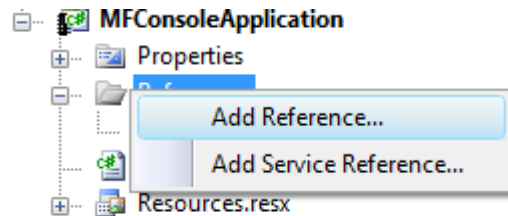
In the new window, select ".NET" tab and then select "Microsoft.SPOT.Native" and click OK.
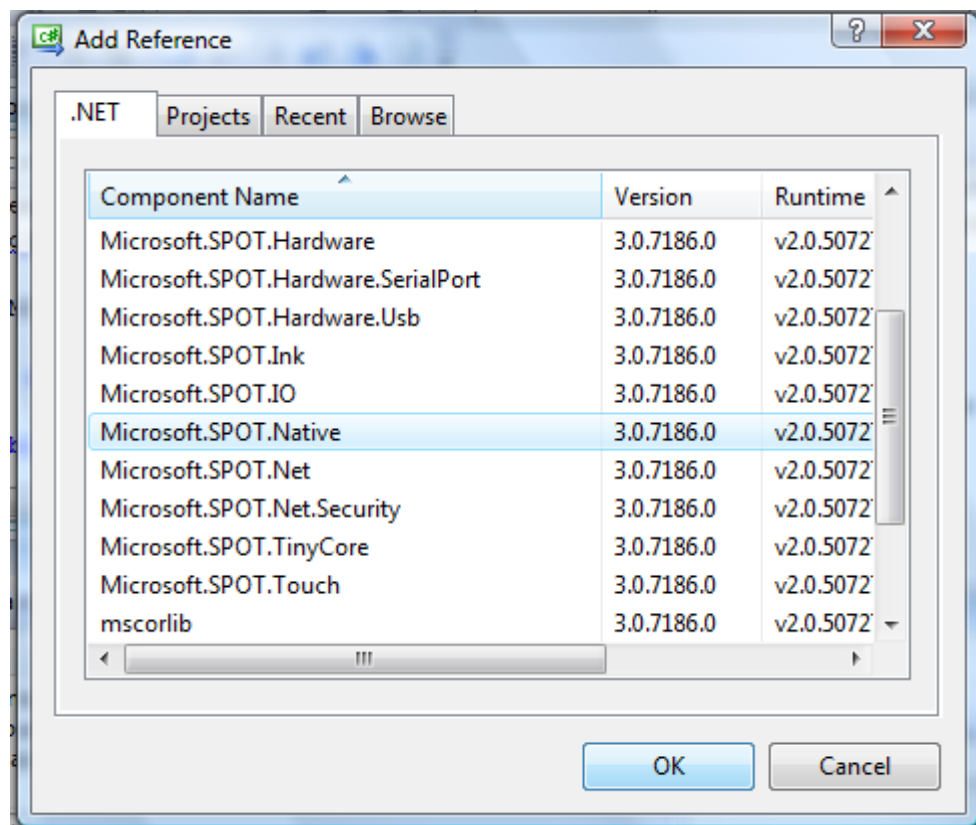
Try the program to make sure its still running. If you have any errors, please go back and read more to fix it before moving on.

What Assemblies to Add?

Throughout this book, I provide many examples but I do not tell you what assemblies I am using. This is really easy to figure out from the documentation but you may find it difficult sometimes. Why not just add them all? As a beginner, your applications are still very small so

you will have a lot of memory even if you add all of the assemblies, even if you are not using them.

The assemblies below are most commonly used. Add them for all of your projects for now. Once you know where everything belongs, you can start removing the ones you don't need.

GHIElectronics.NETMF.Hardware

GHIElectronics.NETMF.IO

GHIElectronics.NETMF.System

Microsoft.SPOT.Hardware

Microsoft.SPOT.Native

Microsoft.SPOT.Hardware.SeriaPort

Microsoft.SPOT.IO

mscorlib

System

System.IO

Don't forget about using one of the following according to what device you are using. These contain the pin definitions.

FEZMini_GHIElectronics.NETMF.FEZ

FEZDomino_GHIElectronics.NETMF.FEZ.

Threading

This can be a very advanced topic. Note that only very basic information is covered here.

Processors/programs only run one instruction at once. Remember how we stepped in the code? Only one instruction got executed and then the flow went on to the next instruction. Then how is it possible that your PC can run multiple programs at the same time? Actually, your PC is never running them a once! What it is doing is running every program for a short time, then it stops it and goes on to run the next program.

Generally, threading is not recommended for beginners but there are things that can be done much easier using threads. For example, you want to blink an LED. It would be nice to blink an LED in a separate thread and never have to worry about it in the main program.

Also, adding delays in the code require the threading namespace. You will understand this better in coming examples.

By the way, LED stands for Light Emitting Diodes. You see LEDs everywhere around you. Take a look at any TV, DVD or electronic device and you will see a little Red or other color light bulb. These are LEDs.

FEZ comes with with a LED library to simplify this even further. This book explains how to directly control pins/devices.

Add "using System.Threading" to your program.

```
using System;
using Microsoft.SPOT;
using System.Threading;
```

That is all we need to use threads! It is important to know that our program itself is a thread. On system execution start-up, C# will look for "Main" and run it in a thread. We want to add a delay in our thread (our program), so it will print the word 'Amazing!' once every second. To delay a "thread", we put it to "Sleep". Note that this Sleep is not for the whole system. It will only "Sleep " the "thread".

Add Thread.Sleep(1000);

The "Sleep" method takes time in milliseconds. So for 1 second we will need 1000 milliseconds.

```
using System;
using Microsoft.SPOT;
using System.Threading;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            while (true)
            {
                Debug.Print("Amazing!");
                Thread.Sleep(1000);
            }
        }

    }
}
```

Try to run the program and look at the output window. If you've tried it on the emulator and it wasn't exactly 1 second, don't worry about it. Try it on real hardware (FEZ) and it will be very

close to 1 second.

Let's create a second thread (our first was automatically created, remember?) We will need to create a new thread object handler (reference) and name it something useful, like MyThreadHandler. And create a new local method and name it MyThread. Then, run the new thread.

We are not using the "Main" thread anymore so I will put it in endless sleep.

Here is the code listing. If you don't understand it then don't worry about it. All is needed at this point is that you know how to "Sleep" a thread.

```csharp
using System;
using Microsoft.SPOT;
using System.Threading;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void MyThread()
        {
            while (true)
            {
                Debug.Print("Amazing!");
                // sleep this thread for 1 seond
                Thread.Sleep(1000);
            }
        }
        public static void Main()
        {
            // create a thread handler
            Thread MyThreadHandler;
            // create a new thread object
            // and assing to my handler
            MyThreadHandler = new Thread(MyThread);
            // start my new thread
            MyThreadHandler.Start();

            /////////////////////////////////
            // Do anything else you like to do here
            Thread.Sleep(Timeout.Infinite);
        }

    }
}
```

9. Digital Input & Output

On processors, there are many "digital" pins that can be used as inputs or outputs. When saying "digital" pins we mean the pin can be "one" or "zero".

Important note: Static discharge from anything including human body will damage the processor. You know how sometimes you touch someone or something and you feel a little electronic discharge? This little discharge is high enough to kill electronic circuits. Professionals use equipments and take precautions handling the static charged in their body. You may not have such equipment so just try to stay from touching the circuit if you don't have to. You may also use an Anti-static wrist band.

NETMF supports digital input and output pins through "Microsoft.SPOT.Hardware" assembly and name space.

Go ahead and add the assembly and namespace like we learned before.
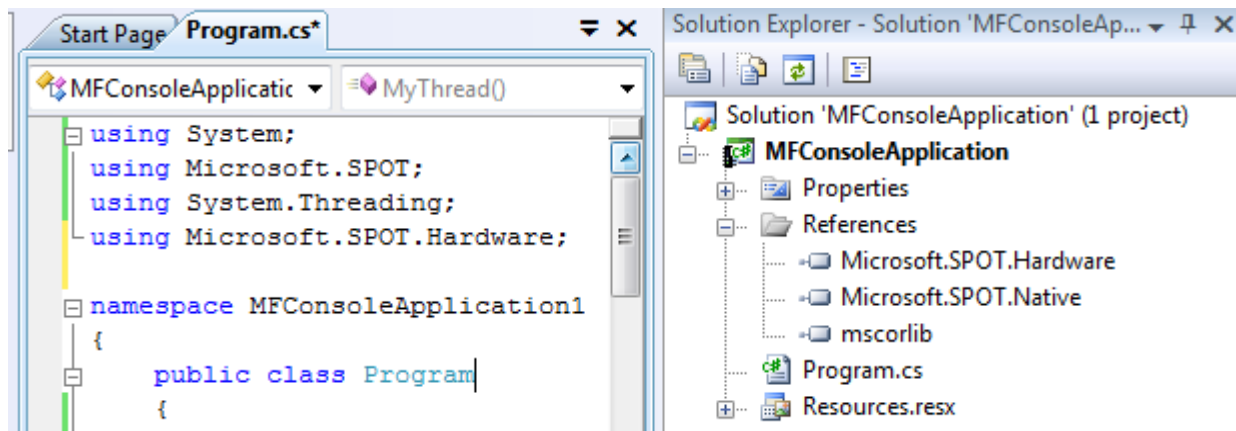
We are now ready to use the digital pins.

9.1. Digital Outputs

We know that a digital output pin can be set to zero or one. Note that one doesn't mean it is 1 volt but it means that the pin is supplying voltage. If the processor is powered off of 3.3V then the state 1 on a pin means that there is 3.3V on the output pin. It is not going to be exactly 3.3V but very close. When the pin is set to zero then it is voltage is very close to zero volts.

Those digital pins are very weak! They can't be used to drive devices that require a lot of power. For example, a motor may run on 3.3V but you can NOT connect it directly to the processor's digital pin. That is because the processor output is 3.3V but with very little power. The best you can do is drive a small LED or "signal" 1 or 0 to another input pin.

All FEZ boards have a LED connected to a digital pin. We want to blink this led.

Digital output pins are controller through OutputPort object. We first create the object handler (reference), and then we make a new OutputPort object and assign it to our handler. When creating a new OutputPort object, you must specify the initial state of the pin, 1 or 0. The one and zero can be referred to high or low and also can be **true for high** and **false for low**. We will make the pin true (high) in this example to turn on our LED by default.

Here is the code using the pin number 4, FEZ Domino on-board LED.

```csharp
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            OutputPort LED;
            LED = new OutputPort((Cpu.Pin)4, true);

            Thread.Sleep(Timeout.Infinite);
        }

    }
}
```

FEZ SDK comes with "FEZMini_GHIElectronics.NETMF.FEZ" and "FEZDomino_GHIElectronics.NETMF.FEZ" assemblies. Add the appropriate assembly to your program then also adds "FEZ_GHIElectronics.NETMF.System".

Now modify the code by adding "using GHIElectronics.NETMF.FEZ" at the top of your code.

Here is the code, this time using the FEZ pin enumeration class.

```csharp
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
```

```
        public static void Main()
        {
            OutputPort LED;
            LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);

            Thread.Sleep(Timeout.Infinite);
        }
    }
}
```

See how much it is much easier? We really do not need to know where the LED is connected.

Run the program and observe the LED. It should be lit now. Things are getting more exciting!

Blink an LED

To blink an LED, we need to set the pin high and delay for some time then we need to set it low and delay gain. Its important to remember to delay twice. Why? It's because our eyes are too slow for computer systems. If the LED comes on and then it turns back off very fast, your eyes will not see that is was on for a very short time.

What do we need to blink a LED? ... We learned how to make a while-loop, we know how to delay, and we need to know how to set the pin high or low. This is done by calling Write method in the OutputPort object. Note that you can't use "OutputPort.Write" This is very wrong because what output ports you are referring to? Instead, use "LED.Write" which make complete scene.

Here is the code to blink the on-board LED on FEZ Domino/Mini

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            OutputPort LED;
            LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
            while (true)
            {
                LED.Write(!LED.Read());

                Thread.Sleep(200);
```

```
                }
            }
        }
}
```

This is another way, simpler way, to blink an LED.

```csharp
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            OutputPort LED;
            LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
            while (true)
            {
                LED.Write(true);
                Thread.Sleep(200);

                LED.Write(false);
                Thread.Sleep(200);
            }
        }
    }
}
```

Let see if you can change the sleep time to make the LED blink faster or slower. Also, try to use a different value for its state so it is on for long time and then it is off for short time.

Important note: Never connect two output pins together. If they are connected and one is set to high and the other is set to low, you will damage the processor. Always connect an output pin to an input, driving circuit or a simple load like an LED.

9.2. Digital Inputs

Digital inputs sense if the state of its pin is high or low. There is limitation on these input pins. For example, the minimum voltage on the pin is 0 volts. A negative voltage may damage the pin or the processor. Also, the maximum you can supply to the pin must be less than the processor power source voltage. All GHI Electronics boards use processors that run on 3.3V

so the highest voltage the pin should see is 3.3V. This is true for ChipworkX but for Embedded Master and USBizi, those processors are 5V-tolerant. This means that even though the processor runs on 3.3V, it is capable of tolerating up to 5V on its inputs. Most digital chips that you would be interfacing to are 5V. Being 5V tolerant allows us to use any of those digital circuits with our processor.

Note: FEZ is based on USBizi and so it is 5V tolerant.

Important note: 5V-tolerant doesn't mean the processor can be powered off 5V. Always power it with 3.3V. Only the input pins can tolerate 5V on them.

InputPort object is used to handle digital input pins. Any pin on the processor GHI uses can be input or output, but of course, not both! Unconnected input pins are called floating. You would think that unconnected input pins are low but this is not true. When a pin is an input and is not connected, it is open for any surrounding noise which can make the pin high or low. To take care of this issue, modern processors include and internal weak pull-down or pull-up resistors, that are usually controlled by software. Enabling the pull-up resistor will pull the pin high. Note that the pull-up resistor doesn't make a pin high but it pulls it high. If nothing is connected then the pin is high by default.

There are many uses for input ports but the most common is to connect it to a button or a switch. FEZ already includes an on-board button connected to the loader pin. The loader pin is used on power up to enter the boot loader but we can still use this pin at run-time. The button is enumerated as "LDR" or "Loader".

The button will connect between ground and the input pin. We will also enable the pull-up resistor. This means that the pin will be high (pull-up) when button is not pressed and low (connected to ground) when the button is pressed.

We will read the status of the button and pass its state to the LED. Note that the pin is high when the button is not pressed (pulled-high) and it is low when the button is pressed. This means the LED will turn off when the button is pressed.

The code:

```csharp
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            OutputPort LED;
            InputPort Button;

            LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
```

```
                  Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, false,
                                       Port.ResistorMode.PullUp);
              while (true)
              {
                  LED.Write(Button.Read());
                  Thread.Sleep(10);
              }
          }
      }
}
```

Creating the InputPort object included "false" in it. This is used for glitch filter. This will be explained later. Also, it maybe is confusing how we passed the state of an InputPort to set an OutputPort. We will revisit this in the next section.

9.3. Interrupt Port

If we want to check the status of a pin, we will always have to check its state periodically. This wastes processor time on something not important. You will be checking the pin, maybe, a million times before it is pressed! Interrupt ports allows us to set a method that will be executed when the button is pressed (when pin is low for example).

We can set the interrupt to fire on many state changes on the pin, when pin is low or maybe when it is high. The most common use is the "on change". The change from low to high or high to low creates a signal edge. The high edge occurs when the signal rises from low to high. The low edge happen when the signal falls from high to low.

In the example below, I am using both edges so our method "IntButton_OnInterrupt" will automatically run whenever the state of our pin changes.

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
        // this moved out here so it can be used by other methods
        static OutputPort LED;

        public static void Main()
        {
            LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
            // the pin will generate interrupt on high and low edges
```

```
            InterruptPort IntButton =
                new InterruptPort((Cpu.Pin)FEZ_Pin.Interrupt.LDR, true,
                                  Port.ResistorMode.PullUp,
                                  Port.InterruptMode.InterruptEdgeBoth);

            // add an interrupt handler to the pin
            IntButton.OnInterrupt +=
                new NativeEventHandler(IntButton_OnInterrupt);

            //do anything you like here
            Thread.Sleep(Timeout.Infinite);
        }

        static void IntButton_OnInterrupt(uint port, uint state,
                                          DateTime time)
        {
            // set LED to the switch state
            LED.Write(state == 0);
        }
    }
}
```

Note: Not all pins on the processor support interrupts, but most of them do. For easier identification of the interrupt pins, use the enumeration for "Interrupt" instead of "Digital", as shown in earlier code.

9.4. Tristate Port

If we want a pin to be an input and output, what can do? A pin can never be in and out simultaneously but we can make it output to set something and then make it input to read a response back. One way is to "Dispose" the pin. We make an output port, use it and then dispose it. Then we can make the pin input and read it.

NETMF supports better options for this, through Tristate port. Tristate means three states; that is input, output low and output high. One minor issue about tristate pins is that if a pin is set to output and then you set it to output again then we will receive an exception. One way to come around this is by checking the direction of the pin before changing it. The direction of the pin is in its property "Active" where false means input and true is output. I personally do not recommend the use of Tristate ports unless absolutely necessary.

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
```

```csharp
public class Program
{
    static void MakePinOutput(TristatePort port)
    {
        if (port.Active == false)
            port.Active = true;
    }
    static void MakePinInput(TristatePort port)
    {
        if (port.Active == true)
            port.Active = false;
    }
    public static void Main()
    {
        TristatePort TriPin =
            new TristatePort((Cpu.Pin)FEZ_Pin.Interrupt.LDR, false,
                             false, Port.ResistorMode.PullUp);
        MakePinOutput(TriPin);// make pin output
        TriPin.Write(true);
        MakePinInput(TriPin);// make pin input
        Debug.Print(TriPin.Read().ToString());
    }
}
```

Note: Due to internal design, TristatePort will only work with interrupt capable digital pins.

Important Note: Be careful not to have the pin connected to a switch then set the pin to output and high. This will damage the processor. I would say, for beginner applications you do not need a tristate port so do not use it till you are comfortable with digital circuits.

10. C# Level 2

10.1. Boolean Variables

We learned how integer variables hold numbers. In contrast, Boolean variables can only be true or false. A light can only be on or off, representing this using an integer doesn't make a lot of sense but using Boolean, it is true for on-state and false for off-state. We have already used those variables to set digital pins high and low, LED.Write(true);

To store the value of a button in a variable we use

```
bool button_state;
button_state = Button.Read();
```

We also used while-loops and we asked it to loop forever, when we used true for the satement

```
while (true)
{
    //code here
}
```

Take the last code we did and modify it to use a boolean, so it is easier to read. Instead of passing the Button state directly to the LED state, we read the button state into button_state boolean then we pass the button_state to set the LED accordingly.

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            OutputPort LED;
            InputPort Button;
            bool button_state;
```

```
                LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);

                Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, false,
                Port.ResistorMode.PullUp);

                while (true)
                {
                    button_state = Button.Read();
                    LED.Write(button_state);
                    Thread.Sleep(10);
                }
            }
        }
}
```

Can you make an LED blink as long as the button is pressed?

```
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            OutputPort LED;
            InputPort Button;

            LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
            Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, false,
                                Port.ResistorMode.PullUp);
            while (true)
            {
                while (Button.Read() == false)//Button is false when pressed
                {
                    LED.Write(true);
                    Thread.Sleep(300);
                    LED.Write(false);
                    Thread.Sleep(300);
                }
            }
        }
    }
}
```

Important note: The == is used to check for eqality in C#. This is different from = which is

used to assign values.

10.2. if-statement

An important part of programming is checking some state and takes action accordingly. For example, "if" the temperature is over 80, turn on the fan.

To try the if-statement with our simple setup, we want to turn on the LED "if" the button is pressed. Note this is the opposite from what we had before. Since in our setup, the button is low when it is pressed. So, to achieve this we want to invert the state of the LED from the state of the button. If the button is pressed (low) then we want to turn the LED on (high). LED need to be checked repeatedly so we will do it once every 10ms.

```csharp
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            OutputPort LED;
            InputPort Button;
            bool button_state;
            LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
            Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, false,
                                    Port.ResistorMode.PullUp);
            while (true)
            {
                button_state = Button.Read();

                if (button_state == true)
                {
                    LED.Write(false);
                }

                if (button_state == false)
                {
                    LED.Write(true);
                }

                Thread.Sleep(10);
            }
        }
    }
}
```

```
}
```

10.3. if-else-statements

We learned how if-statement works. Now, we want to use else-statement. Basically, "if" a statement is true, the code inside the if-statement runs or "else" the code inside else-statement will run. With this new statement, we can optimize the code above to be like this

```csharp
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            OutputPort LED;
            InputPort Button;
            bool button_state;
            LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
            Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, false,
                                Port.ResistorMode.PullUp);
            while (true)
            {
                button_state = Button.Read();

                if (button_state == true)
                {
                    LED.Write(false);
                }
                else
                {
                    LED.Write(true);
                }

                Thread.Sleep(10);
            }
        }
    }
}
```

I will let you in on a secret! We only used if-statement and else-statement in this example for

demonstration purposes only. We can write the code this way.

```csharp
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            OutputPort LED;
            InputPort Button;

            LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
            Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, false,
                                Port.ResistorMode.PullUp);
            while (true)
            {
                LED.Write(Button.Read() == false);
                Thread.Sleep(10);
            }
        }
    }
}
```

Or even this way!

```csharp
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            OutputPort LED;
            InputPort Button;

            LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
            Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, false,
                                Port.ResistorMode.PullUp);
```

```
            while (true)
            {
                LED.Write(!Button.Read());
                Thread.Sleep(10);
            }
        }
    }
}
```

Usually, there are many way to write the code. Use what makes you comfortable and with more experience, you will learn how to optimize the code.

10.4. Methods and Arguments

Methods are actions taken by an object. It can also be called a function of an object. We have already seen methods and have used them. Do you remember object Debug which has Print method? We have already used Debug.Print many times before, where we gave it a "string" to display in the output window. The "string" we passed is called an argument.

Methods can take one or more optional arguments but it can only return one optional value.

The Print method in Debug object only takes one string argument. Other method may not require any arguments or may require more than one argument. For example, a method to draw a circle can take four arguments, DrawCircle(posx, posy, diam, color). An example for returning values can be a method that returns temperature.

So far, we have learned of three variable types, int, string and bool. Will cover other types later but remember that everything we talk about here apply to other variable types.

The returned value can be an optional variable type. If no returned value is used then we replace the variable type with "void". The "return" keyword is used to return values at the end of a method.

Here is a very simple method that "returns" the sum of two integers.

```
int Add(int var1, int var2)
{
    int var3;
    var3 = var1 + var2;
    return var3;
}
```

We started the method with the return value type, "int" followed by the method name. Then we have the argument list. Arguments are always grouped by parenthesis and separated by commas.

Inside the Add method, a local integer variable has been declared, that is var3. Local variables are created inside a method and die once we exit the method. After that, we add our two variables and finally return the result.

What if we want to return a string representing the sum of two numbers? Remember that a string containing number 123 is not the same as an integer containing 123. An integer is a number but a string is an array or characters that represent text or numbers. To humans these are the same things but in computer world this is totally different.

Here is the code to return a string.

```csharp
string Add(int var1, int var2)
{
    int var3;
    var3 = var1 + var2;

    string MyString;
    MyString = var3.ToString();

    return MyString;
}
```

You can see how the returned type was changes to string. We couldn't return var3 because it is integer variable and so we had to convert it to a string. To do that, we create a new variable object named MyString. Then convert var3 "ToString" and place the new string in MyString.

The question now is how come we called a "ToString" method on a variable of type interger? In reality, everything in C# is an object even the built in variable types. This book is not going into these details as it is only meant to get you started.

This is all done in multiple steps to show you how it is done but we can compact everything and results will be exactly the same.

```csharp
string Add(int var1, int var2)
{
    return (var1+var2).ToString();
}
```

I recommend you do not write code that is extremely compact, like the example above, till you are very familiar with the programming language. Even then, there should be limits on how much you compact the code. You still want to be able to maintain the code after sometime and someone else may need to read and understand your code.

10.5. Classes

All objects we talked about so far as actually "classes" in C#. In modern object oriented programming languages, everything is an object and methods always belong to one object. This allows for having methods of the same name but they can be for completely different objects. A "human" can "walk" and a "cat" can also "walk" but do they walk the same way? When you call "walk" method in C# then it is not clear if the cat or the human will walk but using human.walk or cat.walk makes it clearer.

