

Beginners Guide to C# and the .NET Micro Framework

**September 24th 2012
Rev 2.1**



Copyright © 2012 GHI Electronics, LLC
www.GHIElectronics.com
Community: www.TinyCLR.com
Gus Issa

Licensed under Creative Commons Share Alike 3.0



Table of Contents

1.About the Book.....	3	6.3.Interrupt Port.....	30
1.1.Intended Audience.....	3	6.4.Tristate Port.....	31
1.2.Disclaimer.....	3	7.C# Level 2.....	33
2.Introduction.....	4	7.1.Boolean Variables.....	33
2.1.Advantages.....	4	7.2.if-statement.....	35
3..NET Gadgeteer.....	5	7.3.if-else-statements.....	36
4.Getting Started.....	6	7.4.Methods and Arguments.....	38
4.1.System Setup.....	6	7.5.Classes.....	39
4.2.The Emulator.....	6	7.6.Public vs. Private.....	40
Create a Project.....	6	7.7.Static vs. non-static.....	40
Selecting Transport.....	8	7.8.Constants.....	41
Executing.....	9	7.9.Enumeration.....	41
Breakpoints.....	10	8.Assembly/Firmware Matching.....	44
4.3.Running on Hardware.....	11	Boot-up Messages.....	44
MFDeploy can Ping!.....	11	9.Garbage Collector.....	46
Deploying to Hardware.....	12	9.1.Losing Resources.....	47
5.C# Level 1.....	13	9.2.Dispose.....	48
5.1.What is .NET?.....	13	9.3.GC Output Messages.....	49
5.2.What is C#?.....	13	10.C# Level 3.....	50
“Main” is the Starting Point.....	13	10.1.Byte.....	50
Comments.....	14	10.2.Char.....	50
while-loop.....	15	10.3.Array.....	51
Variables.....	16	10.4.String.....	52
Assemblies.....	18	10.5.For-Loop.....	53
What Assemblies to Add?.....	21	10.6.Switch Statement.....	55
Threading.....	22	11.Additional Resources.....	58
6.Digital Input & Output.....	25	Tutorials and Downloads.....	58
6.1.Digital Outputs.....	25	Community: Forum, Codeshare and more.....	58
Blink an LED.....	27	eBooks.....	58
6.2.Digital Inputs.....	28	Books.....	58

1. About the Book

1.1. Intended Audience

This book is for beginners wanting to get started on .NET Micro Framework. No prior knowledge is necessary. The book covers the basics of .NET Micro Framework, Visual Studio and even C#!

If you're 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.

1.2. Disclaimer

This is a free book 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.

2. 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 a PC is 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 won'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 microcontroller you chose (which probably very expensive), a week to learn the processor, a week to write a 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 or months of work.

2.1. Advantages

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

1. It runs on Microsoft's Visual C# Express, free and high-end IDE.
2. .NET Micro Framework is open-source and free.
3. Your same code will run any NETMF device with almost no changes.
4. Full debugging capabilities. (Breakpoints, stepping in code, variables...etc.)
5. Has been tested in many commercial products, with assured quality.
6. Includes many bus drivers.(SPI, UART , I2C...etc.)
7. Eliminates the need to use complicated and long processors' datasheets.
8. If you are already a PC C# programmer then you are know NETMF.