Creating classes is beyond the scope of this book. Here is a very simple class to get you started

```csharp
class MyClass
{
    int Add(int a, int b)
    {
        return a + b;
    }
}
```

10.6. Public vs. Private

Methods can be private to a class or publicly accessible. This is useful only to make the objects more robust from programmer misuse. If you create an object (class) and this object has methods that you do not anyone to use externally then add the keyword "private" before the method return type; otherwise, add the "public" keyword.

Here is a quick example

```csharp
class MyClass
{
    public int Add(int a, int b)
    {
        // the object can use private methods
        // inside the class only
        DoSomething();
        return a + b;
    }
    private void DoSomething()
    {
    }
}
```

10.7. Static vs. non-static

Some objects in life have multiple instances but others only exist once. The objects with multiple instances are non-static. For example, an object representing a human doesn't mean much. You will need an "instance" of this object to represent one human. So this will be something like

human Mike;

We now have a "reference" called Mike of type human. It is important to note that this reference is at this point not referencing to any object (no instance assigned) just yet, so it is referencing NULL.

To create the "new" object instance and reference it from Mike

Mike = new human();

We now can use any of the human methods on our "instance" Mike

Mike.Run(distance);

Mike.Eat();

bool hungry = Mike.IsHungry();

We have used those non-static methods already when we controlled input and output pins.

When creating a new non-static object, the "new" keyword is used with the "constructor" of the object. The constructor is a special type of method that returns no vale and it is only used when creating (construction) new objects.

Static methods are easier to handle because there is only one object that is used directly without creating instances. The easiest example is our Debug object. There is only one debug object in the NETMF system so using its methods, like Print, is used directly.

Debug.Print("string");

I may not have used the exact definitions of static and instances but I wanted to describe it in the simplest possible way.

10.8. Constants

Some variables may have fixed values that never change. What if you accidentally change the value of this variable in your code? To protect it from changing, add the "const" keyword before the variable declaration.

```
const int hours_in_one_day = 24;
```

10.9. Enumeration

An Enumeration is very similar to constant. Let's say we have a device that accepts four commands, those are MOVE, STOP, LEFT, RIGHT. This device is not a human and so these commands are actually numbers. We can create constants (variables) for those four commands like the following

```
const int MOVE = 1;
const int STOP = 2;
const int RIGHT = 3;
const int LEFT = 4;
//now we can send  a command...
SendCommand(MOVE);
SendCommand(STOP);
```

The names are all upper case because this is how programmers usually name constants. Any other programmer seeing an upper case variable would know that this is a constant.

The code above is okay and will work but it will be nicer if we can group those commands

```
enum Command
{
    MOVE = 1,
    STOP = 2,
    RIGHT = 3,
    LEFT = 4,
}
//now we can send  a command...
SendCommand(Command.LEFT);
SendCommand(Command.STOP);
```

With this new approach, there is no need to remember what commands exist and are the command numbers. Once the word "Command" is typed in, Visual Studio will give you a list of available commands.

C# is also smart enough to increment the numbers for enumerations so the code can be like this listing and will work exactly the same way

```
enum Command
{
    MOVE = 1,
    STOP ,
    RIGHT,
    LEFT ,
```

```
}
//now we can send a command...
SendCommand(Command.LEFT);
SendCommand(Command.STOP);
```

11. Assembly/Firmware Matching

NETMF devices usually include extended features. Those extended features require an extra assembly/library to be added to the C# projects so a user can make use of the new features. For example, NETMF doesn't support Analog pins but FEZ and other hardware from GHI does support analog pins. To use the analog pins, you need to add an assembly/library provided by GHI then you have new classes/objects that can be used to access those new features.

Important Note: The firmware will fail to run if the version of the assembly/library that used in the project does not match the version of the firmware.

This is very common issue that users run into when updating the firmware where the application just stop working and debugging seem to fail.

Here is what happens:

Scenario #1: A developer had received a new device. This device happens to have firmware version 1.0 on it. Then the developer went to the website and downloaded the latest SDK. The SDK had firmware version 1.1 in it. When trying to upload a project, VS2010 will fail to attach to the device with no indication why! The developer will now think the new device is not functioning, actually, the device is just fine but, in this example, the firmware is version 1.0 and the assembly is version 1.1 so the system will refuse to run. To fix the issue, update the firmware on the device to match the firmware in the SDK.

Scenario #2: A developer has a perfectly working system that, for example, uses firmware version 2.1. Then a new SDK comes out with firmware version 2.2, so the developer installs the new SDK on the PC then uploads the new firmware to the device (FEZ). When rebooting the device, it stops working because the new loaded firmware is version 2.2 but the user application still uses assembly version 2.1. To fix this issue, open the project that has the user application and remove any device-specific assemblies. After they are removed, go back and add them back. With this move the new files will be fetched from the new SDK.

Boot-up Messages

We can easily see why the system is not running using MFDeploy. NETMF outputs many useful messages on power up. Should the system become unresponsive, fails to run or for any other debug purposes, we can use MFDeploy to display these boot up messages. Also, all "Debug.Print" messages that we usually see on the output window are visible on MFDeploy.

To display the boot up messages, click on "Target->Connect" from the menu then reset the device. Right after you reset the device in one second, **click on "ping"**. MFDeploy will freeze for a second then display a long list of messages.
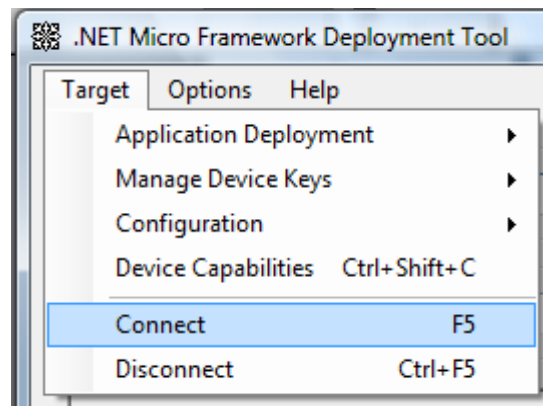
Note that on FEZ Domino the reset button is available on the board. For FEZ Mini, you need to connect a switch to the reset pin or if you have FEZ Mini in the starter kit or robot kit then you can use the reset button on the board.
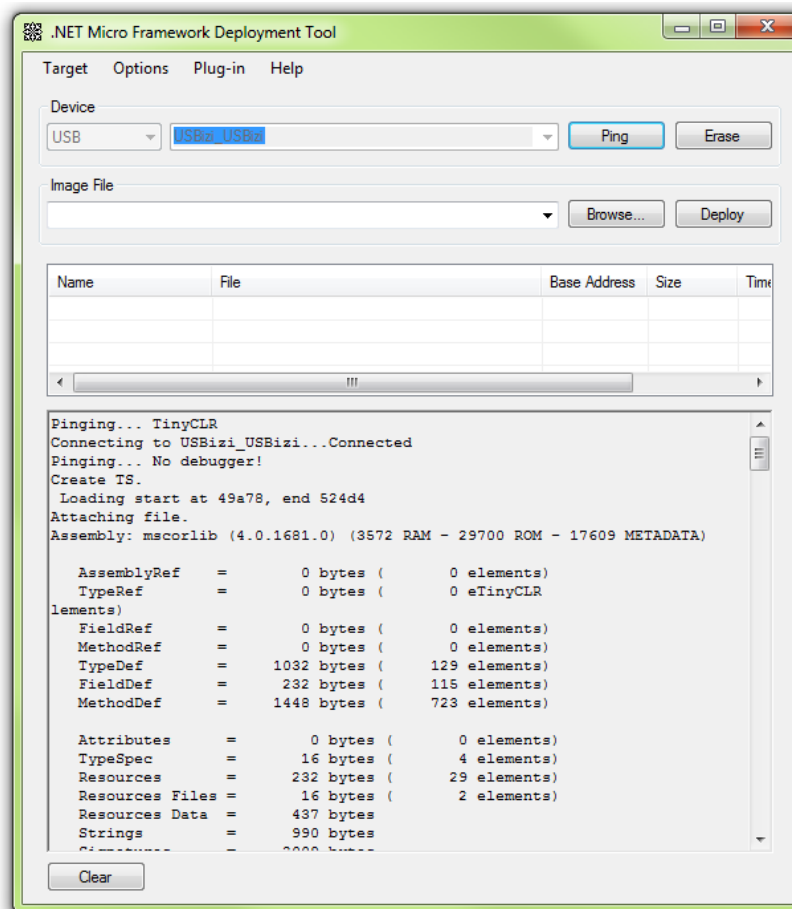
12. Pulse Width Modulation

PWM is a way of controlling how much power is provided to a device. Dimming a LED or slowing down a motor is best done using a PWM signal. If we apply power to a LED it comes completely on and if we shut off the power then it is completely off. Now, what if we turn the LED on for 1ms and then off for 1ms? In reality it is blinking on and off very fast but our eyes would never see it blinking that fast causing our eyes to see the LED to be dimmer than before.

We calculate this using (on/(off+on))x100= 50%. So, the LED is only getting half the energy.

So if we turn the LED on for 1ms and off for 9ms we will end up with (1/(1+9))x100=10% energy, so the LED is very dimmed.

PWM is very simple to generate but if we are going to toggle a pin few hundred or thousand times a second then we are wasting a lot of processor power. There are devices that generate PWM signals more efficiently. Also, many processors include a built-in circuitry needed to generate PWM in hardware. This means, we can setup the PWM hardware to generate a PWM signal and then it is done automatically without the need for processor interaction.

FEZ included few pins that can serve as PWM pins. The FEZ library includes everything needed to activate PWM on any one of those pins.

Here is an example that creates a PWM object that sets the PWM to 10 Khz (10,000 clocks/sec) and the duty cycle to 50 (50% energy)

```
PWM pwm = new PWM((PWM.Pin) FEZ_Pin.PWM.LED);
pwm.Set(10000, 50);
```

FEZ includes an enumeration of what pins have PWM hardware. Using the PWM enumeration, we can easily find the PWM pins on your device. Visual studio will give you a list of them while you are typing your code, just like this image.
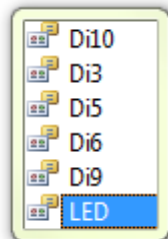
From the list above, you can see that the LED is connected to a PWM signal. Instead of blinking LED, what about we fade it in and out? This will make it look so much cooler! Here is the code.

```csharp
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            sbyte dirr = 1;
            byte duty = 40;
            PWM pwm = new PWM((PWM.Pin)FEZ_Pin.PWM.LED);
            while (true)
            {
                pwm.Set(10000, duty);
                duty = (byte)(duty + dirr);
                if (duty > 90 || duty < 10)
                {
                    dirr *= -1;
                }

                Thread.Sleep(10);
            }

        }
    }
}
```

The code will loop while incrementing the energy of the LED to 90% and then goes back and decrement it to 10% and so on.

Important: PWM will be perfect to control how fast your robot will move or how fast a fan is spinning. A pin with PWM signal can power an LED just fine. This is not the case if we are controlling something with high power demands, like a motor or a light bulb. In this case, we would need to use the PWM signal to control a circuitry and that circuitry will supply the needed power to the device. For example, H-bridge circuits are most commonly used to control the speed and direction of motors.

Important note: All hardware PWM signals share the same clock. Changing the frequency on one will change it on other channels, but duty cycle is independent.

Simulating PWM

PWM pins are controlled internally inside the processor using a special PWM hardware. This means, the hardware internally will toggle the pin. The processor only needs to set some registers and then no processor interaction is needed at all to generate the PWM signal.

GHI's NETMF devices provide an OutputCompare class. Through this class a user can generate about any signal, including PWM. Note that this is done in software so it is better to use the PWM class is possible. Why? When you use the PWM class then the signal for PWM is generated using hardware and the processor doesn't need to do anything. On the other hand, when you use OutputCompare to generate PWM then the processor need to keep track of time and toggle the PWM pin at specific time constants.

The servo motor example provided on www.TinyCLR.com uses OutputCompare. The advantage of this is that you can use any pin to control a servo. It is better to use PWM to control the servo but then you will only be able to use the PWM specific pins.

Controlling Servo Motors and Fine-tune the PWM

We have learned how to control the amount of energy using PWM. There are other good uses for PWM, such as controlling servos. The PWM class provides two methods Set and SetPulse methods. Earlier, we utilized PWM. Set which very well suited for setting energy levels. For controlling servos, using SetPulse is more suitable.

So how do servos work? This is best explained in this image I found on www.pc-control.co.uk
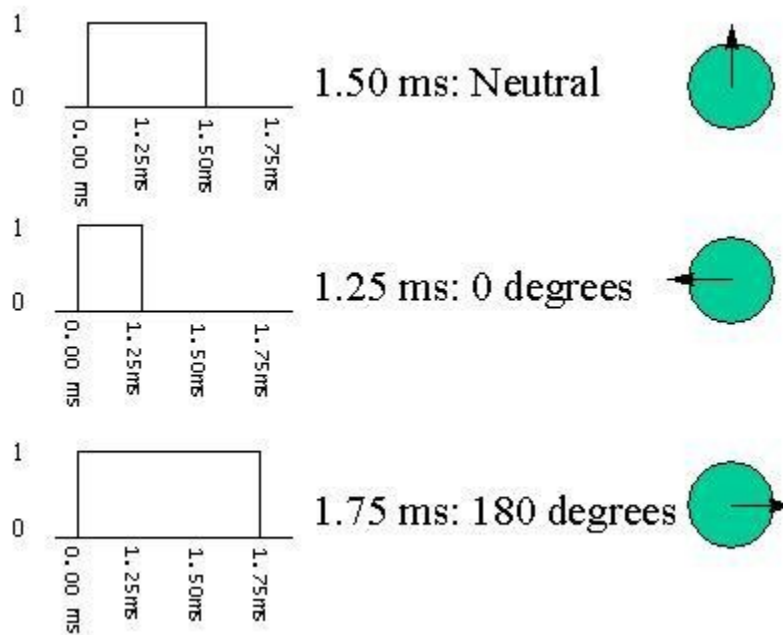
You can control the position of a servo by providing it with a pulse (see image above). If the pulse width is about 1.25ms then the servo is at 0 degrees. Increasing the pulse to 1.25ms will move the servo to 90 degrees (neutral). A wider pulse of 1.75ms will move the servo to 180 degrees. Okay, those is easy to generate using PWM but are still missing one piece of info. The image tells us how wide (in time) is the "high pulse" but what about the "low time"? Servos expect a pulse every 20ms to 30ms but this is not very critical to servos. So, we now have everything we need. Before I proceed, I want to explain what the "period" is. A period of a frequency (signal) is the high time plus the low time. Now we are ready for some code.

The method PWM.SetPulse needs the high time and the period of the signal. Since our servo low time is not critical, can be 20ms to 30ms then I will just use a period of 20ms. This will result in a low time that will vary slightly but again, this is not important to servos. What is important is the high pulse which must be between 1.25ms to 1.75ms.

One last thing and we are done! SetPulse accepts value in nanoseconds but we have value in milliseconds. One millisecond is the equivalent of 1,000,000 (a million) nanoseconds.

Finally, here is the example.

```csharp
using System;
using System.Threading;
using Microsoft.SPOT;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.FEZ;
class Program
{
    public static void Main()
    {
        PWM servo = new PWM((PWM.Pin)FEZ_Pin.PWM.Di5);
        while (true)
        {
            // 0 degrees. 20ms period and 1.25ms high pulse
            servo.SetPulse(20 * 1000 * 1000, 1250 * 1000);
            Thread.Sleep(1000);//wait for a second

            // 90 degrees. 20ms period and 1.50ms high pulse
            servo.SetPulse(20 * 1000 * 1000, 1500 * 1000);
            Thread.Sleep(1000);//wait for a second

            // 180 degrees. 20ms period and 1.75ms high pulse
            servo.SetPulse(20 * 1000 * 1000, 1750 * 1000);
            Thread.Sleep(1000);//wait for a second
        }
        Thread.Sleep(-1);
    }
}
```

12.1. Piezo

Piezoelectric devices are a low cost way of adding tones and beeps to your design. They generate sounds one supplied with a frequency. PWM hardware is perfect for generating frequencies and so they can be perfect to activate a Piezo. We will always use duty cycle of 50% but we will change the frequency to get different tones.

When connecting a Piezo, make sure that it its polarity is correct, plus goes to PWM signal and minus goes to ground.

See the FEZ Piezo speaker component on www.TinyCLR.com

This is a project that decodes a MIDI file from a SD card then plays the main tone on a piezo.

http://www.microframeworkprojects.com/index.php?title=PWM_MIDI_Player

13. Glitch filter

When we used the button earlier (section 9.2), we set it up so when it is pressed, the pin value is low; otherwise, it is high. When you flip a witch or press a button, the button or switch can bounce when it is pressed. In other words, the button generates few on and off before it is settled. This will make it look like if the button was pressed few times. Those few on/off bounces come for a very short time. To eliminate them in hardware, we can add a capacitor between the pin and ground. To handle this in software, we check the time between button presses, if it is a short time then this is a "glitch" and so it needs to be ignored. A user can press the button on and off maybe every 200 ms. So, if the time between presses is 10ms then we know the user couldn't have possible pressed it so fast, and for that, this is a glitch and we need to "glitch filter" it.

Luckily, NETMF includes a glitch filter built internally so we never need to worry about this. When we enable an input pin, we have the option of enabling the glitch filter. The second argument when creating an "InputPort" is a boolean indicating if the glitch filter to be enabled or not. For using switches, you probably would want this to always be true

```csharp
Button = new InputPort((Cpu.Pin)FEZ_Pin.Digital.LDR, true,
Port.ResistorMode.PullUp);
```

The glitch filter time can be changed as follows

```csharp
TimeSpan ts = new TimeSpan(0, 0, 0, 0, 200);//200ms
Microsoft.SPOT.Hardware.Cpu.GlitchFilterTime = ts;
```

Important note: Glitch filter only works on interrupt-capable pins. If you try to use InputPort with glitch filter is set to true and you see an exception then most likely you are using a pin that is not interrupt capable.

14. Analog input & output

Analog pins are usually multiplexed with digital pins. Some of the processor pins can be digital and analog as well but not both at the same time.

14.1. Analog Inputs

Digital input pins can only read high and low (one or zero) but analog pins can read the voltage level. There are limitations on voltages that can be applied to analog inputs. For example, FEZ analog input can read voltages anywhere between zero and 3.3V. When the pins are digital they are tolerant of 5V but when the same pin is set to analog only up to 3.3V can be used. This limitation of analog inputs is not a big issue usually because most analog signals are conditioned to work with the analog inputs. A voltage divider or an op-amp circuit can be used to get a fraction of the actual signal, scale the signal. For example, if we want to measure the battery voltage, which is 6V, then we need to divide the voltage in half, using a voltage divider resistor(s), so the analog pin will only see half the voltage, which is 3V max. In software, we know we have the voltage divided in half so any voltage we see will need to be multiplied by 2 to give us the actual voltage we are trying to measure.

Luckily, GHI implementation of analog inputs already handles signal scaling. When constructing a new analog input object, optionally, you can set the scaling.

The internal reference is 0V to 3.3V so everything you measure needs to take this in mind. The easiest is to set the scale to 0 to 3300. You can think of this as millivolts. If we read 1000 then the input voltage is 1 volt.

```
AnalogIn BatteryVoltage = new AnalogIn((AnalogIn.Pin)FEZ_Pin.AnalogIn.An0);
BatteryVoltage.SetLinearScale(0, 3300);
int voltage = BatteryVoltage.Read();
```

Remember to use the analog pin enumeration to determine what pins can be used as analog.

To print the analog value to the debug output in volts, we would need to convert the value to volts and then to a string.

```
AnalogIn BatteryVoltage = new AnalogIn((AnalogIn.Pin) FEZ_Pin.AnalogIn.An0);
BatteryVoltage.SetLinearScale(0, 3300);
int voltage = BatteryVoltage.Read();
Debug.Print("Voltage = " + (voltage / 1000).ToString() + "." + (voltage %
1000).ToString());
```

We divide by 1000 to get the voltage and then we use the modules to get the fraction.

14.2. Analog Outputs

Analog outputs are like digital outputs where they have limits on how much power they can provide. Analog outputs are even weaker than digital outputs. They are only capable of providing a very little signal to drive the power circuit, maybe drive a power amplifier.

Digital outputs can only be high or low but analog input can be set to any voltage between 0 and 3.3V. GHI implementation for analog outputs allows for automated scaling. You can give it a min, max and the actual value.

An easy test would be to set the min to zero and max to 330V (3.3Vx100) then the value to 100 (1Vx100). This will give us 1V on the pin. We will connect this pin to another analog input to measure the value to verify it is 1V. It may not be exactly 1V but will be very close.

The analog output is multiplexed with analog input 3. When using the analog output, the analog input can' be used on this pin but we can use any other analog input on other pins.

```csharp
using System;
using Microsoft.SPOT;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            AnalogOut VoltLevel =
                new AnalogOut((AnalogOut.Pin)FEZ_Pin.AnalogOut.An3);
            VoltLevel.SetLinearScale(0, 3300);
            VoltLevel.Set(1000);

            AnalogIn PinVoltage =
                new AnalogIn((AnalogIn.Pin)FEZ_Pin.AnalogIn.An0);
            PinVoltage.SetLinearScale(0, 3300);

            while (true)
            {
                int voltage = PinVoltage.Read();
                Debug.Print("Voltage = " + (voltage / 1000).ToString() +
                            "." + (voltage % 1000).ToString());

                Thread.Sleep(200);
```

```
                }
            }
        }
}
```

Connect a wire between An0 and AOUT (An3) and then run this program. Note if no wire is connected then the analog pin is floating and can be of any value. Try touching AIN0 and see how the numbers change then connect to AOUT to see if you get back 1V. If you have a volt meter, connect it to AOUT to verify we have 1V. Note that it is not going to be exactly 1V but very close.

15. Garbage Collector

When programming in older languages like C or C++, programmers had to keep track of objects and release them when necessary. If an object is created and not released then this object is using resources from the system that will never be freed. The most common symptom is memory leaks. A program that is leaking memory will contentiously use more memory till the system run out of memory and probably crash. Those bugs are usually very difficult to find in code.

Modern languages have garbage collector that keeps track of used objects. When the system runs low on memory resources, the garbage collector jumps in and search through all objects and frees the one with no "references". Do you remember how we created objects before using the "new" keyword and then we assigned the object to a "reference"? An object can have multiple references and the garbage collector will not remove the object till it has zero references.

```csharp
// new object
OutputPort Ref1 = new OutputPort(FEZ_Pin.Digital.LED, true);
// second reference for same object
OutputPort Ref2 = Ref1;
// lose the first reference
Ref1 = null;
// Our object is still referenced
// it will not be removed yet
// now remove the second reference
Ref2 = null;
// from this point on, the object is ready to be
// removed by the garbage collector
```

Note that the object is not removed immediately. When needed, the Garbage collector will run and remove the object. This can be an issue in some rare cases because the garbage collector needs some time to search and remove objects. It will only be few milliseconds but what if your application can't afford that? If so, the garbage collector can be forced to run at a desired anytime.

```csharp
//force the garbage collector
Debug.GC(true);
```

15.1. Losing Resources

The garbage collector ease object allocation but it can also cause problems if we are not careful. A good example will be on using digital output pins. Lets say we need a pin to be high. We create an OuputPort object and set the pin high. Later on we lose the "reference" for that object for some reason. The pin will still be high when the reference is lost so all is good so far. After few minutes, the garbage collector kicks in and it finds this unreferenced object, so it will be removed. Freeing an OutputPort will case the pin to change its state to input. Now, the pin is not high anymore!

```csharp
// Turn the LED on
OutputPort LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
LED = null;
// we have lost the reference but the LED is still lit
//force the garbage collector
Debug.GC(true);
// The LED is now off!
```

An important thing to note is that if we make a reference for an object inside a method and the method returns then we have already lost the reference. Here is an example

```csharp
using System;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;

namespace Test
{
    public class Program
    {
        static void TurnLEDOn()
        {
            // Turn the LED on
            OutputPort LED =
                new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
        }
        public static void Main()
        {
            TurnLEDOn();
            // we think that everythign is okay but it is not
            // run the GC
            Debug.GC(true);
            // is LED still on?
        }

    }
}
```

To solve this, we need a reference that is always available. Here is the correct code

```csharp
using System;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;

namespace Test
{
    public class Program
    {
        static OutputPort LED;
        static void TurnLEDOn()
        {
            // Turn the LED on
            LED = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
        }
        public static void Main()
        {
            TurnLEDOn();
            // run the GC
            Debug.GC(true);
            // is the LED on?
        }

    }
}
```

Another good example is using timers. NETMF provide a way to create timers that handle some work after a determined time. If the reference for the timer is lost and the garbage collector had run, the timer is now lost and it will not run as expected. Timers are explained in later chapter.

Important note: If you have a program that is working fine but then right after you see the GC running in the "Output Window" the program stops working or raises an exception then this is because the GC had removed an object that you need. Again, that is because you didn't keep references to the object you want to keep alive.