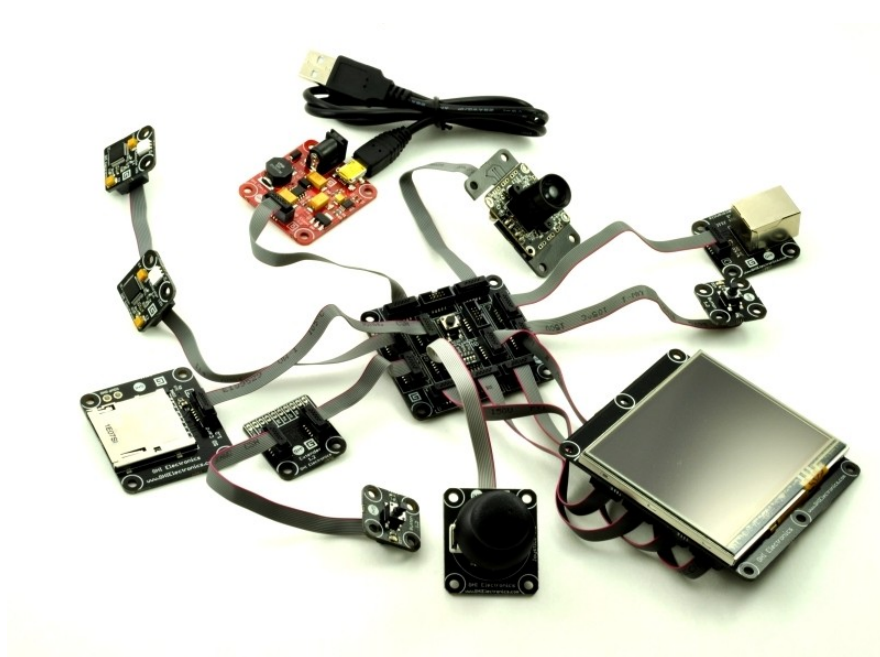
Throughout this document, I will refer to .NET Micro Framework as NETMF.

3. .NET Gadgeteer

.NET Gadgeteer platform definitions take NETMF's flexibility to the next level. It is a set of rules on how hardware modules can interact with a mainboard. The complete specifications and software are open-source. GHI Electronics is proud to be the first to offer a .NET Gadgeteer platform.

More details are found at <http://www.ghielectronics.com/products/dotnet-gadgeteer>

This book is not meant to be used for Gadgeteer devices but it is a great tool to understanding NETMF, which is Gadgeteer's core engine.



4. 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. This page summarizes the firmware update steps http://wiki.tinyclr.com/index.php?title=Firmware_Update

Also, make sure to read the “firmware/assembly matching” section in this book.

4.1. 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**

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

Now, download and install .NET Micro Framework 4.2 SDK (not the porting kit) and the GHI package installer. Both are found on this page

<http://www.ghielectronics.com/support/dotnet-micro-framework>

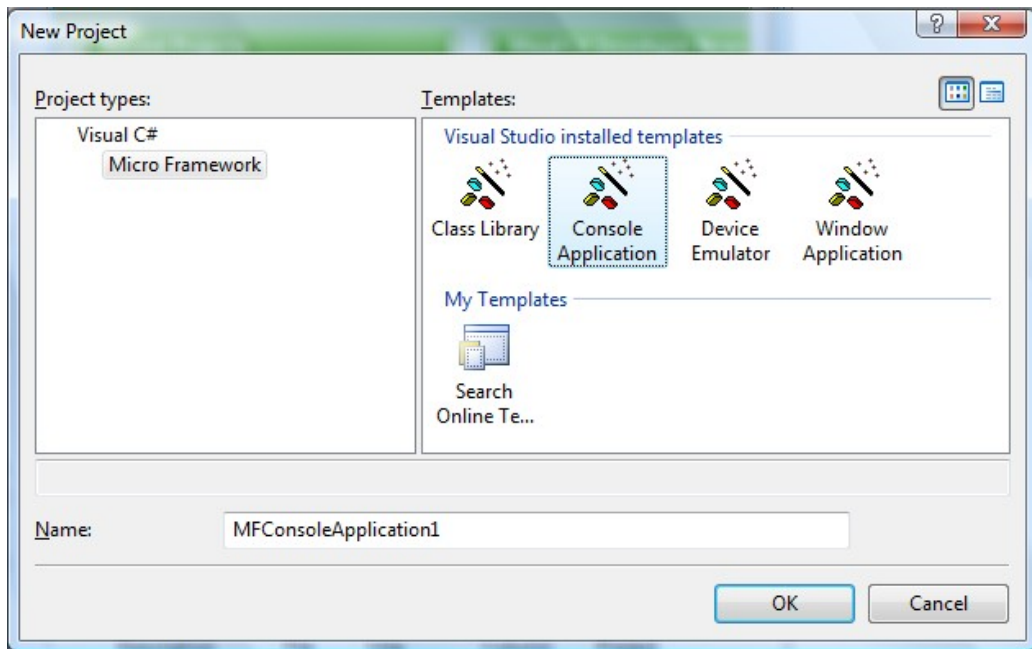
By the way, you may want to bookmark this page as it has about everything you need to use NETMF.