15.2. Dispose

The garbage collector will free objects at some point but if we need to free one particular object immediately? Most objects have a Dispose method. If an object needs to be freed at anytime, we can "dispose" it.

Disposing object is very important in NETMF. When we create a new InputPort object, the assigned pin is reserved. What if we want to use the same pin as an output? Or even use the same pin as an analog input? We will first need to free the pin and then create the new object.

```
OutputPort OutPin = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.Di5, true);
OutPin.Dispose();

InputPort InPort = new InputPort((Cpu.Pin)FEZ_Pin.Digital.Di5, true,
                        Port.ResistorMode.PullUp);
```

15.3. GC Output Messages

When the garbage collector runs, it outputs a lot of useful information to the output window. These messages give you an idea of what is using resources in the system. Although not recommended, you may want to disable those messages to free up the output window for your own usage. This is easily achievable using this line of code.

```
Debug.EnableGCMessages(false);
```

16. C# Level 3

This section will cover all C# materials we wanted to include in this book. A good and free eBook to continue learning about C# is available at

http://www.programmersheaven.com/2/CSharpBook

16.1. Byte

We learned how int are useful to store numbers. They can store very large numbers but every int consumes four bytes of memory. You can think of a byte as a single memory cell. A byte can hold any value from 0 to 255. It doesn't sound like much but this is enough for a lot of things. In C# bytes are declared using "byte" just like how we use "int".

```
byte b = 10;
byte bb = 1000;// this will not work!
```

The maximum number that a byte can hold is 256 [0..255], What's going to happen if we increment it to over 255? Incrementing 255 by one would overlap the value back to zero.

You will probably want to use int for most of your variables but we will learn later where bytes are very important when we start using arrays.

16.2. Char

To represent a language like English, we need 26 values for lower case and 26 for upper case then 10 for numbers and maybe another 10 for symbols. Adding all these up will give us a number that is well less than 255. So a byte will work for us. If we create a table of letters, numbers and symbols, we can represent everything with a numerical value. Actually, this table already exists and it is called ASCII table.

So far a byte is sufficient to store all "characters" we have in English. Modern computer systems have expanded to include other languages, some used very complex non-Latin characters. The new characters are called Unicode characters. Those new Unicode characters can be more than 255 and so a byte is not sufficient and an integer (four bytes) is too much. We need a type that uses 2-bytes of memory. 2-bytes are good to store numbers from 0 to over 64,000. This 2-bytes type is called "short", which we are not using in this book.

Systems can represent characters using 1-byte or using 2-bytes. Programmers decided to create a new type called char where char can be 1-byte or 2-bytes, depending on the system. Since NETMF is made for smaller systems, its char is actually a byte! This is not the case on

a PC where a char is a 2-byte variable!

Do not worry about all this mess, do not use char if you do not have to and if you use it, remember that it is 1-byte on NETMF.

16.3. Array

If we are reading an analog input 100 times and we want to pass the values to a method, it is not practical to pass 100 variables in 100 arguments. Instead, we create an "array" of our variable type. You can create an array of any object. We will mainly be using byte arrays. When you start interfacing to devices or accessing files, you will always be using byte arrays.

Arrays are declared similar to objects.

```
byte[] MyArray;
```

The code above creates a "reference" of an object of type "byte array". This is only a reference but it doesn't have any object yet, it is null. If you forgot what is a reference then go back and read more in C# Level 2 chapter.

To create the object we use "new" keyword and then we need to tell it the size of our array. This size is the count of how many elements on the type we will have in a n array. Our type is a byte and so the number is how many bytes we are allocating in memory.

```
byte[] MyArray;
MyArray = new byte[10];
```

We have created a byte array with 10 elements in it. The array object is referenced from "MyArray".

We now can store/read any of the 10 values in the array by indicating which "index" we want to access.

```
byte[] MyArray;
MyArray = new byte[10];
MyArray[0] = 123;// first index
MyArray[9] = 99;// last index
MyArray[10] = 1;// This is BAD...ERROR!!
```

A very important note here is that indexes start from zero. So, if we have an array of size 10 then we have indexes from 0 to 9. Accessing index 10 will NOT work and will raise an exception.

We can assign values to elements in an array at the time of initialization. This example will store the numbers 1 to 10 in indexes 0 to 9.

```csharp
byte[] MyArray = new byte[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

To copy an array, use the Array class as follows

```csharp
byte[] MyArray1 = new byte[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
byte[] MyArray2 = new byte[10];
Array.Copy(MyArray1, MyArray2, 5);//copy 5 elements only
```

One important and useful property of an array is the Length property. We can use it to determine the length of an array.

```csharp
byte[] MyArray1 = new byte[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
byte[] MyArray2 = new byte[10];
Array.Copy(MyArray1, MyArray2, MyArray1.Length);//copy the whole array
```

16.4. String

We have already used strings in many places. We will review what we have learned and add more details.

Programs usually need to construct messages. Those messages can be a human readable text. Because this is useful and commonly used feature in programs, C# supports strings natively. C# knows if the text in a program is a string if it is enclosed by double-quotes.

This is a string example.

```csharp
string MyString = "Some string";
string ExampleString = "string ExampleString";
```

Whatever inside the double quotes is colored in red and considered to be a string. Note how in the second line I purposely used the same text in the string to match what I used to assign the string. C# doesn't compile anything in quote (red text) but only take it as it is as a string.

You may still have confusion on what is the difference between a integer variable that have 5 in it and a string that have 5 in it. Here is an example code

```
string MyString = "5" + "5";
int MyInteger = 5 + 5;
```

What do you think the actual value of the variables now? For integer, it is 10 as 5+5=10. But for string this is not true. Strings do not know anything about what is in it, text or numbers make no difference. When adding two strings together, a new string is constructed to combine both. And so "5"+"5"="55" and not 10 like integers.

Almost all objects have a ToString method that converts the object information to a printable text. This demonstration shows how ToString works

```
int MyInteger = 5 + 5;
string MyString = "The value of MyInteger is: " + MyInteger.ToString();
Debug.Print(MyString);
```

Running the above code will print

The value of MyInteger is: 10

Strings can be converted to byte arrays if desired. This is important if we want to use a method that only accepts bytes and we want to pass our string to it. If we do that, every character in the string will be converted to its equivalent byte value and stored in the array

```
using System.Text;
.....
.....
byte[] buffer = Encoding.UTF8.GetBytes("Example String");
```

16.5. For-Loop

Using the while-loop is enough to serve all our loop needs but for-loop can be easier to use in some cases. The simplest example is to write a program that counts from 1 to 10. Similarly, we can blink an LED 10 times as well. The for-loop takes three arguments on a variable. It needs the initial value, how to end the loop and what to do in every loop

```
int i;
for (i = 0; i < 10; i++)
{
    //do something
}
```

We first need to declare a variable to use. Then in the for-loop, we need to give it three arguments (initial, rule, action). In the very first loop, we asked it to set variable "i" to zero. Then the loop will keep running as long as the variable "i" is less then 10. Finally, the for-loop will increment variable i in every loop. Let us make a full program and test it.

```csharp
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            int i;
            for (i = 0; i < 10; i++)
            {
                Debug.Print("i= " + i.ToString());
            }
        }

    }
}
```

If we run the program above, we will see that it is printing i from 0 to 9 but not 10. But, we wanted it to run from 1 to 10 and not 0 to 9! To start from 1 and not 0, we need to set i to 1 in the initial loop. Also, to run to 10, we need to tell the for-loop to turn all the way to 10 and not less than 10 so we will change the less than ("<") with less than or equal ("<=")

```csharp
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            int i;
            for (i = 1; i <= 10; i++)
            {
                Debug.Print("i= " + i.ToString());
            }
        }
```

```
        }
}
```

Can we make the 'for' loop count only even numbers (increment by two)?

```
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            int i;
            for (i = 2; i <= 10; i = 1 + 2)
            {
                Debug.Print("i= " + i.ToString());
            }
        }
    }
}
```

The best way to understand for-loops is by stepping in code and seeing how C# will execute it.

16.6. Switch Statement

You will probably not use switch statement for beginner application but you will find it very useful when making large programs, especially when handling state-machines. The switch-statement will compare a variable to a list of constants (only constants) and make an appropriate jump accordingly. In this example, we will read the current "DayOfWeek" value and then from its value we will print the day as a string. We can do all this using if-statement but you can see how much easier switch-statement is, in this case.

```
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
    public class Program
    {
```

```csharp
        public static void Main()
        {
            DateTime currentTime = DateTime.Now;
            int day = (int)currentTime.DayOfWeek;
            switch (day)
            {
                case 0:
                    Debug.Print("Sunday");
                    break;
                case 1:
                    Debug.Print("Monday");
                    break;
                case 2:
                    Debug.Print("Tuesday");
                    break;
                case 3:
                    Debug.Print("Wednsday");
                    break;
                case 4:
                    Debug.Print("Thursday");
                    break;
                case 5:
                    Debug.Print("Friday");
                    break;
                case 6:
                    Debug.Print("Saturday");
                    break;
                default:
                    Debug.Print("We should never see this");
                    break;
            }
        }
    }
}
```

One important note about switch-statement is that it compares a variable to a list of constants. After every "case" we must have a constant and not a variable.

We can also change the code to switch on the enumeration of days as the following.

```csharp
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
```

```
        {
                DateTime currentTime = DateTime.Now;
                switch (currentTime.DayOfWeek)
                {
                    case DayOfWeek.Sunday:
                        Debug.Print("Sunday");
                        break;
                    case DayOfWeek.Monday:
                        Debug.Print("Monday");
                        break;
                    case DayOfWeek.Tuesday:
                        Debug.Print("Tuesday");
                        break;
                    case DayOfWeek.Wednesday:
                        Debug.Print("Wednsday");
                        break;
                    case DayOfWeek.Thursday:
                        Debug.Print("Thursday");
                        break;
                    case DayOfWeek.Friday:
                        Debug.Print("Friday");
                        break;
                    case DayOfWeek.Saturday:
                        Debug.Print("Saturday");
                        break;
                    default:
                        Debug.Print("We should never see this");
                        break;
                }
            }
        }
}
```

Try to step in the code to see how switch is handled in details.

17. Serial Interfaces

There are many serial interfaces available for data transfer between processors. Each interface has its advantages and disadvantages. I will try to cover them in enough details so you can use them with NETMF.

Even though there are many serial interfaces, when we say we talk about serial, we mean UART or RS232. Other interfaces, like CAN and SPI, still transmit its data serially but they are not serial ports!!

If further details are needed, consult the web, especially http://www.wikipedia.org/.

17.1. UART

UART is one of the oldest and most common interfaces. Data is sent out on a UART TXD pin in a sequence at a predefined speed. When the transmitter sends out zeros and ones, the receiver is checking for incoming data on RXD at the same speed that the transmitter is sending. The data is sent one byte at the time. This is one direction for data. To transfer the data in the opposite direction, a similar circuit is constructed at the opposite side. Transmit and receive are completely separate circuits and they can work together or independently. Each side can send data at any time and can also receive data at anytime.

The predefined speed of sending/receiving data is called baud rate. Baud rate is how many bits are transmitted in one second. Usually, one of the standard baud rates are used, like 9600, 119200, 115200 or others.

Through UART, two processors can connect directly by connecting TXD on one side to RXD on the other side and vice versa. Since this is just a digital IO on the processor, the voltage levels on UART TXD/RXD pins will be 0V (low) and 3.3V or 5V (high).

In industrial systems or when long wires are used, 3.3V or even 5V doesn't provide enough room for error. There are standards that define how we can take UART standard signal and convert it to higher voltages or differential signal to allow data transfer with better reliability. The most common one is RS232. Almost all computers are equipped with RS232 port. With RS232, data is simply UART but the voltage levels are converted from TTL (0V to 3.3V) to RS232 levels (-12V to +12V). One important fact about RS232 is that the voltage level is inverted from how we would logically think of it. When the signal is logically low, the voltage is at +12V and when the signal is logically high, the voltage is at -12V.There are many ready chips that convert TTL levels to RS232 levels, like MAX232 and MAX3232.

When we need to interface a processor using UART to a computer's serial port, we need to convert the UART TTL level to RS232 using some circuitry. For beginners or a quick hack, there are companies who provide ready circuits that convert RS232 to UART.

This is just an example:

Copyright © 2010 GHI Electronics, LLC   Page 76/170

http://www.nkcelectronics.com/rs232-to-ttl-converter-board-33v232335.html

In the PC world, USB is more popular than serial ports. Newer computers, especially laptops, do not have serial ports but every computer and laptop does have a USB interface. For that reason, many companies have created a USB to UART chipset. USB<->RS232 cable is a very common device available at any computer store. An interesting product from FTDI is a USB cable with UART interface, note that it is TTL UART not RS232 on the other side of USB, which means it can be connected directly to your processor's TTL UART. The product number for this cable is "TTL-232R-3V3".

To summarize all above, you can interface two processors by connecting UART pins directly. To interface a processor to a PC, you need to convert the UART signals to RS232 or USB using one of the ready circuits like MAX232 for RS232 and FT232 for USB.

NETMF supports serial ports (UART) in the same way the full .NET framework on the PC is supported. To use the Serial port, add "Microsoft.SPOT.Hardware.SerialPort" assembly and add "using System.IO.Ports" at the beginning of your code.

Note that serial ports on PC's and on NETMF are named COM ports and they start from COM1. There is no COM0 on computer systems. This can cause a little confusion when we want to map the COM port to the UART port on the processor because the processor usually start with UART0 not UART1. So, COM1 is UART0 and COM2 is UART1...etc.

PCs have terminal programs that open the serial ports to send/receive data. Any data received will be displayed on the terminal software and any data typed in the terminal will be sent out on the serial port. Teraterm is one of common and free software that we recommended.

Flowing is a program that sends a counter value every 10 times a second. The data is sent at 115200 so make sure the terminal is setup the same way. This program sends the data on COM1 of your NETMF device. This COM number has nothing to do with COM number on your PC. For example, you may have a USB serial port on your PC that maps to COM8 and so you need to open COM8 on your PC, not COM1 but the NETMF program will still use COM1 because it uses UART0 (COM1).

```csharp
using System.Threading;
using System.IO.Ports;
using System.Text;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            SerialPort UART = new SerialPort("COM1", 115200);
            int counter=0;
            UART.Open();
```

```
            while (true)
            {
                // create a string
                string counter_string = "Count: " + counter.ToString() +
                                        "\r\n";
                // convert the string to bytes
                byte[] buffer = Encoding.UTF8.GetBytes(counter_string);
                // send the bytes on the serial port
                UART.Write(buffer, 0, buffer.Length);
                // increment the counter;
                counter++;
                //wait...
                Thread.Sleep(100);
            }
        }
    }
}
```

Note how we ended the string with "\r\n". The "\r" is code to tel the terminal to "return" back to the beginning of the line and "\n" is to add "new" line.

When data is received on UART, it is automatically queued a side so you wouldn't lose any data. Note that there is limit on how much data the system can buffer so if you are debugging or not reading the data then close the serial port; otherwise, the system will become very slow and debugging/execution will be very unreliable and slow. Ideally, events will be used so we are automatically receiving data. We will cover events later.

This example will wait till byte is received then it prints back some string telling you what you have typed in (transmitted).

```
using System.Threading;
using System.IO.Ports;
using System.Text;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            SerialPort UART = new SerialPort("COM1", 115200);
            int read_count = 0;
            byte[] rx_byte = new byte[1];

            UART.Open();
            while (true)
            {

                // read one byte
```

```csharp
                    read_count = UART.Read(rx_byte, 0, 1);
                    if (read_count > 0)// do we have data?
                    {
                        // create a string
                        string counter_string =
                                "You typed: " + rx_byte[0].ToString() + "\r\n";
                        // convert the string to bytes
                        byte[] buffer = Encoding.UTF8.GetBytes(counter_string);
                        // send the bytes on the serial port
                        UART.Write(buffer, 0, buffer.Length);
                        //wait...
                        Thread.Sleep(10);
                    }
                }
            }
        }
}
```

This last example is a loop-back. Connect a wire from TX to RX on your board and it will send data and make sure it is receiving it correctly.

```csharp
using System.Threading;
using System.IO.Ports;
using System.Text;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            SerialPort UART = new SerialPort("COM1", 115200);
            int read_count = 0;
            byte[] tx_data;
            byte[] rx_data = new byte[10];
            tx_data = Encoding.UTF8.GetBytes("FEZ");
            UART.Open();

            while (true)
            {
                // flush all data
                UART.Flush();
                // send some data
                UART.Write(tx_data, 0, tx_data.Length);
                // wait to make sure data is transmitted
                Thread.Sleep(100);
                // read the data
                read_count = UART.Read(rx_data, 0, rx_data.Length);
```

```csharp
                    if (read_count != 3)
                    {
                        // we sent 3 so we should have 3 back
                        Debug.Print("Wrong size: " + read_count.ToString());
                    }
                    else
                    {
                        // the count is correct so check the values
                        // I am doing this the easy way so the code is more clear
                        if (tx_data[0] == rx_data[0])
                        {
                            if (tx_data[1] == rx_data[1])
                            {
                                if (tx_data[2] == rx_data[2])
                                {
                                    Debug.Print("Perfect data!");
                                }
                            }
                        }
                    }
                    Thread.Sleep(100);
                }
            }
        }
}
```

17.2. SPI

SPI uses three or four wires for transferring data. In UART, both sides need to have a predetermined baud rate. This is different on SPI where one of the nodes will send a clock to the other along with data. This clock is used to determine how fast the receiver needs to read the data. If you know electronics, this is a shift register. The clock is always transmitted from the master device. The other device is a slave and it doesn't send a clock but receives a clock from the master.

So, the master will transmit a clock on a SCK (serial clock) pin and will simultaneously transmit the data on MOSI (Master Out Slave In) pin. The slave will read the clock on SCK pin and simultaneously read the data from MOSI pin. So far, this is a one way communication. While data is transmitted in one direction using MOSI another set of data is sent back on MISO (Master In Slave Out) pin. This is all done simultaneously, clock send and receive. It is not possible to only send or only receive with SPI. You will always send a byte and receive a byte back in response. Other data sizes are possible but bytes are most common. NETMF supports 8-bit (byte) and 16-bit (short) data transfers.

Because of this master/slave scheme, we can add multiple slaves on the same bus where the master selects which slave it will swap the data with. Note I am using the word swap because you can never send or receive but you can send and receive (swap) data. The master selects

the slave using SSEL (Slave Select) pin. This pin can be called CS (Chip Select) as well. In theory, the master can have unlimited slaves but it can only select one of them at any given time. The master will only need 3 wires (SCK, MISO, MOSI) to connect to all slaves on the bus but then it needs a separate SSEL pin for each one of the slaves.

Some SPI devices (slaves) can have more than one select pin, like VS1053 MP3 decoder chip that uses one pin for data and one pin for commands but both share the 3 data transfer pins (SCK, MOSI, MISO).

SPI needs more wires than other similar buses but it can transfer data very fast. A 50Mhz clock is possible on SPI, that is 50 million bits in one second.

NETMF devices are always SPI masters. Before creating a SPI object, we would need a SPI configuration object. The configuration object is used to set the states of the SPI pins and some timing parameters. In most cases, you need the clock to be idle low (false) with clocking on rising edge (true) and with zero for select setup and hold time. The only thing you would need to set is the clock frequency. Some devices may accept high frequencies but others do not. Setting the clock to 1000Khz (1Mhz) should be okay for most getting started projects.

This example is sending/receiving (swapping, if you will) 10 bytes of data on SPI channel 1. Note that NETMF start numbering SPI channels (module) from 1 but on processors, the channels start from 0. So, using SPI1 in code is actually using SPI0 on the processor.

```csharp
using System.Threading;
using System.Text;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            SPI.Configuration MyConfig =
                new SPI.Configuration((Cpu.Pin)FEZmini.Pin.Di2x,
                false,0,0,false,true,1000,SPI.SPI_module.SPI1);
            SPI MySPI = new SPI(MyConfig);

            byte[] tx_data = new byte[10];
            byte[] rx_data = new byte[10];

            MySPI.WriteRead(tx_data, rx_data);

            Thread.Sleep(100);
        }
    }
}
```

17.3. I2C

I2C was developed by Phillips to allow multiple chipsets to communicate on a 2-wire bus in home consumer devices, mainly TV sets. Similar to SPI, I2C have a master and one or more slaves on the same data bus. Instead of selecting the slaves using a digital pin like SPI, I2C uses software addressing. Before data is transferred, the master sends out a 7-bit address of the slave device it want communicate with. It also sends a bit indicating that if the master wants to send or receive data. The slaves that see its address on the bus will acknowledge its presence. At this point, the master and send/receive data.

The master will start data transfers with "start " condition before it sends any address or data and then end it with "stop" condition.

I2C NETMF driver is based on transactions. If we want to read from a register on a sensor, we would need to send it the register number and then we need to read the register. Those are two transactions, write then read.

This example will send communicate with I2C device address 0x38. It will write two (register number) and read back the register value

```csharp
using System;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;

namespace Test
{
    public class Program
    {
        public static void Main()
        {
            //create I2C object
            I2CDevice.Configuration con =
                new I2CDevice.Configuration(0x38, 400);
            I2CDevice MyI2C = new I2CDevice(con);

            //create transactions (we need 2 in this example)
            I2CDevice.I2CTransaction[] xActions =
                new I2CDevice.I2CTransaction[2];

            // create write buffer (we need one byte)
            byte[] RegisterNum = new byte[1] { 2 };
            xActions[0] = MyI2C.CreateWriteTransaction(RegisterNum);
            // create read buffer to read the register
            byte[] RegisterValue = new byte[1];
            xActions[1] = MyI2C.CreateReadTransaction(RegisterValue);
```

```
                    // Now we access the I2C bus and timeout in one second
                    // if no responce
                    MyI2C.Execute(xActions, 1000);

                    Debug.Print("Register value: " + RegisterValue[0].ToString());
            }


        }
}
```

17.4. One Wire

GHI exclusively supports one wire devices on NETMF. Dallas semiconductor's one wire devices, like temperature sensors or EEPROMs, use only a single wire for data transfers. Multiple devices can be connected and controlled on a single wire. The one wire class, provide many methods to read and write bytes from one wire devices. It also includes the one wire CRC calculation method as well.

This example will read the temperature from DS18B20 one wire digital thermometer. Note that this is a GHI exclusive feature and so it requires adding the GHI assembly to the build.

```csharp
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.FEZ;

namespace Test
{
    public class Program
    {
        public static void Main()
        {
            // Change this your correct pin!
            Cpu.Pin myPin = (Cpu.Pin)FEZ_Pin.Digital.Di4;

            OneWire ow = new OneWire(myPin);
            ushort temperature;

            // read every second
            while (true)
            {
                if (ow.Reset())
                {
                    ow.WriteByte(0xCC);      // Skip ROM
```

```csharp
                ow.WriteByte(0x44);        // Start temperature conversion

                while (ow.ReadByte() == 0) ;   // wait while busy

                ow.Reset();
                ow.WriteByte(0xCC);      // skip ROM
                ow.WriteByte(0xBE);      // Read Scratchpad

                temperature = ow.ReadByte();                    // LSB
                temperature |= (ushort)(ow.ReadByte() << 8); // MSB

                Debug.Print("Temperature: " + temperature / 16);
                Thread.Sleep(1000);
            }
            else
            {
                Debug.Print("Device is not detected.");
            }

            Thread.Sleep(1000);
          }
        }
      }
}
```

17.5. CAN

Controller Area Network is a very common interface in industrial control and automotive. CAN is very robust and works very well in noisy environments at high speeds. All error checking and recovery methods are done automatically on the hardware. TD (Transmit Data) and RD (Receive Date) are the only two pins needed. These pins carry out the digital signal that need to be converted to analog before it is on the actual wires using the physical layer. Physical layers are sometimes called transceivers.

There are many kinds of physical layers but the most commonly used is high-speed-dual-wire that uses twisted pair for noise immunity. This transceiver can run at up to 1Mbps and can transfer data on very long wires if low bit-rate is used. Data can be transferred between nodes on the bus where any node can transfer at any time and all other nodes are required to successfully receive the data. There is no master/slave in CAN. Also, all nodes must have a predefined bit timing criteria. This is much more complicated that calculating a simple baud rate for UART. For this reason, many CAN bit rate calculators are available.

The CAN peripheral of Embedded Master is same as the popular SJA1000. A quick Internet search for SJA1000 should result in more than one free calculator.

In short, this is a simple way to calculate the bit timing:
   1. Divide the system clock (72Mhz) to get an appropriate clock for the CAN peripheral

(this is NOT the baud rate) This is called the BRP.
2. Figure out how many clocks you need to make up one bit. Usually this is 24 or 16. This is called TQ.
3. Assign values to T1 and T2 where T1+T2+1=TQ

Let us assume we need 250Kbps.
1. From 72Mhz system clock, I want the CAN clock to be 6Mhz so I need to divide by 12 (BRP=12).
2. 6Mhz/250kbps = 24 TQ (we usually want 16 or 24).
3. T1 = 15, T2 = 8 will give us 15 + 8 + 1 = 24 and this is what we need.

I got the T1 and T2 values from: http://www.kvaser.com/can/protocol/index.htm
I picked the first value and subtracted 1 from T1 because the calculator included the sync bit

GHI Electronics is updating the CAN driver for NETMF 4.0 and so the interface may change. Check the documentation for further help.

Here is the code with detailed comments

```csharp
using System.Threading;
using Microsoft.SPOT;
using System;
using GHIElectronics.Hardware;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            // These numbers were calculated using the
            // calculator on this link:
            // http://www.kvaser.com/can/protocol/index.htm
            // We used the very first value from the calculator output
            ////////////////////////////////////////////////////////
            // Bitrate 250Kbps
            // CLK = 72 Mhz, with BRP = 12 -> 6Mhz CAN clock
            // 6Mhz/250Kbps = 24 TQ
            // T1 = 16 minus 1 for sync = 15
            // T2 = 8
            // 15 + 1 + 8 = 24 TQs which is what we need
            ////////////////////////////////////////////////////////
            const int BRP = 12;
            const int T1 = 15;
            const int T2 = 8;
            // For 500Kbps you can use BRP=6 and for 1Mbps you can use BRP=3
            // and for 125Kbps use BRP=24...and so on
            // Keep T1 and T2 the same to keep the sampling pint
```

```csharp
            // the same (between 70% and 80%)

            // Initialize CAN channel, set bit rate
            CAN canChannel = new CAN(CAN.CANChannel.Channel_1,
                ((T2 - 1) << 20) | ((T1 - 1) << 16) | ((BRP - 1) << 0));
            // make new CAN message
            CAN.CANMessage message = new CAN.CANMessage();
            // make a message of 8 bytes
            for (int i = 0; i < 8; i++)
                message.data[i] = (byte)i;
            message.DLC = 8; // 8 bytes
            message.ArbID = 0xAB; // ID
            message.isEID = false; // not extended ID
            message.isRTR = false; // not remote
            // send the message
            canChannel.PostMessage(message);
            // wait for a message and get it.
            while (canChannel.GetRxQueueCount() == 0) ;
            // get the message using the same message object
            canChannel.GetMessage(message);
            // Now "message" contains the data, ID,
            // flags of the received message.
        }
    }
}
```

18. Output Compare

This exclusive GHI feature allows developers to generate all kind of digital signals on any of the digital IOs. For example, OutputCompare can be used to generate a UART signal. Or, generate a signal to control a 38Khz infrared transmitter to simulate a TV remote control.

One very good example is the driver of 2x16 character serial display found on www.TinyCLR.com. The display is controlled by UART. Also, the display doesn't send any data back. All you need is to send some simple control commands serially to the display. Now, we can just connect the display to one of the available serial ports. But then we will lose the port over something so simple plus the UART serial interface uses 2 pins, for transmit and for receive. But, the display doesn't send any data back so basically you will lose the receive pin. The right way to control this serial display is by using OutputCompare. You will only need one pin and you can use any of the digital pins.

So how does OutputCompare work? Basically, you provide the method with an array of time values between each pin toggle. The OutputCompare then goes through the array of time values and generate the signal on the requested pin. So if are going to generate UART, we first need to pre-calculate the values needed to represent the transmitted byte and then provide that to the OutputCompare object.

The example driver should explain how this is done. This is a copy of the driver found on www.TinyCLR.com for the serial LCD driver

```csharp
using System;
using System.Threading;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
namespace GHIElectronics.NETMF.FEZ
{
    public static partial class FEZ_Components
    {
        public class SerialLCD : IDisposable
        {
            const byte DISP_ON = 0xC;    // Turn visible LCD on
            const byte CLR_DISP = 0x01;  // Clear display
            const byte CUR_HOME = 2;     // Move cursor home
                                         // and clear screen memory
            const byte SET_CURSOR = 0x80;   // SET_CURSOR + X :
                                            // Sets cursor position to X
            const byte Move_CURSOR_LEFT = 0x10;

            OutputCompare oc;
            const int MAX_TIMINGS_BUFFER_SIZE = 10;
            uint[] buffer = new uint[MAX_TIMINGS_BUFFER_SIZE];
```

```csharp
                const int BAUD_RATE = 2400;
                const int BIT_TIME_US = 1 * 1000 * 1000 / BAUD_RATE;
                readonly int BYTE_TIME_MS;
                public void Dispose()
                {
                    oc.Dispose();
                    buffer = null;
                }
                private void SendByte(byte b)
                {
                    bool currentPinState;
                    int currentBufferIndex = 0;
                    uint currentStateTiming;
                    // start bit
                    currentPinState = false;
                    currentStateTiming = BIT_TIME_US;
                    // data bits
                    for (int i = 0; i < 8; i++)
                    {
                        bool neededState = (b & (1 << i)) != 0;

                        if (neededState != currentPinState)
                        {
                            buffer[currentBufferIndex] = currentStateTiming;
                            currentStateTiming = BIT_TIME_US;
                            currentPinState = neededState;
                            currentBufferIndex++;
                        }
                        else
                        {
                            currentStateTiming += BIT_TIME_US;
                        }
                    }
                    // stop bit
                    if (currentPinState != true)
                    {
                        buffer[currentBufferIndex] = currentStateTiming;
                        currentBufferIndex++;
                    }
                    oc.Set(false, buffer, 0, currentBufferIndex, false);
                    // wait till data is sent
                    Thread.Sleep(BYTE_TIME_MS);
                }
                public void PutC(char c)
                {
                    SendByte((byte)c);
                }

                private void SendCommand(byte cmd)
                {
                    SendByte(0xFE);
                    SendByte(cmd);
```

```csharp
            }
            public void Print(string str)
            {
                for (int i = 0; i < str.Length; i++)
                    PutC(str[i]);
            }
            public void ClearScreen()
            {
                SendCommand(CLR_DISP);
            }
            public void CursorHome()
            {
                SendCommand(CUR_HOME);
            }
            public void SetCursor(byte row, byte col)
            {
                SendCommand((byte)(SET_CURSOR | row << 6 | col));
            }
            public void MoveLeft()
            {
                SendCommand(Move_CURSOR_LEFT);
            }
            public SerialLCD(FEZ_Pin.Digital pin)
            {
                BYTE_TIME_MS = (int)Math.Ceiling((double)BIT_TIME_US *
                            MAX_TIMINGS_BUFFER_SIZE / 1000);
                oc = new OutputCompare((Cpu.Pin)pin, true,
                            MAX_TIMINGS_BUFFER_SIZE);

                // Initilaize LCD
                SendCommand(DISP_ON);
                SendCommand(CLR_DISP);
            }
        }
    }
}
```

19. Loading Resources

A resource is a file that is included with the application image. If our application depends on a file (image, icon, sound) then we may want to add this file to the application resources. Application can read the file from the file system but then the application becomes dependent on the file system to run.

Resources can also be strings, like a copyright text. This way, we need to update the copyright text, we would only need to update the resources and do nothing in the program code.

Always know how much memory is reserved for the application space. Adding large file will result deploying errors and VS2010 will not tell you it is because the files are too large.

Looking in the "Solution Explorer" window, we see the "Resource.resx" and if we expand it we see "Resources.Designer.cs"
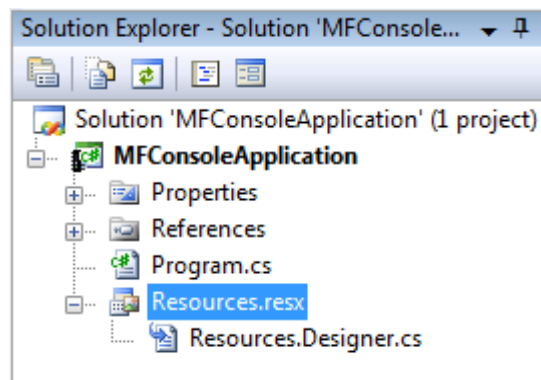
Visual Studio automatically generate the "resources designer" file. Never try to modify this file manually. Instead, double click "Resources.resx" to edit the resources using a tool.
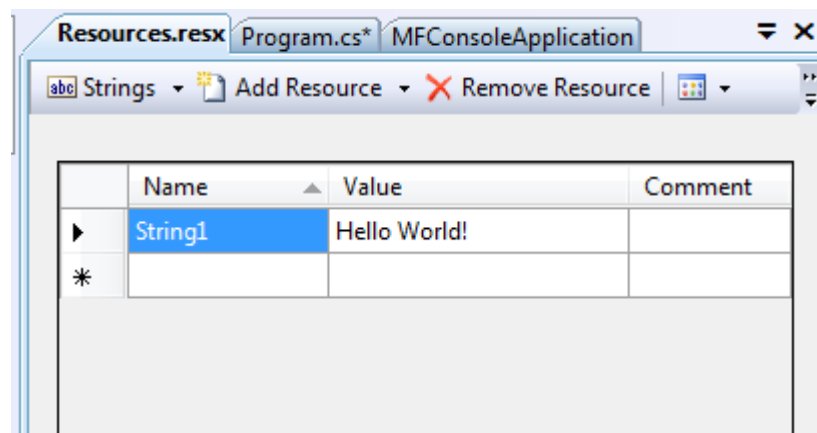
On the resource editor window, the first drop-down list on the left is to select what type of resources we want to edit. You notice how there is already one "string resource" named "String_1" With its "value" set to "Hello World!"

Click right below "String_1" and add a new string resource as shown in image below.
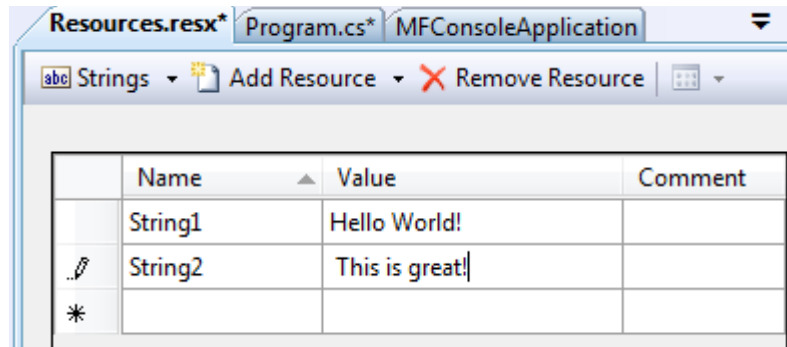
We now have two string resources. Let us use them

```
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(
                Resources.GetString(Resources.StringResources.String1));
            Debug.Print(
                Resources.GetString(Resources.StringResources.String2));
        }
    }
}
```

Try modifying the string resource text and observe how the output changes.

Let us add a file. Create a text file on your desktop and put some text in it. Select "files" from the left drop-down menu then clock on "Add Resource..."
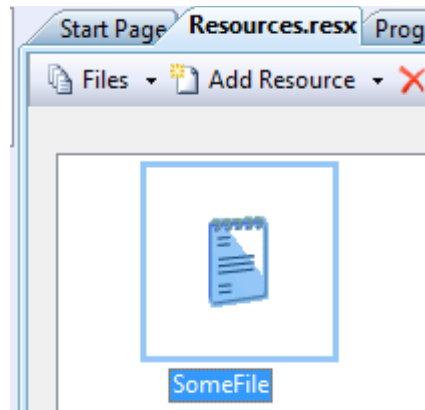
Since this is a text file, Visual Studio is smart enough to add this in a similar way to strings. So to use this resource file, we access it the same way we access a string.

```
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(
                Resources.GetString(Resources.StringResources.SomeFile));
        }
    }
}
```

In this example, I will add a midi file. Later in the book, we will learn how to add a decoder and play this file. First, find your favorite midi file. I did a quick web search and downloaded lambada.mid file.
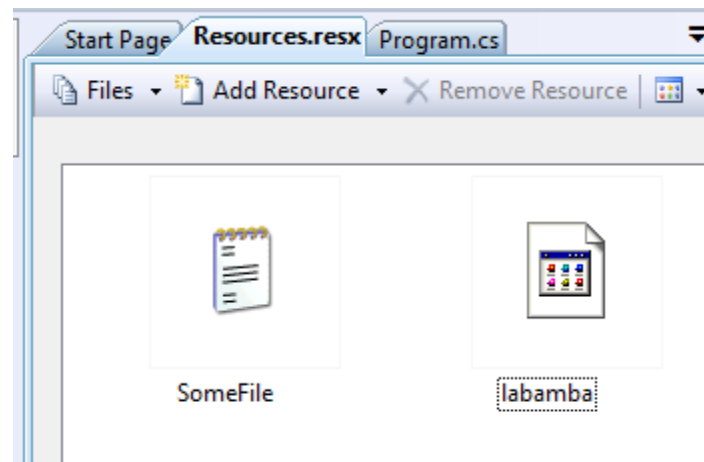
The file is about 33Kb. Which is small in the computer world but for embedded systems it is not small. The example below will work on devices with a lot of RAM, like ChipworkX and Embedded Master. On USBizi and FEZ, it may or may not work. We can still access large files with USBizi and FEZ but instead or reading the whole file at once, we only read some if it, send it to decoder, then we come back and read more. Details and instructions come later.

```csharp
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            byte[] file_data =
                Resources.GetBytes(Resources.BinaryResources.labamba);
            Debug.Print("File Size is " + file_data.Length.ToString());
        }
    }
}
```

20. Displays

20.1. Character Displays

Most character displays use the same interface. Those displays are mostly 2-lines by 16-characters, commonly known as 2x16 character displays. The display can be controlled using 8-bit or 4-bit interface. The 4-bit option is more favorable since it requires less IOs.

The interface uses RS (Data/Instruction), RW(Read/Write), E (Enable) and 4-bit data bus. The display manual is the best resource of information but this is a simple example class to get the display running quick.

GHI offers an alternative to this display. The SerialLCD display offered on www.TinyCLR.com will work on any single pin on FEZ. The included driver makes using these displays even easier.

Using the display driver

```
using GHIElectronics.NETMF.FEZ;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            FEZ_Components.SerialLCD LCD =
                new FEZ_Components.SerialLCD(FEZ_Pin.Digital.Di5);

            LCD.ClearScreen();
            LCD.CursorHome();
            LCD.Print("FEZ Rocks!");
        }
    }
}
```

20.2. Graphical Displays

Native support

NETMF, with its bitmap class, can handle graphics very well. The bitmap class supports

images from BMP, JPG and GIF file types. The images can be obtained from the file system or the network but an easier option is to include the image in a resource.

The bitmap object can be used to draw images or shapes and to draw text using a font. NETMF supports creating fonts using TFConvert tool. I will not cover the use of TFConvert so we will use one of the available fonts.

When we draw using a bitmap object, we are actually drawing on the object (on memory) nothing is visible on the screen. To transfer a bitmap object from memory to the screen, we need to "flush" the bitmap object. An important note here is that flush will only work if the size of the bitmap is exactly the same size of the screen.

If you have an image size 128x128 pixels and want to display it on the screen, we first need to create a new bitmap object with the size of the screen, then create a second bitmap object for the smaller image. Draw the smaller image on the large one and then flush! To eliminate confusion, I always have a one bitmap object called LCD and then everything gets drawn on this object.

We will run all these tests on the emulator instead of hardware as your hardware may not support native graphics.

```csharp
using System.Threading;
using Microsoft.SPOT;
using System;
using Microsoft.SPOT.Presentation;
using Microsoft.SPOT.Presentation.Media;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            Bitmap LCD = new Bitmap(SystemMetrics.ScreenWidth,
                                    SystemMetrics.ScreenHeight);

            //clears the memory and not the display
            LCD.Clear();
            //draw on memory
            LCD.DrawLine(Colors.Green, 1, 10, 10, 40, 40);
            //transfer the bitmap memory to the actual display
            LCD.Flush();
        }
    }
}
```

The code above requires Microsoft.SPOT.TinyCore assembly to run. Then we need to use the presentation namespace so we can get the "SystemMetrics".

Run the code and you will see a green line on the emulator.
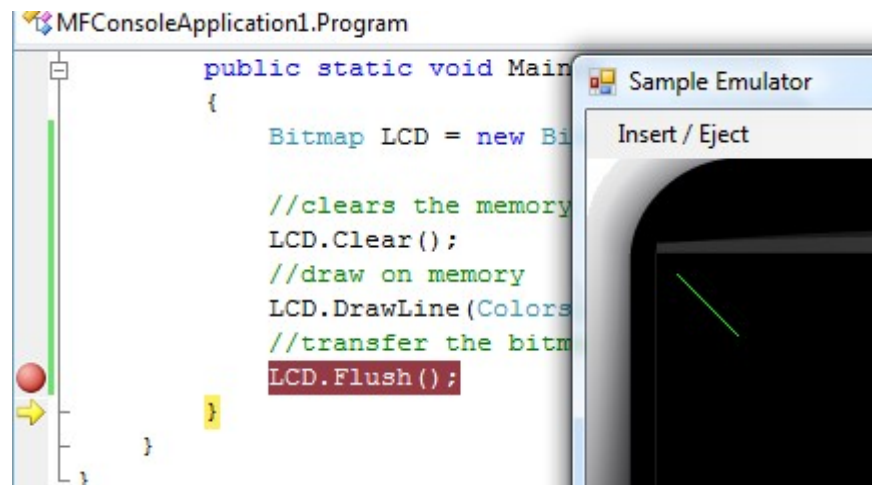
Try to use what we learned on for-loop to create multiple lines.

```csharp
using System.Threading;
using Microsoft.SPOT;
using System;
using Microsoft.SPOT.Presentation;
using Microsoft.SPOT.Presentation.Media;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            Bitmap LCD = new Bitmap(SystemMetrics.ScreenWidth,
                                    SystemMetrics.ScreenHeight);

            //clears the memory and not the display
            LCD.Clear();
            int i;
            for (i = 10; i < 200; i += 4)
            {
                //draw on memory
                LCD.DrawLine(Colors.Green, 1, 10, i, i, 200);
            }
            //transfer the bitmap memory to the actual display
            LCD.Flush();
        }
    }
}
```
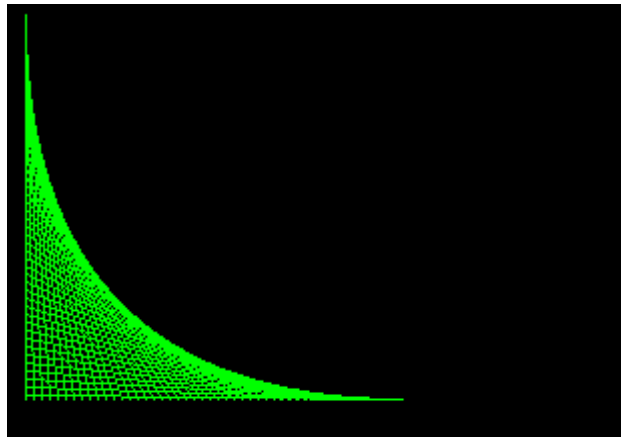
To draw text, we would need to have a font resource first. Add a new resource to the project. You can use one of the resource files coming with NETMF SDK examples.

The samples at ...\Documents\Microsoft .NET Micro Framework 3.0\Samples\

I used "NinaB.tinyfnt" font file. Add the file to your resources like explained in previous chapter. We can now run this code to print on the LCD.

```csharp
using System.Threading;
using Microsoft.SPOT;
using System;
using Microsoft.SPOT.Presentation;
using Microsoft.SPOT.Presentation.Media;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            Bitmap LCD = new Bitmap(SystemMetrics.ScreenWidth,
                                    SystemMetrics.ScreenHeight);
            Font MyFont = Resources.GetFont(Resources.FontResources.NinaB);

            //clears the memory and not the display
            LCD.Clear();
            int i;
            for (i = 10; i < 200; i += 4)
            {
                //draw on memory
                LCD.DrawLine(Colors.Green, 1, 10, i, i, 200);
            }
            // print some text on the screen
            LCD.DrawText("Still Amazing!", MyFont, Colors.Red, 100, 20);
            //transfer the bitmap memory to teh actual display
```

```
            LCD.Flush();
        }
    }
}
```

Non-native support

Many small graphical displays use SPI bus to receive images from the host. NETMF devices usually support large TFT displays that use a special bus to operate, like FEZ Cobra (EMX). But, even if the system doesn't support those displays, like FEZ Rhino (USBizi), user can connect an SPI-based display and display graphics this way. It is also possible to have two displays on system that support native TFT interface. A large display will run on the TFT interface and the smaller display will run on a SPI bus.

Even tough SPI is very fast, displays can have millions of pixels so we need to take this under consideration when selecting a display. Www.TinyCLR.com offers many displays, SPI or TFT that are ideal to work with GHI offers. Below is one of the example. Also check the standard SPI-based display for FEZ Rhino on the website mentioned earlier.
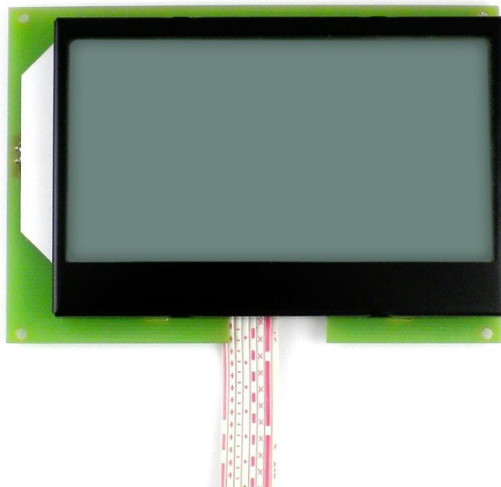
Native-support on non-standard display

A better option is to use the bitmap class to draw text, draw shapes and then transfer the bitmap to your display. You can only use this if Bitmap (graphics) is supported on your device, USBizi doesn't support graphics.

This code will display some data on the F-51852 128x64 pixel display found on the old non-TFT Embedded Master development system.

```
using System;
using System.Text;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Presentation;
using Microsoft.SPOT.Presentation.Media;
```

```csharp
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.System;

    public class Program
    {
        static SPI.Configuration conf = new SPI.Configuration((Cpu.Pin)33,
                false, 0, 0, false, true, 1000, SPI.SPI_module.SPI2);
        static SPI SPI_port = new SPI(conf);
        static OutputPort RST = new OutputPort((Cpu.Pin)9, true);
        static OutputPort DC = new OutputPort((Cpu.Pin)15, true);
        static byte[] _ba = new byte[1];
        static void OPTREX_SSD_WriteCmdByte(byte b)
        {
            DC.Write(false);

            Thread.Sleep(1);

            _ba[0] = b;
            SPI_port.Write(_ba);
        }

        static void OPTREX_SSD_WriteByte(byte b)
        {
            DC.Write(true);

            Thread.Sleep(1);

            _ba[0] = b;
            SPI_port.Write(_ba);
        }
        static void OPTREX_Locate(int x, int y)
        {
            if (y > 7)
                y = 7;

            if (x > 127)
                x = 127;

            OPTREX_SSD_WriteCmdByte((byte)(0X10 | (x >> 4)));//col up
            OPTREX_SSD_WriteCmdByte((byte)(0X00 | (x & 0xF)));//col down

            OPTREX_SSD_WriteCmdByte((byte)(0XB0 | y));//page addr

        }
        public static void Main()
        {

            OPTREX_SSD_WriteCmdByte(0XA2);//bias
            OPTREX_SSD_WriteCmdByte(0XA1);//adc    inverse
            OPTREX_SSD_WriteCmdByte(0Xc0);//common dir...normal
            OPTREX_SSD_WriteCmdByte(0X40);//initial line
            OPTREX_SSD_WriteCmdByte(0X81);//evr set
```

```csharp
            OPTREX_SSD_WriteCmdByte(0X20);
            OPTREX_SSD_WriteCmdByte(0X29);//2B we have -10V........
                                        //wait for stable voltage
        Thread.Sleep(10);
            OPTREX_SSD_WriteCmdByte(0XA4);//turn all on
            OPTREX_SSD_WriteCmdByte(0XE7);//driver
            OPTREX_SSD_WriteCmdByte(0XAF);//lcd on

            //OPTREX_SSD_WriteCmdByte(0XA7);//inverse
            OPTREX_SSD_WriteCmdByte(0XA6);//no inverse


            OPTREX_SSD_WriteCmdByte(0XB0);//page addr
            OPTREX_SSD_WriteCmdByte(0X10);//col
            OPTREX_SSD_WriteCmdByte(0X00);//col

        int x = 20;
        int y = 50;
        int dx = -2;
        int dy = -3;

        Bitmap bb = new Bitmap(128, 64);
        byte[] bitmapbytes;
        Font fnt = Resources.GetFont(Resources.FontResources.small);
        byte[] vram = new byte[128 * 64 / 8];
        byte[] singleline = new byte[128];
        while (true)
        {
            bb.Clear();
            bb.SetPixel(0, 0, Color.White);
            bb.SetPixel(0, 2, Color.White);
            bb.SetPixel(2, 0, Color.White);
            bb.SetPixel(2, 2, Color.White);
            bb.DrawText("Rocks!", fnt, Color.White, 20, 45);
            bb.DrawEllipse(Color.White, x, y, 5, 3);

            x += dx;
            y += dy;
            if (x < 0 || x > 128)
                dx = -dx;
            if (y < 0 || y > 64)
                dy = -dy;

            bitmapbytes = bb.GetBitmap();
            Util.BitmapConvertBPP(bitmapbytes, vram,
                            Util.BPP_Type.BPP1_x128);

            for (int l = 0; l < 8; l++)
            {
                OPTREX_Locate(0, l);
                DC.Write(true);
                Array.Copy(vram, l * 128, singleline, 0, 128);
```

```
                    SPI_port.Write(singleline);
                }
                Thread.Sleep(1);
            }
        }
    }
```

21. Time Services

In computer systems, time can mean two things. The system time, which is the processor ticks, is used to handle threads timing and all timing management in the system. On the other hand, the real time clock is used to time human time, like minutes, hours and even dates.

21.1. Real Time Clock

All GHI NETMF devices have a built in RTC. The RTC job is to keep track of time and date. Even if the system is off, the RTC will be running off a backup battery. Keeping RTC running while system is off requires a 32.768Khz oscillator and a 3V coin battery on a specific pin,usually called VBAT. All GHI boards and modules except FEZ Mini already have 32.768Khz crystal on-board. Only 3V disc battery is needed.

It is important to understand that the time services in NETMF is not directly connected to RTC. For example, you can set the "NETMF system time" and from that point on, time will be accurately available. This doesn't mean the RTC hardware has the correct time. Also, a user maybe reading time from a time server over the network and wanting to use that instead of RTC.

Here is an example that sets the time first then print the time every 100ms. This sets the NETMF time and not RTC hardware. Loosing power will cause the clock to reset to a random value.