4.2. The Emulator

NETMF includes an emulator that allows you to run NETMF applications right on your 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 should now have the “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 only has 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.

```
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 to look like the listing below

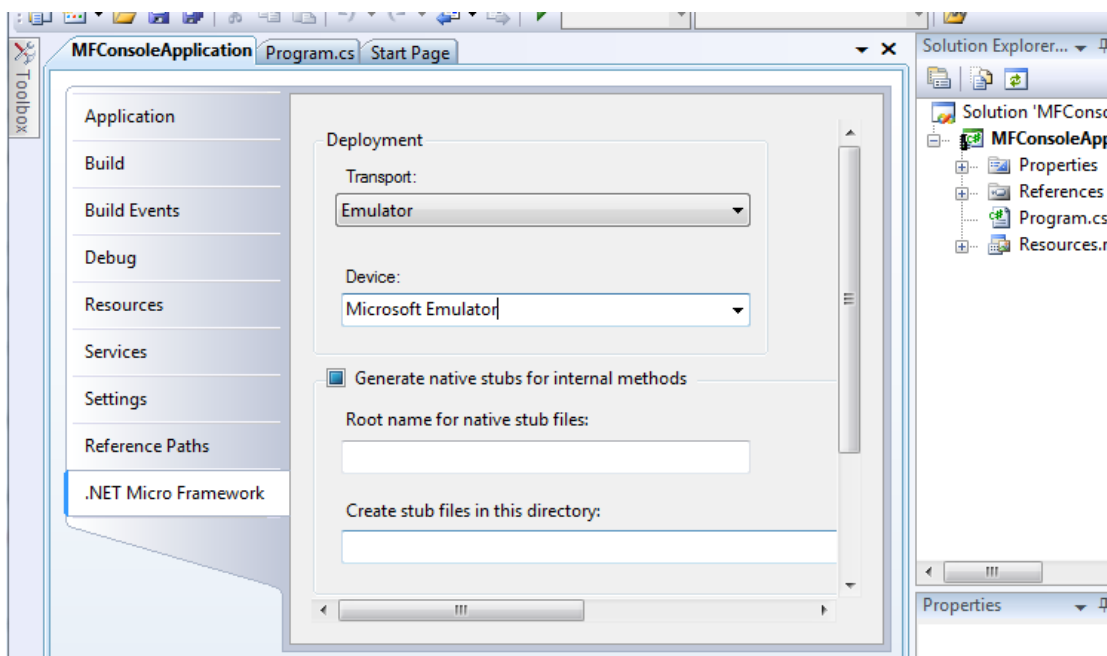
```
using System;
using Microsoft.SPOT;

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's 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

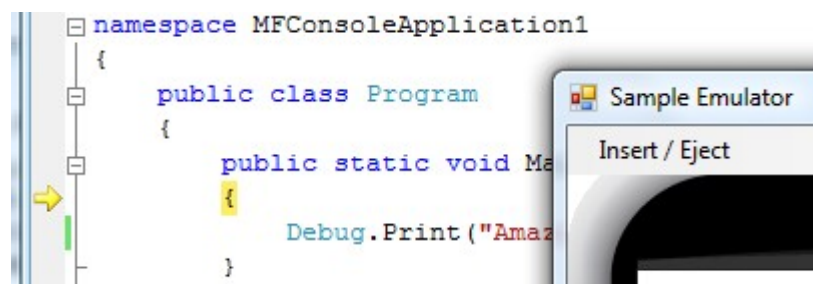
If you are not seeing such option, then you need to enable expert settings first, found at Tools-> Settings-> Expert Settings.