```csharp
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using System;

namespace MFConsoleApplication1
{
    public class Program
    {
        public static void Main()
        {
            // set the time to 9/9/2010 at 9:09:09
            DateTime time = new DateTime(2010, 9, 9, 9, 9, 9);
            Utility.SetLocalTime(time);
            while (true)
            {
                Debug.Print(DateTime.Now.ToString());
                Thread.Sleep(100);
            }
        }
    }
}
```

```
        }
    }
}
```

To use the RTC hardware, we first need to check if the RTC hardware has the valid time or not. Maybe this is a new battery or the new system and the RTC has not been set yet. If RTC has a valid time then we can read the RTC (hardware) and use that to set the NETMF system time (software). If time is not valid, then you will need to set the RTC to the correct time.

```csharp
using System;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;

    public class Program
    {
        public static void Main()
        {
            // To keep track of time,
            // set it at the beginning of your application from the RTC.
            // If it was NOT set before and currently running using
            // the battery (not exhausted), set it to a fixed time.
            if (RealTimeClock.IsTimeValid == false)
                RealTimeClock.SetTime(new DateTime(2010, 03, 01, 12, 0, 0, 0));

            Utility.SetLocalTime(RealTimeClock.GetTime());
        }
    }
```

21.2. Timers

Micro Framework includes 2 timer classes, Timer and ExtendedTimes. Timer class is the same one included in full framework where ExtendedTimer is specific to NETMF with extra functionality. For basic beginner projects, I suggest you keep on using threading before getting into timers. I will only provide an example in this book. Our example will create a timer that will run after five seconds and then it will keep firing every second.

```csharp
using System.Threading;
using Microsoft.SPOT;
using System;

namespace MFConsoleApplication1
{
    public class Program
    {
```

```csharp
        static void RunMe(object o)
        {
            Debug.Print("From timer!");
        }
        public static void Main()
        {
            Timer MyTimer =
                new Timer(new TimerCallback(RunMe), null, 5000, 1000);
            Debug.Print(
                    "The timer will fire in 5 seconds and then fire
                     priodically every 1 second");
            Thread.Sleep(Timeout.Infinite);
        }
    }
}
```

22. USB Host

There is almost always confusion between USB host and USB device. The USB host is the system that connects to multiple USB devices. For example, the PC is a USB host and it can connect to multiple USB devices like mice, keyboards and mass storage devices. Implementing a USB device is rather simple but implementing a host is far more complicated.

USB host is an exclusive feature from GHI Electronics. With this exclusive feature, you can connect almost any USB device to GHI's NETMF products (USBizi, Embedded Master, ChipworkX). This feature opens new possibilities for embedded systems. Your product can now connect to a standard USB keyboard and can also access files on a USB thumb drive!

USB is a hot pluggable system which means any device can be connected or disconnected any time. There are events generated when devices are connected or disconnected. The developer should subscribe to these events and handle devices accordingly. Since this is a beginner book, I will assume that the device will always be connected to the system.

With USB HUB support, devices can be connected directly to the USB host port or a user may connect multiple USB devices through a USB hub.

First, let us detect what devices are connected. The first thing to do is start the system manager then we can get a list of available devices. Remember that we need to add the GHI library assembly to our resources.

```csharp
using System;
using System.Threading;
using Microsoft.SPOT;

using GHIElectronics.NETMF.USBHost;

namespace Test
{
  class Program
  {
    public static void Main()
    {
      // Subscribe to USBH events.
      USBHostController.DeviceConnectedEvent += DeviceConnectedEvent;
      USBHostController.DeviceDisconnectedEvent += DeviceDisconnectedEvent;

      // Sleep forever
      Thread.Sleep(Timeout.Infinite);
    }

    static void DeviceConnectedEvent(USBH_Device device)
    {
```

```
          Debug.Print("Device connected...");
          Debug.Print("ID: " + device.ID + ", Interface: " +
                    device.INTERFACE_INDEX + ", Type: " + device.TYPE);
      }

      static void DeviceDisconnectedEvent(USBH_Device device)
      {
         Debug.Print("Device disconnected...");
           Debug.Print("ID: " + device.ID + ", Interface: " +
                      device.INTERFACE_INDEX + ", Type: " + device.TYPE);
       }
    }
}
```

When we detect a device, we can communicate with it directly. This requires a lot of knowledge on USB devices. Fortunately, most devices fall under standard classes and GHI already provide drivers for them.

22.1. HID Devices

Human Interface Devices like mice, keyboards and joysticks are directly supported. Using HID is event based. Events are methods you create and then you subscribe them to a certain event. When that event fires, your method will get executed automatically.

```
using System;
using System.Threading;
using Microsoft.SPOT;
using GHIElectronics.NETMF.USBHost;

namespace Test
{
    public class Program
    {
        static USBH_Mouse mouse;
        public static void Main()
        {
            // Subscribe to USBH event.
            USBHostController.DeviceConnectedEvent += DeviceConnectedEvent;

            // Sleep forever
            Thread.Sleep(Timeout.Infinite);
        }

        static void DeviceConnectedEvent(USBH_Device device)
        {
            if (device.TYPE == USBH_DeviceType.Mouse)
            {
```

```
                Debug.Print("Mouse Connected");
                mouse = new USBH_Mouse(device);
                mouse.MouseMove += MouseMove;
                mouse.MouseDown += MouseDown;
            }
        }

        static void MouseMove(USBH_Mouse sender, USBH_MouseEventArgs args)
        {
            Debug.Print("(x, y) = (" + sender.Cursor.X + ", " +
                        sender.Cursor.Y + ")");
        }

        static void MouseDown(USBH_Mouse sender, USBH_MouseEventArgs args)
        {
            Debug.Print("Button down number: " + args.ChangedButton);
        }
    }
}
```

Accessing Joysticks is very similar. Here is the example modified to work with joysticks

```
using System;
using System.Threading;
using Microsoft.SPOT;
using GHIElectronics.NETMF.USBHost;
namespace Test
{
    public class Program
    {
        static USBH_Joystick j;
        public static void Main()
        {
            // Subscribe to USBH event.
            USBHostController.DeviceConnectedEvent += DeviceConnectedEvent;

            // Sleep forever
            Thread.Sleep(Timeout.Infinite);
        }

        static void DeviceConnectedEvent(USBH_Device device)
        {
            if (device.TYPE == USBH_DeviceType.Joystick)
            {
                Debug.Print("Joystick Connected");
                j = new USBH_Joystick(device);
                j.JoystickXYMove += JoystickXYMove;
                j.JoystickButtonDown += JoystickButtonDown;
            }
```

```
        }

        static void JoystickButtonDown(USBH_Joystick sender,
                                       USBH_JoystickEventArgs args)
        {
            Debug.Print("Button Pressed: " + args.ChangedButton);
        }

        static void JoystickXYMove(USBH_Joystick sender,
                                   USBH_JoystickEventArgs args)
        {
            Debug.Print("(x, y) = (" + sender.Cursor.X + ", " +
                        sender.Cursor.Y + ")");
        }
    }
}
```

22.2. Serial Devices

Serial (UART) communication is a very common interface. There are many companies that create chips that convert USB to serial. GHI supports chipsets from FTDI, Silabs and Prolific.

Also,there is a standard USB class defined for serial communication called CDC (Communication Device Class). This class is supported as well.

Note here that the USB chipsets are made to be somewhat customized. So a company can use a FTDI chip to make their product run on USB and they will change the strings in USB descriptors so when you plug in their device to a PC you will see the company name not FTDI name. They can also change the USB VID/PID, vendor ID and product ID. A good example is a USB GPS device. Almost all those USB GPS devices use prolific chip, which is supported by GHI. Many of the interface products on the market use FTDI chipset.

```
using System;
using System.Text;
using System.Threading;

using Microsoft.SPOT;

using GHIElectronics.NETMF.USBHost;

namespace Test
{
    class Program
    {
        static USBH_SerialUSB serialUSB;
        static Thread serialUSBThread; // Prints data every second
```

```csharp
 public static void Main()
{
    // Subscribe to USBH event.
    USBHostController.DeviceConnectedEvent += DeviceConnectedEvent;

    // Sleep forever
    Thread.Sleep(Timeout.Infinite);
}

static void DeviceConnectedEvent(USBH_Device device)
{
    Debug.Print("Device connected");

    switch (device.TYPE)
    {
        case USBH_DeviceType.Serial_FTDI: // FTDI connected
            serialUSB = new USBH_SerialUSB(device, 9600,
                System.IO.Ports.Parity.None, 8,
                System.IO.Ports.StopBits.One);
            serialUSB.Open();
            serialUSBThread = new Thread(SerialUSBThread);
            serialUSBThread.Start();

            break;

        case USBH_DeviceType.Unknown: // SiLabs but not recognized
            // force SiLabs
            USBH_Device silabs = new USBH_Device(device.ID,
                device.INTERFACE_INDEX,
                USBH_DeviceType.Serial_SiLabs, device.VENDOR_ID,
                device.PRODUCT_ID, device.PORT_NUMBER);
            serialUSB = new USBH_SerialUSB(silabs, 9600,
                System.IO.Ports.Parity.None, 8,
                System.IO.Ports.StopBits.One);
            serialUSB.Open();
            serialUSBThread = new Thread(SerialUSBThread);
            serialUSBThread.Start();

            break;
    }
}

static void SerialUSBThread()
{
    // Print "Hello World!" every second.
    byte[] data = Encoding.UTF8.GetBytes("Hello World!\r\n");
    while (true)
    {
        Thread.Sleep(1000);

        serialUSB.Write(data, 0, data.Length);
    }
```

```
            }
        }
}
```

22.3. Mass Storage

Storage devices like USB hard drives and USB thumb memory drives use the same USB class, MSC (Mass Storage Class). GHI library directly support those devices. USB only defines how to read/write raw sectors on the media. The operating system then handles the file system. NETMF supports FAT32 and FAT16 file system. To access the files on a USB media, we first need to detect it then we need to mount the media.

```csharp
using System;
using System.IO;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.IO;
using GHIElectronics.NETMF.IO;
using GHIElectronics.NETMF.USBHost;

namespace Test
{
    class Program
    {
        public static void Main()
        {
            // Subscribe to RemovableMedia events
            RemovableMedia.Insert += RemovableMedia_Insert;
            RemovableMedia.Eject += RemovableMedia_Eject;

            // Subscribe to USB events
            USBHostController.DeviceConnectedEvent += DeviceConnectedEvent;

            // Sleep forever
            Thread.Sleep(Timeout.Infinite);
        }

        static void DeviceConnectedEvent(USBH_Device device)
        {
            if (device.TYPE == USBH_DeviceType.MassStorage)
            {
                Debug.Print("USB Mass Storage detected...");
                //....
                //....
            }
        }
    }
```

```
}
```

The next section explains how to access the files on the USB memory.

23. File System

File system is supported in NETMF 3.0. GHI adds more functionality to the standard support. For example, a SD card can be mounted to the file system or mounted to the USB device MSC service. When mounted to the filer system, developers can access the files. But, when mounted to the USB device MSC, a PC connected to the USB port will see a USB card reader with SD card. This is good for creating data logger for example. The device will log data on the SD card and when the device is plugged to a PC, the device will become a card reader for the same SD memory card. GHI's persistent storage class is used to handle mounting devices on the system.

This section will only cover using persistent storage device with internal file system.

23.1. SD Cards

First we need to detect the SD card insertion. SD card connectors usually have a little switch internally that closes when a card is inserted. In this example, I will assume the card is always inserted and there is no need to look for detection. The example will list all files available in the root directory.

```csharp
using System;
using System.IO;
using System.Threading;

using Microsoft.SPOT;
using Microsoft.SPOT.IO;

using GHIElectronics.NETMF.IO;

namespace Test
{
    class Program
    {
        public static void Main()
        {
            // ...
            // SD Card is inserted
            // Create a new storage device
            PersistentStorage sdPS = new PersistentStorage("SD");

            // Mount the file system
            sdPS.MountFileSystem();
```

```
                    // Assume one storage device is available, access it through
                    // Micro Framework and display available files and folders:
                    Debug.Print("Getting files and folders:");
                    if (VolumeInfo.GetVolumes()[0].IsFormatted)
                    {
                        string rootDirectory =
                            VolumeInfo.GetVolumes()[0].RootDirectory;
                        string[] files = Directory.GetFiles(rootDirectory);
                        string[] folders = Directory.GetDirectories(rootDirectory);

                        Debug.Print("Files available on " + rootDirectory + ":");
                        for (int i = 0; i < files.Length; i++)
                            Debug.Print(files[i]);

                        Debug.Print("Folders available on " + rootDirectory + ":");
                        for (int i = 0; i < folders.Length; i++)
                            Debug.Print(folders[i]);
                    }
                    else
                    {
                        Debug.Print("Storage is not formatted. Format on PC with
                                    FAT32/FAT16 first.");
                    }

                    // Unmount
                    sdPS.UnmountFileSystem();
                }
            }
        }
```

There is more than one way to open files. I will only cover FileStream objects. This example will open a file and write a string to it. Since FileStream will only take byte arrays, we need to convert our string to byte array.

```
using System.Threading;
using System.Text;
using Microsoft.SPOT;
using System.IO;
using Microsoft.SPOT.IO;
using GHIElectronics.NETMF.IO;

namespace MFConsoleApplication1
{
    public class Program
    {
        static void Main()
        {
            // ... check if SD is inserted
```

```
            // SD Card is inserted
            // Create a new storage device
            PersistentStorage sdPS = new PersistentStorage("SD");

            // Mount the file system
            sdPS.MountFileSystem();

            // Assume one storage device is available,
            // access it through NETMF
            string rootDirectory = VolumeInfo.GetVolumes()[0].RootDirectory;
            FileStream FileHandle = new FileStream(rootDirectory +
                                        @"\hello.txt", FileMode.Create);
            byte[] data =
                Encoding.UTF8.GetBytes("This string will go in the file!");
            // write the data and close the file
            FileHandle.Write(data, 0, data.Length);
            FileHandle.Close();

            // if we need to unmount
            sdPS.UnmountFileSystem();

            // ...
            Thread.Sleep(Timeout.Infinite);

        }
    }
}
```

Verify the file if it is on the card using a PC and memory card reader if you like. Now, we want to open the same file and read the string we stored earlier.

```
using System.Threading;
using System.Text;
using Microsoft.SPOT;
using System.IO;
using Microsoft.SPOT.IO;
using GHIElectronics.NETMF.IO;

namespace MFConsoleApplication1
{
    public class Program
    {
        static void Main()
        {
            // ... check if SD is inserted

            // SD Card is inserted
            // Create a new storage device
            PersistentStorage sdPS = new PersistentStorage("SD");
```

```
            // Mount the file system
            sdPS.MountFileSystem();

            // Assume one storage device is available,
            // access it through NETMF
            string rootDirectory = VolumeInfo.GetVolumes()[0].RootDirectory;
            FileStream FileHandle = new FileStream(rootDirectory +
                    @"\hello.txt", FileMode.Open, FileAccess.Read);
            byte[] data = new byte[100];
            // write the data and close the file
            int read_count = FileHandle.Read(data, 0, data.Length);
            FileHandle.Close();
            Debug.Print("The size of data we read is: " +
                    read_count.ToString());
            Debug.Print("Data from file:");
            Debug.Print(data.ToString());

            // if we need to unmount
            sdPS.UnmountFileSystem();

            // ...
            Thread.Sleep(Timeout.Infinite);

        }
    }
}
```

23.2. USB Mass Storage

Files are handled on USB exactly the same way it is done on SD. The only difference is in how we detect a USB device and mount it. For SD, we could use an input pin to detect the card. On USB, we use events to detect a new media.

```
using System;
using System.IO;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.IO;
using GHIElectronics.NETMF.IO;
using GHIElectronics.NETMF.USBHost;

namespace Test
{
    class Program
    {
```

```csharp
        // Hold a static reference in case the GC kicks in and disposes it
        // automatically, note that we only support one in this example!
        static PersistentStorage ps;

        public static void Main()
        {
            // Subscribe to RemovableMedia events
            RemovableMedia.Insert += RemovableMedia_Insert;
            RemovableMedia.Eject += RemovableMedia_Eject;

            // Subscribe to USB events
            USBHostController.DeviceConnectedEvent += DeviceConnectedEvent;

            // Sleep forever
            Thread.Sleep(Timeout.Infinite);
        }

        static void DeviceConnectedEvent(USBH_Device device)
        {
            if (device.TYPE == USBH_DeviceType.MassStorage)
            {
                Debug.Print("USB Mass Storage detected...");
                ps = new PersistentStorage(device);
                ps.MountFileSystem();
            }
        }

        static void RemovableMedia_Insert(object sender, MediaEventArgs e)
        {
            Debug.Print("Storage \"" + e.Volume.RootDirectory +
                        "\" is inserted.");
            Debug.Print("Getting files and folders:");
            if (e.Volume.IsFormatted)
            {
                string[] files = Directory.GetFiles(e.Volume.RootDirectory);
                string[] folders =
                        Directory.GetDirectories(e.Volume.RootDirectory);

                Debug.Print("Files available on " +
                            e.Volume.RootDirectory + ":");
                for (int i = 0; i < files.Length; i++)
                    Debug.Print(files[i]);

                Debug.Print("Folders available on " +
                            e.Volume.RootDirectory + ":");
                for (int i = 0; i < folders.Length; i++)
                    Debug.Print(folders[i]);
            }
            else
            {
                Debug.Print("Storage is not formatted. Format on PC with
                            FAT32/FAT16 first.");
```

```
            }
        }

        static void RemovableMedia_Eject(object sender, MediaEventArgs e)
        {
            Debug.Print("Storage \"" + e.Volume.RootDirectory +
                        "\" is ejected.");
        }
    }
}
```

You can see from the code above how after we mount the USB drive to the file system, everything work exactly the same as SD cards.

23.3. File System Considerations

NETMF support for FAT File System is only capable of FAT32 and FAT16. A media formatted as FAT12 will not work. This shouldn't be an issue since FAT12 is no longer in use.

The file system does a lot of data buffering internally to speed up the file access time and to increase the life of the flash media. When you write data to a file, it is not necessary that the data is written on the card. It is probably saved somewhere in the internal buffers. To make sure the data is stored on the media, we need to "flush" the data. Flushing or closing a file is the only way to guarantee that the data you are trying to write are now on the actual media.

That is on file level. On media level, there are also information that may not take immediate effect. For example, if you delete a file and remove the card from the system, the file is probably not actually erased. To guarantee the file is erased (media is updated) you need to run VolumeInfo.FlushALL

Ideally, you would unmounted the media **before** it is removed from the system. This may not be always possible and so a flush on a file or a FlushAll on media will guarantee your data is saved so there is no lost data if the media was removed at some point.

24. Networking

Networks are an essential part of our work and living. Almost every home is connected to a network (internet) and most businesses can't function without an internal network (LAN or WiFi) that is connected to an external network (internet). All these networks have a standard way for communication, they all use TCP/IP protocol. There are actually a few protocols that handle different tasks in the network DNS, DHCP, IP, ICMP, TCP, UDP, PPP...and many more!

NETMF supports TCP/IP networks through standard .NET sockets. A socket is a virtual connection between 2 devices on a network.

GHI extended the TCP/IP support to cover PPP and WiFi. Through PPP, two devices can connect through serial connection. Serial connection can be a phone-line modem or a 3G/GPRS modem. With PPP, NETMF devices can connect to the internet using 3G cell-phone networks. It is also possible to connect two NETMF devices through wired or wireless (XBee/Bluetooth/others) serial connection.

Also, with WiFi support, NETMF devices can connect to standard secure or unsecured wireless networks.

NETMF also support SSL for secure connection.

The support for networks is standard and complete (HTTP, SSL, Sockets...etc.) on EMX, Embedded Master and for ChipworkX.

24.1. USBizi (FEZ) Network Support

When it comes to ram-hungry networking, USBizi will require some external support. For example, USBizi wired networking is done through Wiznet W5100 Hardwired TCP/IP chip. When USBizi want to make a network connection, all it has to do is send a request to Wiznet 5100 of the connection an then the chip will do the rest. Same as for data transfer, where USBizi will hand the data to W5100 and then this chip will take care of the rest.

The Wiznet W5100 works over SPI bus. Current driver is provided by GHI to handle W5100 right from C#. This is a beta driver and it is for testing purposes only. There is development that is being done to provide W5100 driver right into the USBizi firmware. This is provided by GHI for convenience to its customers and to optimize the W5100 driver to work more efficiently. Until GHI driver is available, anyone with some experience can implement a W5100 driver to fit their needs.

24.2. Raw TCP/IP vs. Sockets

This is an area when most designers miss few important points. When selecting a WiFi or GPRS/3G module, there are two main module categories to select from. Modules with sockets and modules with raw TCP/IP. The modules with sockets handle all TCP/IP work internally and give you a high level socket access. This means that all the work is being done already inside the module. Once you set your IP and port number, all you have to do is send and receive data from the module. The catch is, this is very limited. The module will have many limitations, like socket count. Even if your system is very powerful, with megabytes of RAM, you still is limited by the module's features. For this reason, using these high level modules is ideal for small systems.

Let me give you an example, Roving Networks provides a module called WiFly. This module has a built in TCP/IP stack and only one socket support. Using the module is very easy as you only need to connect it to one of the serial ports on your system then, with simple serial commands, you can read/write data from the one available socket. This is enough to make a simple web server or telnet connection, for configuration or data transfer. This is perfect for small systems with limited memory and low resources, like USBizi (FEZ). The module does everything for you, just send and receive data serially.

If you are implementing a web server that provides some values like temperature and humidity then this is all you need and can be easily implemented using USBizi (FEZ) at a very low cost. An easy prototype is done by connecting the WiFly shield to FEZ Domino. The SparkFun WiFly shield is showing in the image to the right.
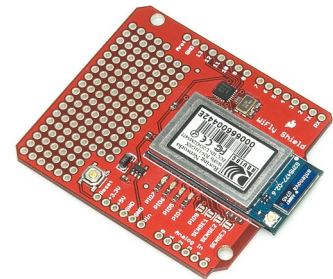
The negative side about using these simple modules is that you are limited very few sockets, very few connections. What if you need more? What if you want to implement SSL (secure sockets)? For those, you will need a WiFi module that doesn't handle TCP/IP internally. The TCP/IP work will need to be done outside the module. This is where devices like EMX and ChipworkX come in. These devices are powerful, with a lot of resources. They have built in TCP/IP stack with SSL/HTTP/DHCP...etc. So connecting module like ZeroG to EMX or ChipworkX will empower the device with full blown and secure WiFi connection.

What about GPRS and 3G modems? The same applies to these modules. Some have built in sockets like SM5100B but others work over PPP just like any PC modem, like Telit modules for example. If you need a real network connection with full blown TCP/IP stack then you need EMX or ChipworkX with a standard PPP modems, just like how would your PC connect to the internet using a standard modem.

If you need a simple and low-cost connection then USBizi (FEZ) can be used with SM5100B. The SparkFun Cellular Shield showing on the right, plugs right into FEZ Domino.

24.3. Standard .NET Sockets

The socket supported on NETMF is very similar to the full .NET Framework. The NETMF SDK includes many examples for using sockets, client and server. Also, many projects are available showing the different possibilities

Twitter client:

http://www.microframeworkprojects.com/index.php?title=MFTwitter

Google maps:

http://www.microframeworkprojects.com/index.php?title=GoogleMaps

RSS client:

http://www.microframeworkprojects.com/index.php?title=RSS_n_Weather

MP3 internet radio:

http://www.microframeworkprojects.com/index.php?title=ShoutcastClient

Web server:

http://www.microframeworkprojects.com/index.php?title=WebServer

24.4. Wi-Fi (802.11)

WiFi is the most common way for computer networking. It allows secure data transfers at high rates to multiple connections. WiFi allows connections between two nodes, called Ad-Hoc. The more common way is to connect multiple computers to an access point. WiFi is not simple and not designed to be embedded friendly. GHI Electronics is the only company that offers WiFi option for its NETMF devices.

WiFi is designed to work with TCP/IP stacks (network sockets on NETMF) and so it can only be used with systems that already support TCP/IP, like EMX and ChipworkX.

GHI's WiFi support uses ZeroG's ZG2100 and ZG2101 that use internal or external antennas respectively. For prototyping, the WiFi expansion board is a good start.
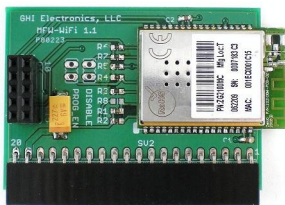
As explained in earlier section, USBizi (FEZ) can be used with modules with built in TCP/IP stack and socket support like the SparkFun shield with WiFly modules from Roving Networks.
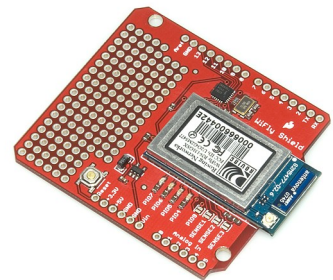
24.5. GPRS and 3G Mobile Networks

EMX and ChipworkX have built in TCP/IP stack and PPP support. You can connect any standard modem and use it with few simple steps.

As far as USBizi, a connection can be made to a mobile network using modems with built in TCP/IP like SM5100B. SparkFun Cellular shield with SM5100B is shown below.
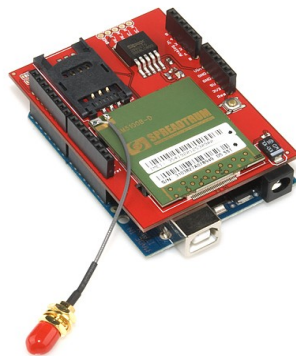
25. Cryptography

Cryptography has been an important part of technology for long years. Modern cryptography algorithms can be very resource hungry. Since NETMF is made with little devices in mind, the NETMF team had to be careful selecting what algorithms to support. The algorithms are XTEA and RSA.

25.1. XTEA

XTEA, with its 16byte (128bit) key, is considered to be very secure and at the same time it doesn't require a lot of processing power. Originally, XTEA was designed to work on chunks of eight bytes only. This can be a problem if encrypting data that it size is not multiple of 8. The implementation of XTEA on NETMF allows for encrypting data of any size.

Encrypting and decrypting data is straight forward. Here is an example:

```csharp
using System;
using System.Text;
using Microsoft.SPOT;
using Microsoft.SPOT.Cryptography;
public class Program
{
    public static void Main()
    {
        // 16-byte 128-bit key
        byte[] XTEA_key = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8,
                                       9, 10, 11, 12, 13, 14, 15, 16 };
        Key_TinyEncryptionAlgorithm xtea =
                new Key_TinyEncryptionAlgorithm(XTEA_key);
        // The data we want to encrypt
        string original_string = "FEZ is so easy!";
        //must be more than 8 bytes

        //convert to byte array
        byte[] original_data = UTF8Encoding.UTF8.GetBytes(original_string);

        //Encrypt the data
        byte[] encrypted_bytes = xtea.Encrypt(original_data, 0,
                                        original_data.Length, null);
        //Decrypt the data
        byte[] decrypted_bytes = xtea.Decrypt(encrypted_bytes, 0,
                                        encrypted_bytes.Length, null);
        //print the decrypted data
        string decrypted_string =
            new string(Encoding.UTF8.GetChars(decrypted_bytes));
        Debug.Print(original_string);
```

```
            Debug.Print(decrypted_string);
        }
}
```

The encrypted data has the same size as the unencrypted data.

XTEA on PCs

Now you can share data between NETMF devices securely using XTEA but what about sharing data with a PC or other systems? The book "Expert .NET Micro Framework", Second Edition by Jens Kuhner includes source examples showing how to implement XTEA on a PC. Using Jen's code, you can encrypt data then send to any NETMF device or vice-versa. Even if you do not own the book, the source code is online, found in chapter 8.

Download the source code from here http://apress.com/book/view/9781590599730

25.2. RSA

XTEA is very secure but has an important limitation, a key must be shared. For example, in order for any two systems to share encrypted data, they first both must share the same key. If a system tried to send the key to the other system then anyone spying on the data can get the key and use it to decrypt the data. Not very secure anymore!

RSA overcomes this problem by providing a private and public key combination. This may not make sense but the key used for encryption can't be used to decrypt the data. A different key is needed to decrypt the data. Lets say system 'A' needs to read some secure data from system 'B'. The first thing system 'A' will do is send a public key to system 'B'. System 'B' will now encrypt the data using the public key and send the encrypted data to the PC. A hacker can see the encrypted data and can see the public key but without the private key, decrypting the data is near impossible. Finally, system 'A' can decrypt the data with its private key.

By the way, this is how secure websites work.

NETMF devices can't generate keys. The keys are generated on a PC using a tool called MetaDataProcessor. Open the command prompt and enter this command

cd "C:\Program Files (x86)\Microsoft .NET Micro Framework\v4.0\Tools"

Note that you may need to change this folder depending on where you have installed .NET Micro Framework.

Generate the keys using the following:

MetadataProcessor.exe -create_key_pair c:\private.bin c:\public.bin

The keys are in binary but the tool can convert the keys to readable text using these commands

MetadataProcessor.exe -dump_key c:\public.bin >> c:\public.txt

MetadataProcessor.exe -dump_key c:\private.bin >> c:\private.txt

Now, copy the key to our example program.

Note that the public key always starts with 1,0,1 and the rest are zeros. We can use this info to optimize our code as showing.

```csharp
using System;
using System.Text;
using Microsoft.SPOT;
using Microsoft.SPOT.Cryptography;
public class Program
{
    public static void Main()
    {
        //this is shared between public and private keys
        byte[] module = new byte[] { 0x17, 0xe5, 0x27, 0x40, 0xa9, 0x15,
0xbd, 0xfa, 0xac, 0x45, 0xb1, 0xb8, 0xe1, 0x7d, 0xf7, 0x8b, 0x6c, 0xbd, 0xd5,
0xe3, 0xf4, 0x08, 0x83, 0xde, 0xd6, 0x4b, 0xd0, 0x6c, 0x76, 0x65, 0x17, 0x52,
0xaf, 0x1c, 0x50, 0xff, 0x10, 0xe8, 0x4f, 0x4f, 0x96, 0x99, 0x5e, 0x24, 0x4c,
0xfd, 0x60, 0xbe, 0x00, 0xdc, 0x07, 0x50, 0x6d, 0xfd, 0xcb, 0x70, 0x9c, 0xe6,
0xb1, 0xaf, 0xdb, 0xca, 0x7e, 0x91, 0x36, 0xd4, 0x2b, 0xf9, 0x51, 0x6c, 0x33,
0xcf, 0xbf, 0xdd, 0x69, 0xfc, 0x49, 0x87, 0x3e, 0x1f, 0x76, 0x20, 0x53, 0xc6,
0x2e, 0x37, 0xfa, 0x83, 0x3d, 0xf0, 0xdc, 0x16, 0x3f, 0x16, 0xe8, 0x0e, 0xa4,
0xcf, 0xcf, 0x2f, 0x77, 0x6c, 0x1b, 0xe1, 0x88, 0xbd, 0x32, 0xbf, 0x95, 0x2f,
0x86, 0xbb, 0xf9, 0xb4, 0x42, 0xcd, 0xae, 0x0b, 0x92, 0x6a, 0x74, 0xa0, 0xaf,
0x5a, 0xf9, 0xb3, 0x75, 0xa3 };
        //the private key...for dycrypting
        byte[] private_key = new byte[] { 0xb9, 0x1c, 0x24, 0xca, 0xc8, 0xe8,
0x3d, 0x35, 0x60, 0xfc, 0x76, 0xb5, 0x71, 0x49, 0xa5, 0x0e, 0xdd, 0xc8, 0x6b,
0x34, 0x23, 0x94, 0x78, 0x65, 0x48, 0x5a, 0x54, 0x71, 0xd4, 0x1a, 0x35, 0x20,
0x00, 0xc6, 0x0c, 0x04, 0x7e, 0xf0, 0x34, 0x8f, 0x66, 0x7f, 0x8a, 0x29, 0x02,
0x5e, 0xe5, 0x39, 0x60, 0x15, 0x01, 0x58, 0x2b, 0xc0, 0x92, 0xcd, 0x41, 0x75,
0x1b, 0x33, 0x49, 0x78, 0x20, 0x51, 0x19, 0x3b, 0x26, 0xaf, 0x98, 0xa5, 0x4d,
0x14, 0xe7, 0x2f, 0x95, 0x36, 0xd4, 0x0a, 0x3b, 0xcf, 0x95, 0x25, 0xbb, 0x23,
0x43, 0x8f, 0x99, 0xed, 0xb8, 0x35, 0xe4, 0x86, 0x52, 0x95, 0x3a, 0xf5, 0x36,
0xba, 0x48, 0x3c, 0x35, 0x93, 0xac, 0xa8, 0xb0, 0xba, 0xb7, 0x93, 0xf2, 0xfd,
0x7b, 0xfa, 0xa5, 0x72, 0x57, 0x45, 0xc8, 0x45, 0xe7, 0x96, 0x55, 0xf9, 0x56,
0x4f, 0x1a, 0xea, 0x8f, 0x55 };
        // the public key, always starts with 0x01, 0x00, 0x01,...
        // and the reast are all zeros
        byte[] public_key = new byte[128];
        public_key[0] = public_key[2] = 1;

        Key_RSA rsa_encrypt = new Key_RSA(module, public_key);
        Key_RSA rsa_decrypt = new Key_RSA(module, private_key);

        // The data we want to encrypt
        string original_string = "FEZ is so easy!";
```

```csharp
        //convert to byte array
        byte[] original_data = UTF8Encoding.UTF8.GetBytes(original_string);
        //Encrypt the data
        byte[] encrypted_bytes =
            rsa_encrypt.Encrypt(original_data, 0, original_data.Length, null);
        //Decrypt the data
        byte[] decrypted_bytes =
            rsa_decrypt.Decrypt(encrypted_bytes, 0, encrypted_bytes.Length,
                            null);
        //print the decrypted data
        string decrypted_string =
            new string(Encoding.UTF8.GetChars(decrypted_bytes));
        Debug.Print("Data Size= " + original_string.Length +
                    " Data= " + original_string);
        Debug.Print("Encrypted Data size= " + encrypted_bytes.Length +
                    " Decrypted Data= " + decrypted_string);
    }
}
```

RSA encrypted data size is not the same as the raw data size. Keep this in mind when planning on transferring or saving RSA encrypted data.

RSA is far more processor intensive than XTEA which can be a problem for small systems. My suggestion is to start an RSA session to exchange XTEA key and some security info and then switch to XTEA.

26. XML

26.1. XML in Theory

Extensible Markup Language (XML) is a standard for containing electronic data. When you want to transfer some info between two devices, you can set some rules on how the data is to be packed and sent from device A. On the other side, device B receives the data and knows how to unpack it. This created some difficulties in the past before XML. What if you were sending the data to a system implemented by different designer? You will have to explain the other designer how you have packed your data so he/she can unpack it. Now, designers can select to use XML to pack and unpack the data.

XML is extensively used daily in many ways. For example, when a website's shopping cart wants to know how much will be the shipping cost on a certain package, you will need to pack your shipment details in XML format and then send to FedEx (for example). Then FedEx website will read the info and send the cost info back in XML format as well.

The usefulness of XML can also be utilized in other ways. Let's say you are designing a data logger. Let's also assume the end users will need to configure the data logger to fit their needs. When a user configures the device, you need to store the info internally somehow. You can save the data with your own format which requires extra coding and debugging, or better just use XML.

All GHI Electronics NETMF devices have built in XML reader and writer (packer and un-packer).

Here is an example XML file that will help in our data logger design.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<NETMF_DataLogger>
  <FileName>Data</FileName>
  <FileExt>txt</FileExt>
  <SampleFreq>10</SampleFreq>
</NETMF_DataLogger>
```

The previous XML example includes a root element and three child elements. I chose for the file to look that way but you can, for example, make all info to be root elements. XML is very flexible, sometimes too flexible actually! Back to our example, the root element "NETMF_DataLogger" contains three pieces of info that are important for our logger. It contains the file name, the file extension and a frequency of our saved data. With this

example, the logger will create a file called Data.txt and then will log data into that file 10 times every second.

Other important use for us "Embedded developers" is sharing data with the big system, I mean your PC. Since PCs with all operating systems do support XML in a way or another, you can send/receive data from the PC using XML.

Spaces and layout do not mean anything to XML, we (humans) need them to make things easier to read. The previous example can be stored without the spaces and layout like this.

```xml
<?xml version="1.0" encoding="utf-8" ?><NETMF_DataLogger>
<FileName>Data</FileName>
<FileExt>txt</FileExt><SampleFreq>10</SampleFreq></NETMF_DataLogger>
```

See why spaces are important to us human being!

You can also add comments inside XML files, comments do not mean anything to XML but can help in manual reading of the files

```xml
<?xml version="1.0" encoding="utf-8" ?>
<!--This is just a comment-->
<NETMF_DataLogger>
   <FileName>Data</FileName>
   <FileExt>txt</FileExt>
   <SampleFreq>10</SampleFreq>
</NETMF_DataLogger>
```

Finally, XML support attributes. An attribute is an extra info given to an element. But why do you need an attribute if you can add another element to describe this extra information? You really do not need to use attributes and I would say if you do not have a good reason then just do not use them. I will no be explaining attributes in this book.

26.2. Creating XML

GHI Electronics' NETMF devices support reading and writing XML format. Reading and writing XML files work over streams which means any stream you already have or implement can work with XML. For example, we will use the built in MemoryStream and FileStream but you can create your own stream as well, which is not covers in this book.

This code shows how to make an XML document in memory. The code represent the our earlier XML example

```
using System.IO;
using System.Xml;
using System.Ext.Xml;
using Microsoft.SPOT;

public class Program
{
    public static void Main()
    {
        MemoryStream ms = new MemoryStream();

        XmlWriter xmlwrite = XmlWriter.Create(ms);

        xmlwrite.WriteProcessingInstruction("xml",
                "version=\"1.0\" encoding=\"utf-8\"");
        xmlwrite.WriteComment("This is just a comment");
        xmlwrite.WriteStartElement("NETMF_DataLogger");//root element
        xmlwrite.WriteStartElement("FileName");//child element
        xmlwrite.WriteString("Data");
        xmlwrite.WriteEndElement();
        xmlwrite.WriteStartElement("FileExt");
        xmlwrite.WriteString("txt");
        xmlwrite.WriteEndElement();
        xmlwrite.WriteStartElement("SampleFeq");
        xmlwrite.WriteString("10");
        xmlwrite.WriteEndElement();
        xmlwrite.WriteEndElement();//end the root element

        xmlwrite.Flush();
        xmlwrite.Close();
        //////// display the XML data ///////////
        byte[] byteArray = ms.ToArray();
        char[] cc = System.Text.UTF8Encoding.UTF8.GetChars(byteArray);
        string str = new string(cc);
        Debug.Print(str);
    }
}
```

Important note: On NETMF, XML writer and XML reader libraries are available from two different libraries. The XML reader comes from assembly "System.Xml" but the XML writer comes from "MFDpwsExtensions"! If you want to know why then you need to check with Microsoft! Also, the XML reader is in the "System.Xml" namespace but the XML writer is in "System.Ext.Xml". To make life easier, just include "System.Xml" and MFDpwsExtensions" assemblies whenever you need to get started with XML. Also your code should include the two needed namespaces just like I did in previous example.

Note: When you try to add an assembly you will notice that there are two assemblies for XML, the "System.Xml" and "System.Xml.Legacy". Never use the "legacy" driver, it is slow and needs a lot of memory. It is there for older systems that doesn't have a built in support for XML. All GHI Electronics; NETMF devices have a built in XML support (very fast!) and so you

should always use "System.Xml"

When running the example above, we will see the output XML data at the end. The data is correct but it is not formatted to be "human" friendly. Note that we are reading and writing XML files on a very small system so the less info (spaces/formatting) the better it is. So it is actually better not to have any extra spaces or formatting but for the sake of making things look pretty, we will add new lines as follows

```csharp
using System.IO;
using System.Xml;
using System.Ext.Xml;
using Microsoft.SPOT;

public class Program
{
    public static void Main()
    {
        MemoryStream ms = new MemoryStream();

        XmlWriter xmlwrite = XmlWriter.Create(ms);

        xmlwrite.WriteProcessingInstruction("xml",
            "version=\"1.0\" encoding=\"utf-8\"");
        xmlwrite.WriteComment("This is just a comment");
        xmlwrite.WriteRaw("\r\n");
        xmlwrite.WriteStartElement("NETMF_DataLogger");//root element
        xmlwrite.WriteString("\r\n\t");
        xmlwrite.WriteStartElement("FileName");//child element
        xmlwrite.WriteString("Data");
        xmlwrite.WriteEndElement();
        xmlwrite.WriteRaw("\r\n\t");
        xmlwrite.WriteStartElement("FileExt");
        xmlwrite.WriteString("txt");
        xmlwrite.WriteEndElement();
        xmlwrite.WriteRaw("\r\n\t");
        xmlwrite.WriteStartElement("SampleFeq");
        xmlwrite.WriteString("10");
        xmlwrite.WriteEndElement();
        xmlwrite.WriteRaw("\r\n");
        xmlwrite.WriteEndElement();//end the root element

        xmlwrite.Flush();
        xmlwrite.Close();

        //////// display the XML data ///////////
        byte[] byteArray = ms.ToArray();
        char[] cc = System.Text.UTF8Encoding.UTF8.GetChars(byteArray);
        string str = new string(cc);
```

```
            Debug.Print(str);
        }
}
```

26.3. Reading XML

Creating XML files is actually easier than parsing (reading) them. There are many ways to read the XML file but basically you can just go through the file and read one piece at the time till you reach the end. This code example creates an XML data and it reads it back.

```csharp
using System.IO;
using System.Xml;
using System.Ext.Xml;
using Microsoft.SPOT;

public class Program
{
    public static void Main()
    {
        MemoryStream ms = new MemoryStream();

        XmlWriter xmlwrite = XmlWriter.Create(ms);

        xmlwrite.WriteProcessingInstruction("xml",
            "version=\"1.0\" encoding=\"utf-8\"");
        xmlwrite.WriteComment("This is just a comment");
        xmlwrite.WriteRaw("\r\n");
        xmlwrite.WriteStartElement("NETMF_DataLogger");//root element
        xmlwrite.WriteString("\r\n\t");
        xmlwrite.WriteStartElement("FileName");//child element
        xmlwrite.WriteString("Data");
        xmlwrite.WriteEndElement();
        xmlwrite.WriteRaw("\r\n\t");
        xmlwrite.WriteStartElement("FileExt");
        xmlwrite.WriteString("txt");
        xmlwrite.WriteEndElement();
        xmlwrite.WriteRaw("\r\n\t");
        xmlwrite.WriteStartElement("SampleFeq");
        xmlwrite.WriteString("10");
        xmlwrite.WriteEndElement();
        xmlwrite.WriteRaw("\r\n");
        xmlwrite.WriteEndElement();//end the root element

        xmlwrite.Flush();
        xmlwrite.Close();

        //////// display the XML data ///////////
        byte[] byteArray = ms.ToArray();
        char[] cc = System.Text.UTF8Encoding.UTF8.GetChars(byteArray);
```

```csharp
        string str = new string(cc);
        Debug.Print(str);

        ///////////read xml
        MemoryStream rms = new MemoryStream(byteArray);

        XmlReaderSettings ss = new XmlReaderSettings();
        ss.IgnoreWhitespace = true;
        ss.IgnoreComments = false;
        //XmlException.XmlExceptionErrorCode.
        XmlReader xmlr = XmlReader.Create(rms,ss);
        while (!xmlr.EOF)
        {
            xmlr.Read();
            switch (xmlr.NodeType)
            {
                case XmlNodeType.Element:
                    Debug.Print("element: " + xmlr.Name);
                    break;
                case XmlNodeType.Text:
                    Debug.Print("text: " + xmlr.Value);
                    break;
                case XmlNodeType.XmlDeclaration:
                    Debug.Print("decl: " + xmlr.Name + ", " + xmlr.Value);
                    break;
                case XmlNodeType.Comment:
                    Debug.Print("comment " +xmlr.Value);
                    break;
                case XmlNodeType.EndElement:
                    Debug.Print("end element");
                    break;
                case XmlNodeType.Whitespace:
                    Debug.Print("white space");
                    break;
                case XmlNodeType.None:
                    Debug.Print("none");
                    break;
                default:
                    Debug.Print(xmlr.NodeType.ToString());
                    break;
            }
        }
    }
}
```

27. Expanding I/Os

An application may require more digital pins or more analog pins than what is available on the processor. There are ways to expand what is available.

27.1. Digital

The easiest way to expand digital pins is by using a shift register. The shift register will then connect to the SPI bus. Using SPI, we can send or get the state of its pins. Shift registers can be connected in series so in theory we can have unlimited digital pins.

Shift registers usually have eight digital pins. If we connect three of them to a device over SPI, we will have 24 new digital pin but we only use the SPI pins on the processor.

Another good example is the IO40 board that runs on I2C bus. Also, this board can be chained for a maximum of 320 IOs!
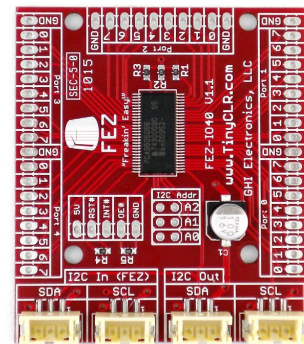
Button Matrix

Devices like microwave ovens have many buttons on the front. A user will never need to press two buttons at the same time so we can "matrix" those buttons. If we have 12 buttons on our system then we will need 12 digital inputs from the processor to read them all. Connecting these buttons in a 4x3 matrix will still give us 12 buttons but we are only using seven pins from the processor instead of 12. There are many off-the-shelf button matrix that can be integrated in your product.

To connect buttons in a matrix, we will wire our circuit so there are rows and columns. Each button will connect to one row and one column. That is all for hardware! Note how if we are not using a matrix then the button will connect to an input pin and ground.

To read the buttons state, make all processor pins connecting to rows outputs and the ones connecting to columns inputs. Set one and only one of the rows high and the rest of all rows to low. We are now selecting what row of buttons we will read. Now, read the state of all buttons in that row (you are reading the columns now). When complete, set the one row back to low and then go to the next one and set it high then go back to read the columns. Keep repeating until every row have been set to high once.