Executing

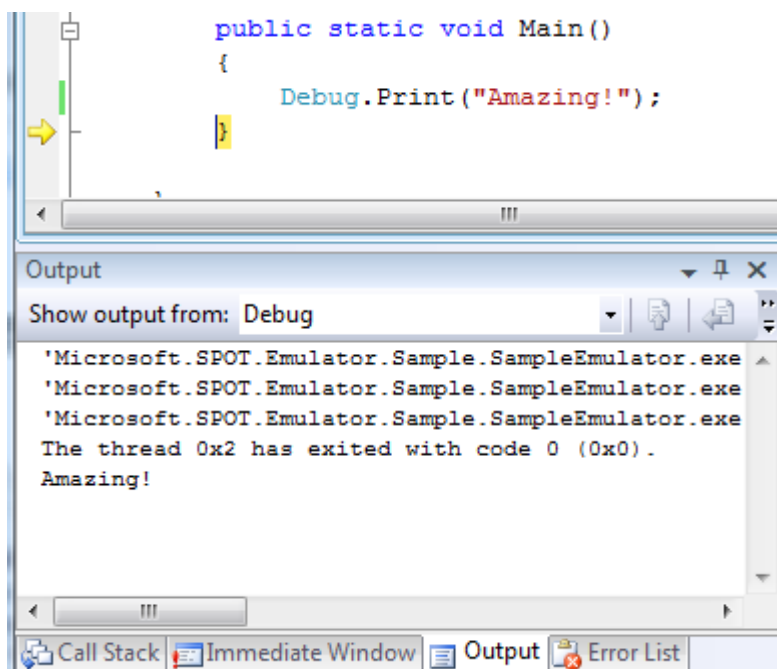
Finally, we are ready to run our first application. Press F5 key on the computer. This is a very useful shortcut and you will be using it a lot to run your applications. After you press F5, the application will be compiled and loaded on the emulator and in couple seconds everything will stop! That is 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 code. This is indicated by the yellow arrow.



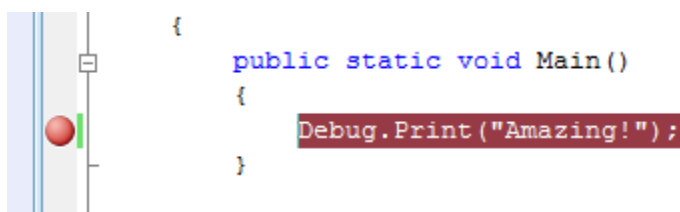
C# applications always start from a method called Main and this is where the arrow stopped. Press F11 again and the debugger will run the next line of code, which is the line you changed before. You probably have guessed it right, this line will print “Amazing!” to the debug window. The debug window 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.



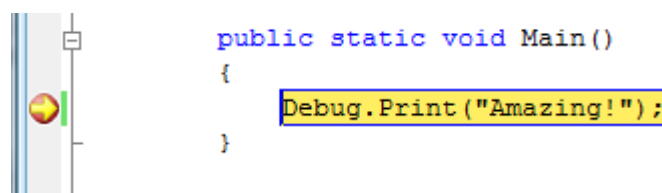
If you now press F11 again, the program will end and the emulator will exit.

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 to the left of the line that prints “Amazing!” This will show a red dot which indicates a breakpoint.



Now press F5 to run the software and when the application reaches the breakpoint, the debugger will pause, as shown in the image below



Now, you can step in the code using F11 or continue execution using F5.

4.3. Running on Hardware

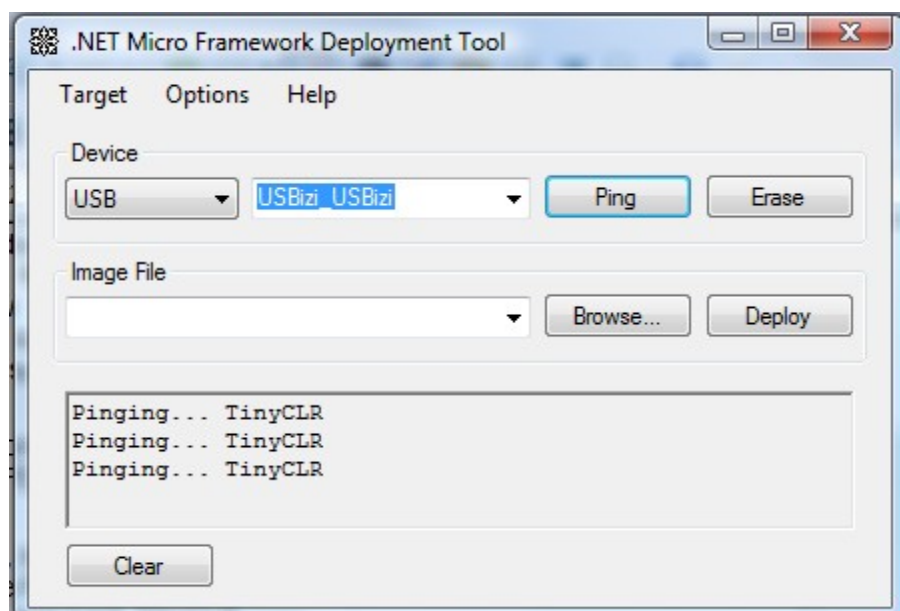
Running NETMF applications on hardware is very simple. Instructions can be slightly different on different hardware. This book uses FEZ for demonstration purposes but any other hardware will work similarly.

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” means that MFDeploy will say “Hi” to the device and then checks if the device will respond with “Hi” back. This is good to make sure the device is connected properly and transport with it has no issues.

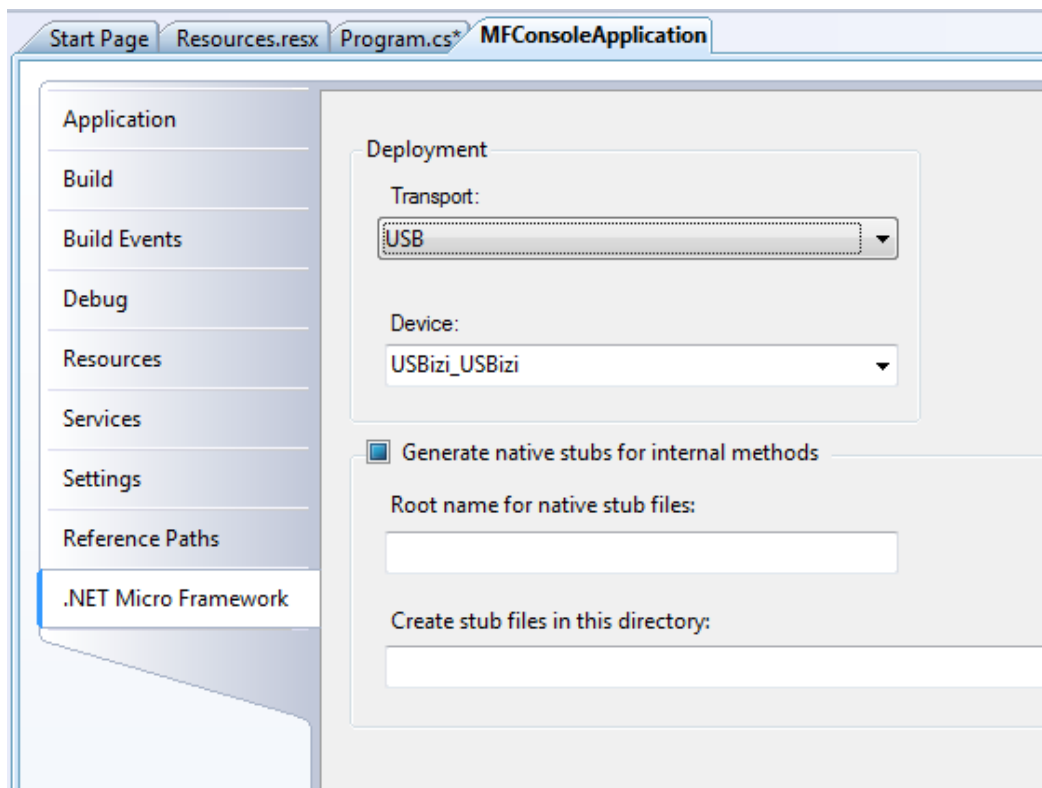
Open MFDeploy and connect FEZ using the included USB cable to your PC. If this is the first time you plugged in FEZ, Windows will look for the drivers and automatically find them. If not, supply the driver from the SDK folder and wait till windows is finished.