This example assumes we have a matrix with these connections. three rows connected to pins (1, 2, 3) and three columns connected to pins (4, 5, 6)

```csharp
using System.Threading;
using System;
using Microsoft.SPOT.Hardware;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
    {
        static OutputPort[] Rows = new OutputPort[3];
        static InputPort[] Colms = new InputPort[3];

        static bool ReadMatrix(int row, int column)
        {
            bool col_state;

            //select a row
            Rows[row].Write(true);
            //read the column
            col_state = Colms[column].Read();
            // deselectt the row
            Rows[row].Write(false);

            return col_state;
        }

        static void Main()
        {
            // initialize rows to outputs and low
            Rows[0] = new OutputPort((Cpu.Pin)1, false);
            Rows[1] = new OutputPort((Cpu.Pin)2, false);
            Rows[2] = new OutputPort((Cpu.Pin)3, false);

            //initialize inputs with pull down
            Colms[0] = new InputPort((Cpu.Pin)4, true,
                                Port.ResistorMode.PullDown);
            Colms[1] = new InputPort((Cpu.Pin)5, true,
                                Port.ResistorMode.PullDown);
            Colms[2] = new InputPort((Cpu.Pin)6, true,
                                Port.ResistorMode.PullDown);

            while (true)
            {
                bool state;

                // read the button on the first row and first column
                state = ReadMatrix(0, 0);//we count from zero
                Debug.Print("Buton state is: " + state.ToString());

                // read the button on the third row and second column
                state = ReadMatrix(2, 1);//we count from zero
                Debug.Print("Buton state is: " + state.ToString());
```

```
                Thread.Sleep(100);
            }
        }
    }
}
```

27.2. Analog

There are hundreds or thousands of analog chipsets available that run on SPI, I2C, one wire...etc. Some read 0V to 5V and other read -10V to +10V. Basically, there are too many options for reading more analog inputs to your device.

Some chips have specific tasks. If we need to measure temperature, we can connect a temperature sensor to an analog pin and then read the analog value and convert that to temperature. That is an option but a better option will be to use a digital temperature sensor that run on I2C, one wire or SPI. This will give us a more accurate temperature reading and will also save an analog input for other uses.

Analog Buttons

One trick to connect many buttons using a single pin is by using an analog pin. The buttons will be connected to resistors. This allows each button to set out a different voltage if pressed. That voltage can then be measured to determine which button has been pressed.

28. USB Client

I want to start this section by pointing out that this is a little advanced topic. If you just started out with NETMF then you may want to save it for later.

28.1. Serial (COM) Debugging

By default, all GHI's NETMF devices use USB for deploying and debugging. Optionally, a developer may want to use the USB client (not the host) for something other than debugging. This is actually supported by NETMF and GHI adds much functionality making it very easy to setup.

Important note: You can't use the USB client port for both debugging and something else. Once you decide you want to use the USB client port for your own use then you no longer can use the USB port for debugging. By the way, GHI did support debugging and customizing the USB interface simultaneously through using multiple interfaces on the same USB interface but this caused a huge load of support due to improper use and so simultaneous use support was dropped.

Let me give you an example on the USB client usage. Let us say you are making a device that that reads temperature and humidity...etc. and logs all this data on an SD card. Also, this device can be configured, like to set the time or give file names...etc. You want the device to be configured over USB. So when your device plugs into a USB port, you want it to show as a virtual serial port. This way, anyone can open a terminal software (like TeraTerm) to connect to your device to configure it. This is where USB client becomes very useful. There is no need to add extra cost to the design but adding additional RS232 serial ports or USB<->serial chipsets. The built-in USB client port can be configured to act as a CDC device, virtual COM port. But, there is a catch! The catch is, you still need to connect the PC to your device serially for debugging and deploying applications since the USB client port is used by your end application. The good news is that you only need the serial interface in the development stage but, when you deploy the product, the serial port is no longer needed. For example, you may use the RS232 shield on FEZ Domino in development stage but then you do not need it when you are done debugging.

Now, you have COM1 connected to your device and want to use that for debugging instead of USB. The next step is to configure the device to use serial for debugging. This is a device specific so you need to check your device's user manual. For example, if you are using FEZ
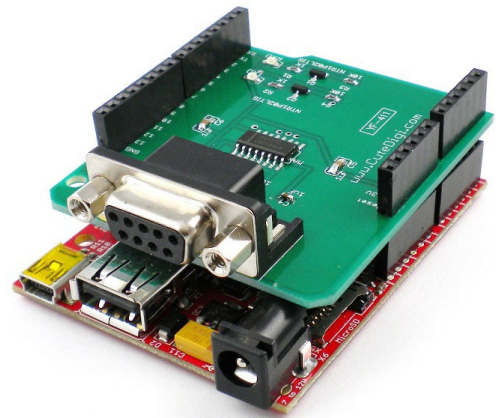
Domino, there is jumper labeled MODE that you can place to select serial debugging. If you are using FEZ Cobra then the pin LMODE need to be connected to 3.3V to select serial debugging. Remember that once you connect the pin high or low then you can no longer use it in your application. For example, the MODE pin on FEZ Domino is the same pin used for LED and it is also a PWM pin. Once you short the pin to ground (through the jumper), you should never attempt to use this pin, never try to use the LED or you may damage your device!

28.2. The Setup

Okay, enough talking and let us set up. I am using FEZ Domino with RS232 shield but you can use any device of your choice. I have the RS232 shield plugged into FEZ Domino and also connected to my PC using an RS232<->USB cable (no serial port on my PC). I also did place the MODE jumper and connected the USB cable from domino to this PC as well. After, placing the MODE jumper, connecting the USB cable from FEZ Domino to a PC will not load any drivers on windows (you will not hear the "USB connect" sound).

Finally, we want to make sure we can ping the device using MFDeploy tool. We did that before using USB and now we want to use serial. Note that even though I have FEZ Domino+RS232 shield connected to my PC's USB port through the RS232<->USB cable, this is COM and not USB, it is a virtual COM to be exact.

So, open MFDeploy and select COM and then you will have a list of the available serial ports. Select the one connected to your device and click ping. You should see "TinyCLR" back. If you don't then go back and check your setup.
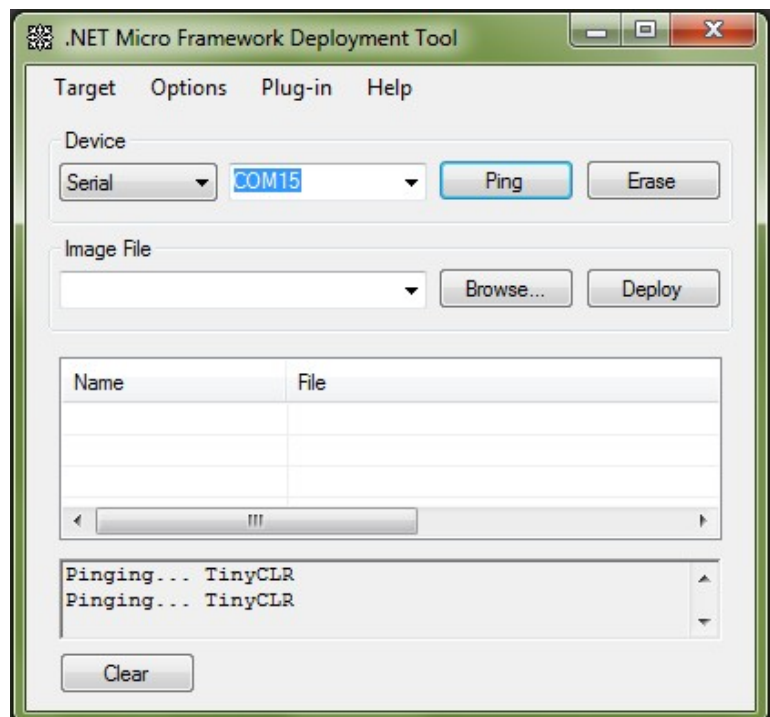
28.3. Mouse, the Perfect Prank

Here is the master plan! You want to prank someone, get it video taped and then send me the funny video. This is how you do it. Setup your FEZ to emulate a USB mouse then make the mouse move in a circle every few minutes. I bet you it will feel like there is a ghost on your machine. The good news is that windows can have multiple mouse devices so FEZ will be hidden in the back.

```csharp
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.USBClient;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.FEZ;

public class Program
{
    public static void Main()
    {
        FEZ_Components.LED led = new FEZ_Components.LED(FEZ_Pin.Digital.Di3);
        led.StartBlinking(100, 200);


        // Check debug interface
        if (Configuration.DebugInterface.GetCurrent() ==
Configuration.DebugInterface.Port.USB1)
            throw new InvalidOperationException("Current debug interface is
USB. It must be changed to something else before proceeding. Refer to your
platform user manual to change the debug interface.");

        // Start Mouse
        USBC_Mouse mouse = USBClientController.StandardDevices.StartMouse();

        // Move pointer in a swirl
        const int ANGLE_STEP_SIZE = 15;
        const int MIN_CIRCLE_DIAMETER = 50;
        const int MAX_CIRCLE_DIAMETER = 200;
        const int CIRCLE_DIAMETER_STEP_SIZE = 1;

        int diameter = MIN_CIRCLE_DIAMETER;
        int diameterIncrease = CIRCLE_DIAMETER_STEP_SIZE;
        int angle = 0;
        int factor;
        Random rnd = new Random();
        int i = 0;

        while (true)
        {
```

```csharp
             // we want to do it every sometime randomely
             i = rnd.Next(5000) + 5000;//between 5 and 10 seconds
            Debug.Print("Delaying for " + i + " ms");
            Thread.Sleep(i);
            i = rnd.Next(200) + 100;//do it for a short time
            Debug.Print("Looping " + i + " times!");

            while (i-- > 0)
            {
                // Check if connected to PC
                if (USBClientController.GetState() ==
                    USBClientController.State.Running)
                {
                    // Note Mouse X, Y are reported as change in position
                    // (relative position, not absolute)
                    factor = diameter * ANGLE_STEP_SIZE *
                            (int)System.Math.PI / 180 / 2;
                    int dx = (-1 * factor *
                            (int)Microsoft.SPOT.Math.Sin(angle) / 1000);
                    int dy = (factor *
                            (int)Microsoft.SPOT.Math.Cos(angle) / 1000);

                    angle += ANGLE_STEP_SIZE;
                    diameter += diameterIncrease;

                    if (diameter >= MAX_CIRCLE_DIAMETER ||
                        diameter <= MIN_CIRCLE_DIAMETER
                        )
                        diameterIncrease *= -1;

                    // report mouse position
                    mouse.SendData(dx, dy, 0, USBC_Mouse.Buttons.BUTTON_NONE);
                }

                Thread.Sleep(10);
            }
        }
    }
}
```

28.4. Keyboard

Emulating a keyboard is as easy as emulating a mouse. The following example will create a USB Keyboard and send "Hello world!" to a PC every second.

```csharp
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
```

```csharp
using GHIElectronics.NETMF.USBClient;
using GHIElectronics.NETMF.Hardware;

public class Program
{
    public static void Main()
    {
        // Check debug interface
        if (Configuration.DebugInterface.GetCurrent() ==
            Configuration.DebugInterface.Port.USB1)
            throw new InvalidOperationException("Current debug interface is USB.
            It must be changed to something else before proceeding.
            Refer to your platform user manual to change the debug interface.");
        // Start keyboard
        USBC_Keyboard kb = USBClientController.StandardDevices.StartKeyboard();
        Debug.Print("Waiting to connect to PC...");
        // Send "Hello world!" every second
        while (true)
        {
            // Check if connected to PC
            if (USBClientController.GetState() ==
                USBClientController.State.Running)
            {
                // We need shift down for capital "H"
                kb.KeyDown(USBC_Key.LeftShift);
                kb.KeyTap(USBC_Key.H);
                kb.KeyUp(USBC_Key.LeftShift);
                // Now "ello world"
                kb.KeyTap(USBC_Key.E);
                kb.KeyTap(USBC_Key.L);
                kb.KeyTap(USBC_Key.L);
                kb.KeyTap(USBC_Key.O);
                kb.KeyTap(USBC_Key.Space);
                kb.KeyTap(USBC_Key.W);
                kb.KeyTap(USBC_Key.O);
                kb.KeyTap(USBC_Key.R);
                kb.KeyTap(USBC_Key.L);
                kb.KeyTap(USBC_Key.D);
                // The "!"
                kb.KeyDown(USBC_Key.LeftShift);
                kb.KeyTap(USBC_Key.D1);
                kb.KeyUp(USBC_Key.LeftShift);
                // Send an enter key
                kb.KeyTap(USBC_Key.Enter);
            }
            Thread.Sleep(1000);
        }
    }
}
```

28.5. CDC -Virtual Serial

Serial ports are the most common interface, especially on the embedded system world. This is an ideal solution for devices to transfer data between PCs and embedded devices (FEZ). To combine the popularity and usefulness of USB with the easiness of serial, we have virtual USB devices. To windows applications or to devices, a virtual serial port works just like a serial port but, in reality, it is actually a USB port.

One important thing I want to mention here is that, usually, CDC drivers handle one transaction in every frame. The max EP size on USB is 64 bytes and there 1000 frames/seconds on full-speed USB. This means, the maximum transfer rate for CDC drivers is 64KB/sec. I think, Microsoft realized the needs for CDC and higher transfer rate and did enhance this limitation. Last time I tested the transfer speed on my win7 machine, I was able to see about 500KB/sec.

The following will create a USB CDC and send "Hello world!" to PC every second.

```csharp
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;

using GHIElectronics.NETMF.USBClient;
using GHIElectronics.NETMF.Hardware;

public class Program
{
    public static void Main()
    {
        // Check debug interface
        if (Configuration.DebugInterface.GetCurrent() ==
            Configuration.DebugInterface.Port.USB1)
            throw new InvalidOperationException("Current debug interface is USB.
             It must be changed to something else before proceeding.
             Refer to your platform user manual to change the debug interface.");

        // Start CDC
        USBC_CDC cdc = USBClientController.StandardDevices.StartCDC();

        // Send "Hello world!" to PC every second. (Append a new line too)
        byte[] bytes = System.Text.Encoding.UTF8.GetBytes("Hello world!\r\n");
        while (true)
        {
            // Check if connected to PC
            if (USBClientController.GetState() !=
                USBClientController.State.Running)
            {
                Debug.Print("Waiting to connect to PC...");
            }
            else
```

```
            {
                cdc.Write(bytes, 0, bytes.Length);
            }
            Thread.Sleep(1000);
        }
    }
}
```

28.6. Mass Storage

One of the great GHI unique features of USB client is supporting Mass Storage Class (MSC). This feature allows access to the connected media right from USB. Let me explain this through some example. A data logger application needs to save data to an SD card or USB memory. When the user is done collecting data, they can plug the USB data logger to the PC and now the PC can detect the device as a mass storage device. The user can then transfer the files using standard operating system. Think of the device as a memory card reader. We can even enhance our logger where the USB client interface can be CDC to configure the device and later dynamically switch to MSC to transfer files.

Once very common question on GHI support is "Why can't I access the media while the media is also accessed externally (from windows)?". I will try to explain this better. A media is accessed though a class called PersistentStorage. The PersistentStorage object can then be accessed by the internal file system **or** externally through the USB MSC. Okay, but why not both? It can't be both because files systems cash a lot of info about the media to speed up file access time. Accessing the media simultaneously will definitely cause corruption to the media and therefor, simultaneous access is not allowed. Note that you can easily switch back and forth between internal file system and USB MSC.

This example code assumes an SD card is always plugged in. It enables MSC showing the device (I am using FEZ Domino) as a card reader.

```
using System;
using System.IO;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;

using GHIElectronics.NETMF.USBClient;
using GHIElectronics.NETMF.IO;
using GHIElectronics.NETMF.Hardware;

namespace USBClient_Example
{
```

```csharp
public class Program
{
    public static void Main()
    {
        // Check debug interface
        if (Configuration.DebugInterface.GetCurrent() ==
            Configuration.DebugInterface.Port.USB1)
            throw new InvalidOperationException("Current debug interface is
            USB. It must be changed to something else before proceeding.
            Refer to your platform user manual to change the debug
            interface.");

        // Start MS
        USBC_MassStorage ms =
                USBClientController.StandardDevices.StartMassStorage();

        // Assume SD card is connected
        PersistentStorage sd;
        try
        {
            sd = new PersistentStorage("SD");
        }
        catch
        {
            throw new Exception("SD card not detected");
        }
        ms.AttachLun(0, sd, " ", " ");


        // enable host access
        ms.EnableLun(0);

        Thread.Sleep(Timeout.Infinite);
    }
}
```

28.7. Custom Devices

The GHI USB client Support even allows you to control the USB client in anyway you like. This feature require good knowledge of USB. If you do not know what is EndPoint and Pipe then do not attempt to create custom devices. Also, it is very important to have the device configured correctly the first time it is plugged into windows. This is important because windows stores a lot of information in its registry. So, if you change the configuration of your device after you had it plugged into windows the first time then windows may not see the changes since it will be using the old configuration from its registry.

Basically, stay away from USB Client Custom Devices unless you really have good reason to use them and you have very good knowledge in USB and in windows drivers.

I am not going to explain Custom Devices any further but just wanted to clear out why I decided not to do so. The standard built in classes, explained earlier, should be enough to cover most of your needs.

29. Low Power

Battery powered devices must limit power usage as much as possible.

Devices may lower the power consumption in many ways:

1. Reduce processor clock
2. Shutdown the processor when system is idle (keep peripherals and interrupts running)
3. Shutdown specific peripherals
4. Hibernate the system

A NETMF device may optionally support any of these methods. Consult with your device's user manual to learn more about what is directly supported. In general, all GHI NETMF devices shutdown the processor when in idle state. For example, this will reduce the power on FEZ Rhino by about 45mA. You really do not have to do anything special, as soon as the system is idle, the processor is automatically shut off while interrupts and peripherals are still active.

Also, all peripherals (ADC, DAC, PWM, SPI, UART...) are disabled by default to lower power consumption. They are automatically enabled once they are used.

If all that is not enough, you can completely hibernate the system. Just remember that when the system is hibernating, it is not executing and peripherals are not functional. For example, data coming in on UART will NOT wake up the system. You will simply lose the incoming data. Waking up from hibernate is a system dependent feature, but usually specific pin(s) can be toggled to wake the system up.

Note: GHI realized that the standard NETMF power mode is not suitable for hibernate so a GHI specific method is implemented to handle the system hibernation.

Important Note: When the device is in hibernation, USB stops working. You can't step in code or access the device. If your program always puts the device in sleep then you will not be able to load a new program. Basically, lock out the device. You need to enter the boot loader to erase all to unlock the device from its hibernation!

Important Note: When you hibernate, the system clock stops (not RTC) so the NETMF time will be off. You will need to read the time from RTC and set the NETMF sytsem time after waking up from hibernation.

Waking up from hibernate is only possible on few events. Check your device user manual or the library documentation for details. An easy way to wake up from hibernate is by using the alarm feature (when available). For example, FEZ Rhino includes the 32Khz clock needed for the built in RTC/alarm but FEZ Mini doesn't. See this link for more details

http://www.tinyclr.com/compare

In this example, I set the RTC to some random but valid time and then I blink an LED. Every three seconds, the device sets the alarm to wake up in 10 seconds and then it put the device into hibernation. The device will be completely dead (again, no USB debugging) but you should be able to see this on the LED. When the device wakes up, it continue on blinking the LED. At the end, the LED will blink for 3 seconds and then stops for 10 seconds.

```csharp
using System;
using System.Threading;
using Microsoft.SPOT;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.Hardware.LowLevel;

using GHIElectronics.NETMF.FEZ;
public class Program
{
    public static void Main()
    {
        //blink LED
        FEZ_Components.LED led = new FEZ_Components.LED(FEZ_Pin.Digital.LED);
        led.StartBlinking(100, 100);
        //set to any random but valid time
        RealTimeClock.SetTime( new DateTime(2010, 1, 1, 1, 1, 1));
        while (true)
        {
            Thread.Sleep(3000);//blink LED for 3 seconds
            RealTimeClock.SetAlarm(RealTimeClock.GetTime().AddSeconds(10));
            Debug.Print("Going to sleep for 10 seonds!");
            // sleep for 10 seconds
            Power.Hibernate(Power.WakeUpInterrupt.RTCAlarm);

            Debug.Print("Good Morning!");
        }
    }
}
```

Another option is to wake up on interrupt port. You have to be careful with this because any interrupt on any pin will cause this wake up. For example, the WiFi module on FEZ Cobra internally uses one of the interrupt pins and so this will wake up the system. You need to disable WiFi before hibernating. This is not the only trick that you need to be aware of! Look at the following or try it. It will not work!

```csharp
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.Hardware.LowLevel;
```

```csharp
using GHIElectronics.NETMF.FEZ;
public class Program
{
    public static void Main()
    {
        //blink LED
        FEZ_Components.LED led = new FEZ_Components.LED(FEZ_Pin.Digital.LED);
        led.StartBlinking(100, 100);
        //setup the interrupt pin
        InterruptPort LDR = new InterruptPort((Cpu.Pin)0,false,
                                Port.ResistorMode.PullUp,
                                Port.InterruptMode.InterruptEdgeLow);
        while (true)
        {
            Thread.Sleep(3000);//blink LED for 3 seconds
            // sleep
            Power.Hibernate(Power.WakeUpInterrupt.InterruptInputs);
            //we get here when we wakeup
        }
    }
}
```

Why did the example above not work? When you create an interrupt (or input) pin, interrupts are enabled only if glitch filter is used or if an event handler is installed. So, to make the work above work, you only need to enable the glitch filter. Here is the code that works.

```csharp
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.Hardware.LowLevel;

using GHIElectronics.NETMF.FEZ;
public class Program
{
    public static void Main()
    {
        //blink LED
        FEZ_Components.LED led = new FEZ_Components.LED(FEZ_Pin.Digital.LED);
        led.StartBlinking(100, 100);
        //setup the interrupt pin with glitch filter enableled
        InterruptPort LDR = new InterruptPort((Cpu.Pin)0, true,
                                Port.ResistorMode.PullUp,
                                Port.InterruptMode.InterruptEdgeLow);
        while (true)
        {
            Thread.Sleep(3000);//blink LED for 3 seconds
            // sleep
            Power.Hibernate(Power.WakeUpInterrupt.InterruptInputs);
            //we get here when we wakeup
```

```
        }
    }
}
```

Another option is to install an event handler for the button like the following.

```csharp
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.Hardware.LowLevel;

using GHIElectronics.NETMF.FEZ;
public class Program
{
    public static void Main()
    {
        //blink LED
        FEZ_Components.LED led = new FEZ_Components.LED(FEZ_Pin.Digital.LED);
        led.StartBlinking(100, 100);
        //setup the interrupt pin
        InterruptPort LDR = new InterruptPort((Cpu.Pin)0, false,
                              Port.ResistorMode.PullUp,
                              Port.InterruptMode.InterruptEdgeLow);
        LDR.OnInterrupt += new NativeEventHandler(LDR_OnInterrupt);

        while (true)
        {
            Thread.Sleep(3000);//blink LED for 3 seconds
            // sleep
            Power.Hibernate(Power.WakeUpInterrupt.InterruptInputs);
            //we get here when we wakeup
        }
    }

    static void LDR_OnInterrupt(uint data1, uint data2, DateTime time)
    {
        // empty for now!
    }
}
```

Please note that using InputPort is as good as using the InterruptPort since, internally, the interrupts are used when glitch filter is enabled. Here is the example using InputPort instead of InterruptPort.

```csharp
using System;
using System.Threading;
```

```
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.Hardware.LowLevel;

using GHIElectronics.NETMF.FEZ;
public class Program
{
    public static void Main()
    {
        //blink LED
        FEZ_Components.LED led = new FEZ_Components.LED(FEZ_Pin.Digital.LED);
        led.StartBlinking(100, 100);
        //setup the interrupt pin
        InterruptPort LDR = new InterruptPort((Cpu.Pin)0, false,
                                  Port.ResistorMode.PullUp,
                                  Port.InterruptMode.InterruptEdgeLow);
        LDR.OnInterrupt += new NativeEventHandler(LDR_OnInterrupt);

        while (true)
        {
            Thread.Sleep(3000);//blink LED for 3 seconds
            // sleep
            Power.Hibernate(Power.WakeUpInterrupt.InterruptInputs);
            //we get here when we wakeup
        }
    }

    static void LDR_OnInterrupt(uint data1, uint data2, DateTime time)
    {
        // empty for now!
    }
}
```

This example will blink an LED when up and then when the LDR button is pressed, the system will go into deep sleep for 10 se

This is an example for FEZ Domino (USBizi) that will go into deep sleep and wake up when I press a button??????????

```
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
using GHIElectronics.NETMF.Hardware.LowLevel;

using GHIElectronics.NETMF.FEZ;
public class Program
```

```
{
    public static void Main()
    {
        //blink LED
        FEZ_Components.LED led = new FEZ_Components.LED(FEZ_Pin.Digital.LED);
        led.StartBlinking(100, 100);
        //setup the interrupt pin
        InputPort LDR = new  InputPort((Cpu.Pin)0, true,
                                Port.ResistorMode.PullUp);
        while (true)
        {
            Thread.Sleep(3000);//blink LED for 3 seconds
            // sleep
            Power.Hibernate(Power.WakeUpInterrupt.InterruptInputs);
            //we get here when we wakeup
        }
    }
}
```

Finally, you can wake up on more than one events. For example, you can wake up if a button is pressed or the alarm is fired.

```
Power.Hibernate(Power.WakeUpInterrupt.InterruptInputs |
Power.WakeUpInterrupt.RTCAlarm);
```

30. Watchdog

In embedded system world, devices are usually always running and with no user interaction. So if something went wrong, it would be very beneficial if we have an automatic reset button. Watchdog is your reset button!

System Execution Recovery

Assume you are making a smart vending machine that reports its inventory over the network. If your code raises an exception that wasn't handled properly then your program will end. Program ended means the vending machine no longer work. Someone will have to drive to the vending machine to reset it, or better, just use watchdog.

When you enable the watchdog, you give it a timeout to reset after. Then, you keep resetting the watchdog timer periodically. If the system locks up then the watchdog counter will reach the timeout, which in turn, will reset the system.

Important note: GHI realized that the built-in watchdog in NETMF is not what customers need; therefore, GHI implemented its own version of watchdog. Do not use the watchdog from Microsoft.SPOT.Hardware, instead, use the watchdog from GHIElectronics.NETMF.Hardware.LowLevel. If both namespaces must be used then you will have ambiguity errors and so you need to specifically call out the full path of the watchdog you need. For example, instead of using Watchdog.Enable(timeout), use GHIElectronics.NETMF.Hardware.LowLevel.Watchdog.Enable(timeout).

This example shows how to set the watchdog to 5 seconds timeout and create a thread to clear the watchdog every 3 seconds. Should anything go wrong, the device will reset in 5 seconds.

Important Note: Once you enable the Watchdog, it cannot be disabled. So you have to keep resetting the timeout. This is done to assure that no system corruption will disable watchdog accidentally.

```csharp
using System;
using System.Threading;
using GHIElectronics.NETMF.Hardware.LowLevel;

public class Program
{
    public static void Main()
    {
        // Timeout 5 seconds
        uint timeout = 1000 * 5;

        // Enable Watchdog
        Watchdog.Enable(timeout);
```

```csharp
        // Start a time counter reset thread
        WDTCounterReset = new Thread(WDTCounterResetLoop);
        WDTCounterReset.Start();

        // ....
        // your program starts here

        // If we exit the program,
        // the thread will stop working and the system will reset!
        Thread.Sleep(Timeout.Infinite);
    }

    static Thread WDTCounterReset;
    static void WDTCounterResetLoop()
    {
        while (true)
        {
            // reset time counter every 3 seconds
            Thread.Sleep(3000);

            Watchdog.ResetCounter();
        }
    }
}
```

You may be thinking, if the software locked up then how would the code that handles watchdog ever run? On low level, the watchdog is supported in hardware not software. This means that the counter and the and reset mechanism is done inside the processor, without the need for any software.

Limiting Time-Critical Tasks

Back to our vending machine example, but this time we want to handle a different possible problem. Since NETMF is not real time, tasks can take longer than expected. If a person stepped up to the vending machine and entered what they want to buy, the machine will now turn a motor on which in turn will push the item out to the user. Let us say this was timed so the motor has to be on for one second. Now, what if it happened that at the same time the motor is running another thread started using the SD card. Let us also assume that the card had some problem which caused 3 second delay in reading the SD card. The 3 seconds delay while the motor is running will result in 3 items being pushed out to the buyer, but we only wanted to push one!

If we used watchdog and set it to one second, then the user will have one item and when time is exceeding one second, the vending machine will reset, which will stop the motor from pushing more items out.

Here is a simple example.

```csharp
// ....
// ....
// Timeout 1 second
uint timeout = 1000;

//....user buys something

// Enable Watchdog
Watchdog.Enable(timeout);
//turn motors on
//...
//stop motors
//...
// We don't need Watchdog anymore, we have to keep resetting the timeout
Watchdog.Enable(Watchdog.MAX_TIMEOUT);
WDTCounterReset.Start();    // See this in the earlier example

// ....
// your program starts here
// ....
// ....
```

Detecting Watchdog Cause

In some cases, you need to know if the system did reset because of a watchdog to log this info or run some recovery procedures.

This is how it works

```csharp
using System;
using System.Threading;
using Microsoft.SPOT;
using GHIElectronics.NETMF.Hardware.LowLevel;


public class Program
{
    public static void Main()
    {
        // you can read this flag ***ONLY ONCE*** on power up
        if (Watchdog.LastResetCause == Watchdog.ResetCause.WatchdogReset)
        {
            Debug.Print("Watchdog did Reset");
        }
        else
        {
            Debug.Print("Reset switch or system power");
        }
    }
}
```

31. Wireless

Wireless technologies are becoming an essential part of our life. Some applications require high rate data transfer, others require very low power. Some require a point to point connection, others require a mesh network.

The biggest challenge when designing a wireless device is certification. Not only this needs to be done for different countries, it is very costly. You can easily spend 50,000 USD on certifying a wireless device. Fortunately, there are companies who offer certified wireless modules. When using a certified module, your product may not need any certification or the certification will be much easier.

31.1. Zigbee (802.15.4)

Zigbee is designed to be used in low-power battery-operated sensors. Many sensors can connected on one Zigbee network. Also, it requires very low power but data transfer rate is not very fast.

One very common implementation of Zigbee is XBee modules offered from Digi. There are many types of XBee modules. Some are very low power and other provides a very high output capable of transferring data up to 15 miles! Those modules are also offered on on-board antenna or with a connector for external antenna.

The modules have a very simple interface that runs on UART. With UART, the modules can be interfaced to any NETMF device. If creating a connection between two modules then XBee modules establish a connection automatically. If connecting multiple nodes then we would need to send some serial commands to configure our network. I suggest you start with automatic point-to-point connection.

Connecting XBee to FEZ Mini or FEZ Domino can be easily done using the Xbee expansion component.
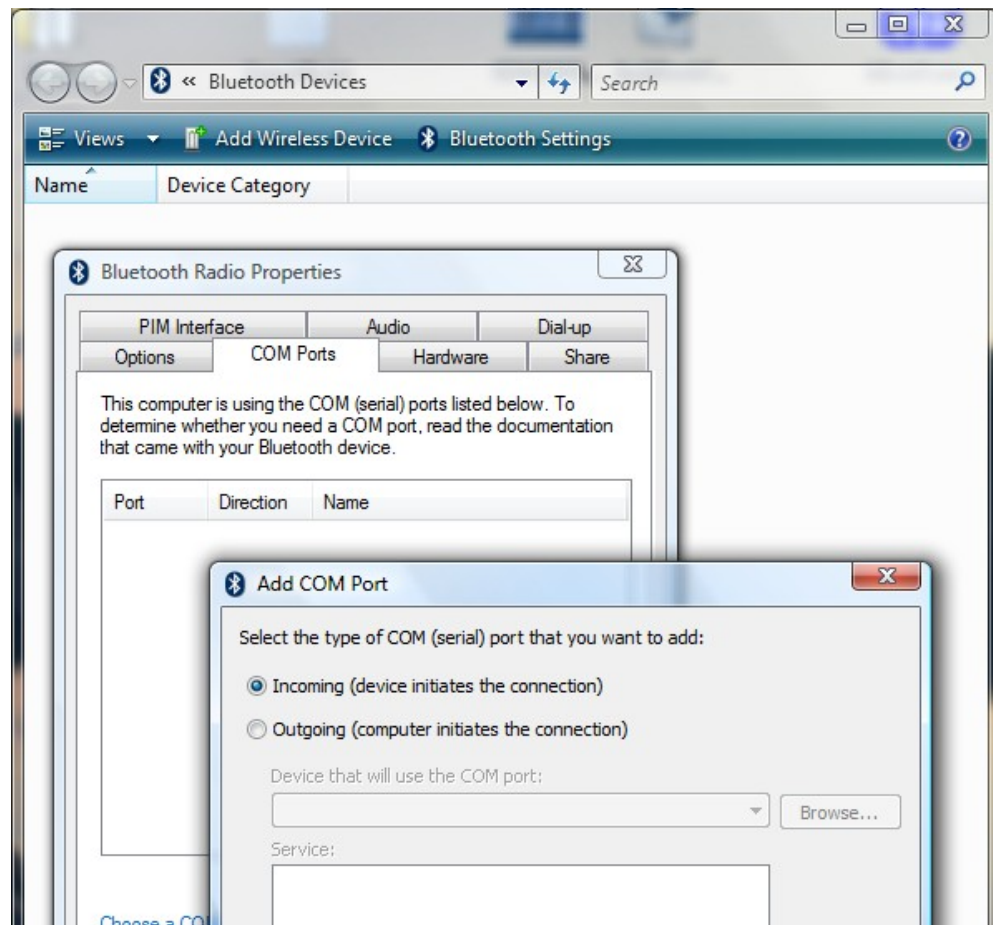
31.2. Bluetooth

Almost every cellphone has everything needed to connect to Bluetooth devices. Bluetooth technology defines multiple profiles for connections. The audio profile is useful to connect the Bluetooth ear-piece to the phone. For data transfers, SPP (Serial Port Profile) is used to establish a connection that simulates a serial connection. This is very similar to how XBee modules connect. Most phones have Bluetooth but most do not implement SPP so creating a serial connection from your phone is probably not possible.

On windows, we can create a serial Bluetooth connection with few clicks.

1. Find "Bluetooth Devices" in "Control Panel"

2. Click on "Bluetooth Settings"

3. View the "COM ports" tab

4. If you have COM ports installed then you already have SPP enables on your PC.

5. To add new ports, click on "Add..."

6. Create one port for incoming data and one for outgoing.

Windows creates two ports, one for incoming and one for outgoing data. This can be confusing to Windows users because they can't use the same port to send and receive data!

On the embedded side, there are many serial modules that already include Bluetooth hardware and software, including SPP (Serial Port Profile). Those can easily connect to any NETMF device's serial port.

Connecting Bluetooth to FEZ Mini or FEZ Domino can be easily done using the Bluetooth interface component.

31.3. Nordic
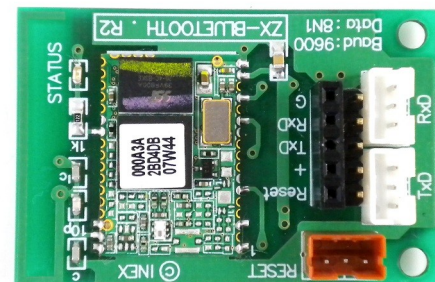
Nordic semiconductor created it own digital wireless chips, NRF24L01. These low-power chips use 2.4Ghz which is a free band in many countries. Nordic wireless chips allow point-to-point or multi-point wireless connections.

Olimex offers breakout boards for NRF24L01. Those boards can connect directly to most GHI's NETMF boards.

This is a project (and video) showing two NETMF devices using NRF24L01

http://www.microframeworkprojects.com/index.php?title=SimpleWireless

32. Objects in Custom Heap

Note that this topic is about very large memory allocation and so it doesn't apply to USBizi.

Managed systems like NETMF requires very complex memory management. To limit the overhead on small systems, NETMF heap only supports allocating objects up to 700KBs. Larger objects are not possible. Beginning in NETMF version 4.0, larger buffers can be allocated using a separate heap called **custom heap**. Now, you can create very large buffers and very large bitmaps. Internally, these objects are not in the standard managed heap but they are in custom heap. This introduced a new important question. How much memory is reserved for custom heap and how much for managed heap?

GHI provides APIs allowing you to set the size of each heap, custom and managed.

```csharp
using System;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
class Program
{

    public static void Main()
    {
        // set the heap size to 4MB
        if (Configuration.Heap.SetCustomHeapSize(4 * 1024 * 1024))
        {
            // this will only take effect after resetting the system
            PowerState.RebootDevice(false);
        }
        // ...
        // you can now use large objects up to 4MB
    }
}
```

32.1. Management of Custom Heap

Unlike the regular heap which is fully managed, the custom heap is managed but with few important points that need to be considered. In order for objects in custom heap to be cleared automatically, you must fulfill three requirements:

- The object reference must be lost. (This is all you need for regular objects).

- The garbage collector must run, you probably need to force it.

- The system must come idle in order for finalize to run and "Dispose" object.

I will not go into detail on garbage collectors, dispose, or finalizer...etc. What you need to know is that large object that are on custom heap are not easily cleared from the system so you must always dispose the object when you are done using it.

This is how you can dispose large objects

```csharp
using System;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
class Program
{

    public static void Main()
    {
        // allocate a 1MB buffer
        LargeBuffer lb = new LargeBuffer(1024 * 1024);
        // use the buffer
        lb.Bytes[5] = 123;
        // ....
        // when done, dispose the object to empty the memory
        lb.Dispose();
    }
}
```

The better option is to use the "using" statement. This statement will automatically call dispose once the execution is leaving the "using" curly brackets.

```csharp
using System;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
class Program
{
    public static void Main()
    {
        // allocate a 1MB buffer
        using (LargeBuffer lb = new LargeBuffer(1024 * 1024))
        {
            // use the buffer
            lb.Bytes[5] = 123;
            // ...
        }
        // Dispose was called automatically
        // ...
    }
}
```

32.2. Large Bitmaps

Bitmaps larger than 750KB/2 are automatically allocated on the custom heap. For example, a 800x480 bitmap will need 800x480x4 bytes, that is about 1.5MB. This large bitmap is on the custom heap and so you should always dispose it when you are done using it, or utilize the "using" statement.

```csharp
using System;
using Microsoft.SPOT;
using Microsoft.SPOT.Presentation.Media;
class Program
{
    public static void Main()
    {
        using(Bitmap largeBitmap = new Bitmap(800,480))
        {
            // assumign we have a display that is 800x480
            // draw a circle
            largeBitmap.DrawEllipse(Colors.Green, 100, 100, 10, 10);
            // draw other things
            // ...
            largeBitmap.Flush();//flush the Bitmap object to the display
        }
        // once we are here largeBitmap Dispose was automatically called
        // ...
    }
}
```

32.3. LargeBuffer

We already used LargeBuffer in earlier example. Only use this if you need to allocate a buffer that is larger than 750KB. On an embedded system, you shouldn't need or use such very large buffers. Although not recommended, this is available in case it is needed.

```csharp
using System;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.Hardware;
class Program
{
    public static void Main()
    {
        // allocate a 1MB buffer
        using (LargeBuffer lb = new LargeBuffer(1024 * 1024))
        {
```

```
                // use the buffer
                lb.Bytes[5] = 123;
                byte b = lb.Bytes[5];
            }
        // Dispose was called automatically
        // ...
    }
}
```

33. Thinking Small

Many NETMF developers come from the PC world. They are used to write code that runs fine on a PC but then it will not run efficiently on an embedded device. The PC can be 4GHz with 4GB of RAM. NETMF devices have less than 1% of the resources available on a PC. I will cover different areas where you should always think small.

33.1. Memory Utilization

With limited RAM, developers should only use what they really need. PC programmers tend to make a large buffer to handle the smallest task. Embedded Developers study what they need and only allocate the needed memory. If I am reading data from UART, I can very well use 100 byte buffer to read the data and 1000 byte buffer will work as well. While I am analyzing the code, I noticed that I always read about 40 bytes from UART in my program loop. I do send a large buffer but I only get back 40 bytes. So, why would I want to use a buffer larger than 40 bytes? Maybe I will make it a bit large just in case but defiantly not 1000 bytes!

On some drivers, the NETMF system does a lot of buffering internally. For example, file system, UART, USB drivers all have internal buffers in native code, to keep the data ready until the developer uses the data from managed code. If we need a 1 megabyte file, we do not need a large buffer at all. We create a small buffer and then send the data in chunks to the file system. To play a 5 megabyte MP3 file, we only need 100 byte buffer that will read chunks from the file and pass to the MP3 decoder.

33.2. Object Allocation

Allocating and freeing objects is very costly. Only allocate objects when you really needs them. Also, you are making an embedded device; therefore, a lot of objects that you will be using are always used. For example, you will always use the LCD or always use the SPI.

Consider the following code

```csharp
using System.Threading;
using System;
using Microsoft.SPOT.Hardware;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
```

```
        {
            static void WriteRegister(byte register_num, byte value)
            {
                SPI _spi = new SPI(new
                        SPI.Configuration(Cpu.Pin.GPIO_NONE,false,0,0,false,
                        true,1000,SPI.SPI_module.SPI1));

                byte[] buffer = new byte[2];
                buffer[0] = register_num;
                buffer[1] = value;
                _spi.Write(buffer);
            }
            public static void Main()
            {
                WriteRegister(5, 100);
            }
        }
}
```

In order for me to write a single byte to a register on a SPI-chip, I had allocated SPI object, SPI.Configuration object and a byte array. Three objects for sending one byte! This is okay if you only need to do this a few times at initialization stage but if you are continuously using the WriteRegister method then this is not the right way. For starters, this method will run very slow so you wouldn't be able to "WriteRegister" fast enough. Maybe this is sending graphics to the display or sending MP3 data to a decoder. This means that our function will be called few thousand times every second. As second problem is that these objects are created used and then left for the garbage collector to remove. The garbage collector will have to jump in and remove all these unused objects from memory which will stop the program execution for few milliseconds.

Here is the code modified to test the method when called 1000 times.

```
using System.Threading;
using System;
using Microsoft.SPOT.Hardware;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
    {
        static void WriteRegister(byte register_num, byte value)
        {
            SPI _spi =
                new SPI(new SPI.Configuration(Cpu.Pin.GPIO_NONE,
                        false,0,0,false,
                        true,1000,SPI.SPI_module.SPI1));
            byte[] buffer = new byte[2];
```

```
                buffer[0] = register_num;
                buffer[1] = value;
                _spi.Write(buffer);
                _spi.Dispose();
        }
        public static void Main()
        {
            long ms;
            long ticks = DateTime.Now.Ticks;
            for (int i = 0; i < 1000; i++)
                WriteRegister(5, 100);
            ticks = DateTime.Now.Ticks - ticks;
            ms = ticks / TimeSpan.TicksPerMillisecond;
            Debug.Print("Time = " + ms.ToString());
        }
    }
}
```

When running the code on on FEZ (USBizi) we notice that the Garbage Collector had to run 10 times. The garbage collector prints it activity on the output window. Time taken for the code to run is 1911 ms, which is about 2 seconds!

Now, let us modify the code as showing below. We now have the SPI object created globally and will always be there. We are still allocating the buffer in every loop.

```
using System.Threading;
using System;
using Microsoft.SPOT.Hardware;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
    {
        static SPI _spi = new SPI(new SPI.Configuration(
            Cpu.Pin.GPIO_NONE, false, 0, 0, false,
            true, 1000, SPI.SPI_module.SPI1));

        static void WriteRegister(byte register_num, byte value)
        {
            byte[] buffer = new byte[2];
            buffer[0] = register_num;
            buffer[1] = value;
            _spi.Write(buffer);
            _spi.Dispose();
        }
        public static void Main()
        {
            long ms;
```

```
            long ticks = DateTime.Now.Ticks;
            for (int i = 0; i < 1000; i++)
                WriteRegister(5, 100);
            ticks = DateTime.Now.Ticks - ticks;
            ms = ticks / TimeSpan.TicksPerMillisecond;
            Debug.Print("Time = " + ms.ToString());
        }
    }
}
```

In the second example, the garbage collector had to run only twice and it took only 448 milliseconds, about half a second to run. We only moved one line of code and it is 4 times faster.

Let us move the buffer globally and see.

```
using System.Threading;
using System;
using Microsoft.SPOT.Hardware;
using Microsoft.SPOT;

namespace MFConsoleApplication1
{
    public class Program
    {
        static SPI _spi = new SPI(new SPI.Configuration(
            Cpu.Pin.GPIO_NONE, false, 0, 0, false,
            true, 1000, SPI.SPI_module.SPI1));
        static byte[] buffer = new byte[2];
        static void WriteRegister(byte register_num, byte value)
        {
            buffer[0] = register_num;
            buffer[1] = value;
            _spi.Write(buffer);
            _spi.Dispose();
        }
        public static void Main()
        {
            long ms;
            long ticks = DateTime.Now.Ticks;
            for (int i = 0; i < 1000; i++)
                WriteRegister(5, 100);

            ticks = DateTime.Now.Ticks - ticks;
            ms = ticks / TimeSpan.TicksPerMillisecond;
            Debug.Print("Time = " + ms.ToString());
        }
    }
}
```

We now have 368 milliseconds and garbage collector didn't run at all!

One quick check you can do on your device is to check the output to see how often does the garbage collector run. On systems with large memory like ChipworkX this will not help much so you still need to analyze the code manually.

34. Missing Topics

These are topics not covered by this book. I will give a very quick review to cover what the topic is about. I may expand this book to cover these topics in future.

34.1. WPF

Windows Presentation Foundation is a new and flexible way to create graphical user interface applications. GHI's EMX and ChipworkX support WPF. USBizi and FEZ do not support this feature.

34.2. DPWS

Device Profile for Web Services allows networked devices to be automatically detected and used on the network. DPWS requires full .NET sockets support.

34.3. Extended Weak Reference

Extended Weak Reference(EWR) allows developers to save little data in nonvolatile memory. EWR is used more before File System was introduces to NETMF.

34.4. Serialization

Serialization is a way to convert an object to a series of bytes that represent that object. An object Mike made from a Human type can be serialized into a byte array and then this data is transferred to another device. The other device knows what a human type is but doesn't know anything about Mike. It will take this data to construct a new object based on it and it now has a copy of the object Mike.

Using serialization in NETMF is a very bad idea unless you really needs it. There are few methods that will help you in extract or place values in array. You should be using these methods instead of serialization which is too slow.

34.5. Runtime Loadable Procedures

Runtime Loadable Procedures(RLP) is a GHI exclusive feature that allows users to write native (assembly/C) code for a device then load it and use it through managed (C#) at runtime. Native code is thousands times faster but it is not easy to manage. Specific tasks like

calculating CRC that is a very processor intensive function are a perfect for for RLP. The complete application is made using manage code (C#) but then only CRC calculating method is written in native code (assembly/C).

34.6. Databases

A database stores data in a way where queering for data is easy. Looking up a product or sorting numbers is very fast because of the indexing databases do internally.

34.7. Touch Screen

NETMF supports touch screens. Touch screens are a good combination with TFT displays. A developer can create a graphical application using WPF and then the user can control it using the touch screen.

34.8. Events

If we have a project that receives data from serial ports, we need to read the serial port continuously. We may not have any data but we do not know will we check for the data. It will be more efficient if we can be notified if the serial driver had received data. This notification comes from an event that fires when the serial driver receives data.

The same applies to interrupt ports covered before. This book doesn't cover the creation of events but we already seen how they are used in interrupt ports and used in using a mouse with USB host support.

34.9. USB Host Raw

We have learned how to access some USB devices using the GHI exclusive USB host support. GHI allows users to write managed drivers for almost any USB device.

Accessing USB directly is considered a very advanced feature and is left out of this book.

This is a project that uses USB raw access to read an XBOX Controller:

http://www.microframeworkprojects.com/index.php?title=Xbox_Controller

Another interesting project is the NXT drivers allowing users to control LEGO NXT Mindstorm right from C# and Visual Studio.

http://www.microframeworkprojects.com/index.php?title=NXT_Mindstorm

35. Final Words

If you found this book useful and it saved you few minutes of research then I have accomplished what I had in mind. I very much thank you for your downloading and reading this book.

35.1. Further Reading

This book only covers the basics of C# and .NET Micro Framework. This is a list of some resources to learn more:

- My blog is always a good place to visit
  http://tinyclr.blogspot.com/
- The Micro Framework Project website is an excellent resource
  http://www.microframeworkprojects.com/
- A good and free eBook to continue learning about C# is available at
  http://www.programmersheaven.com/2/CSharpBook
- Jens Kuhner excellent book on .NET Micro Framework
  http://www.apress.com/book/view/9781430223870

- USB complete is an excellent book on USB
  http://www.lvr.com/usbc.htm

- Wikipedia is my favorite place for information about everything!
  http://en.wikipedia.org/wiki/.NET_Micro_Framework

- .NET Micro Framework main page on Microsoft's website
  http://www.microsoft.com/netmf

35.2. Disclaimer

This is a free book only if you download it directly from GHI Electronics. Use it for your own knowledge and at your own risk. Neither the writer nor GHI Electronics is responsible for any damage or loss caused by this free eBook or by any information supplied by it. There is no guarantee any information in this book is valid.

USBizi, Embedded Master, EMX, ChipworkX, RLP and FEZ are trademarks of GHI Electronics, LLC

Visual Studio and .NET Micro Framework are trademarks or registered trademarks of Microsoft Corporation.