In the drop-down menu, select USB. You should see your device showing in the device list. In my example, I see USBizi. Select your device and click the “Ping” button. You should now see TinyCLR.



Deploying to Hardware

Now that we checked that the hardware is connected using MFDeploy, we need to go back to Visual Studio. From the project properties, select USB for transport and then your device (mine is USBizi). 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 back to the output window on Visual Studio.

5. C# Level 1

This book is not meant to cover C# in details but I will cover most of basics to help you get started.

5.1. What is .NET?

.NET Framework was developed to standardize programming. (Note how I am talking about the full .NET Framework and not the **Micro** Framework.) Once you program in .NET, you are no longer concerned of the underlying operating system. It offers a set of libraries that developers can use from many programming languages.

The .NET Framework runs on PCs and not on smaller devices, because it is a very large framework. Also, the full framework has many things (methods) 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 requires an operating system to run.

.NET Micro Framework is the smallest version of those frameworks. It removed more libraries and it became OS independent. Because of the similarity among these three frameworks, almost the 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 works the same way, when using .NET.

5.2. What is C#?

C and C++ are the most popular programming languages. C# is an updated and modernized version of C and C++. It includes everything you would expect from 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 brackets. In our first program, we only had one line of code between our 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 a 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#.

```
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

```
using System;
using Microsoft.SPOT;

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

while-loop

It is time for our first keyword, “while”. The while-loop starts and ends 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 a program that continuously prints “Amazing!” endlessly. This endless loop has no ending so it will always be “true”.

```
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 the “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 an 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 reserved for you. This memory will be referenced to as MyVar. You can give it any name you'd like, as long as the name doesn't contain spaces. Now, you can put any integer number into this memory/variable.

```
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
{
    public class Program
    {
```

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

Notice how the while-loop statement is not “true” anymore, but it is `MyVar>0`. This means keep looping as long as `MyVar`’s value is greater 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 the numbers 1 through 10. OK, we know how to make a variable and we know how to increment it but how do we print a number to the debug output window? Simply giving `MyVar` to `Debug.Print` will give you an error and it won't work. This is because `Debug.Print` will only accept strings, not integers. How do we convert an integer variable “ToString”? It is very simple, call `MyVar.ToString()`. That was easy!

```
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

```
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 developers to use the code but they don't have access to the assemblies source code. We have already used `Debug.Print`. Who made the `Debug` class/object and who made the `Print` method that is in it? Those are made by the 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 still 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, but what is a

namespace? Programs are split into regions or “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.

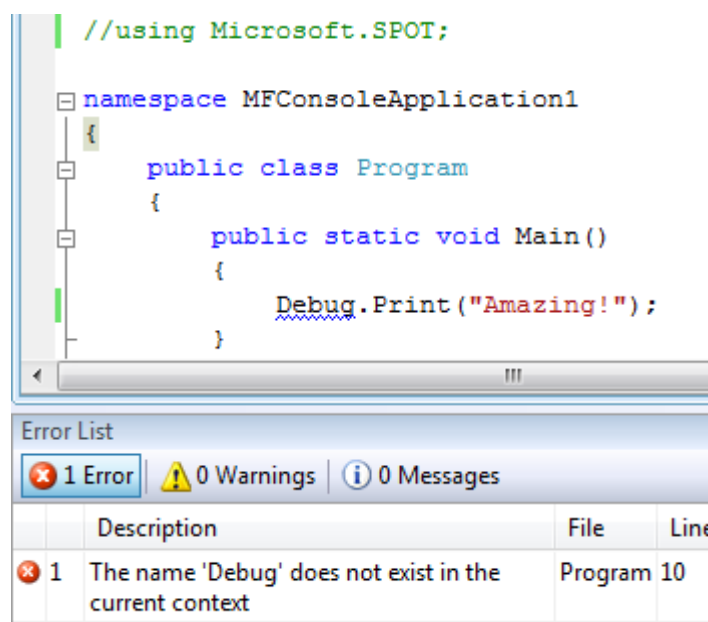
The “name” for our program's “space” is `namespace MFConsoleApplication1`

To “use” another name space like “Microsoft.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 backwards compatibility. In short, SPOT is NETMF!

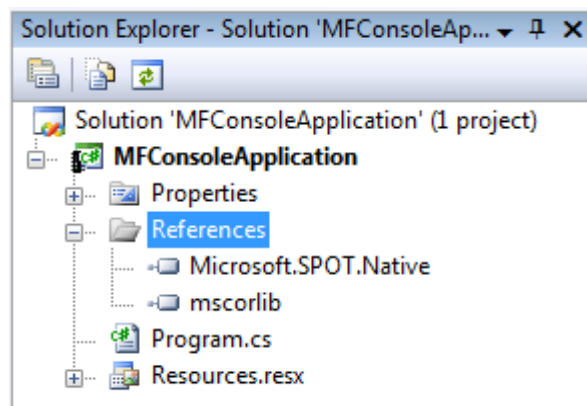
Back to coding, now try to remove or comment out `using Microsoft.SPOT;` and your code will not work anymore.

Here is the error message shown after I commented out `using Microsoft.SPOT;`

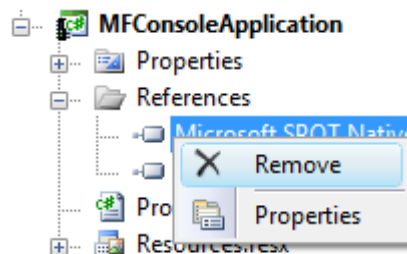


We used the assemblies but where are they added?

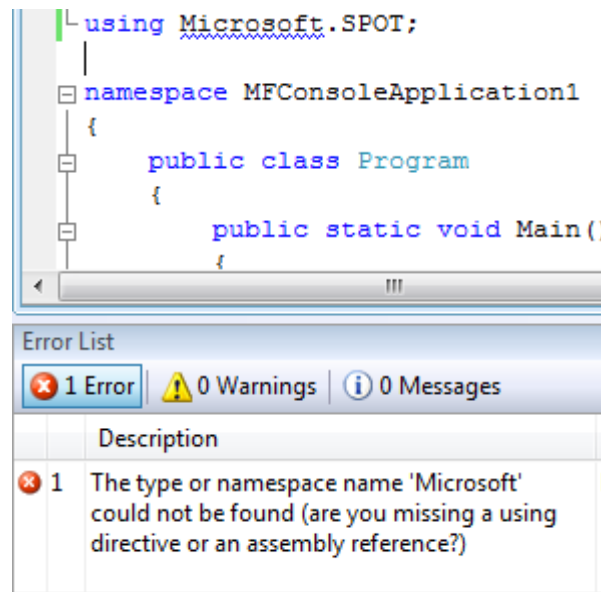
Uncomment the code and make sure it still works. Now take a look at the “Solution Explorer” window. Click the little + sign by the word “References” and you should see two assemblies.



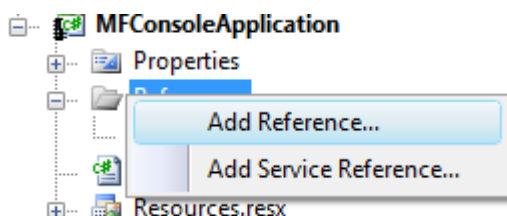
Now, right-click on “Microsoft.SPOT.Native” then click “Remove”



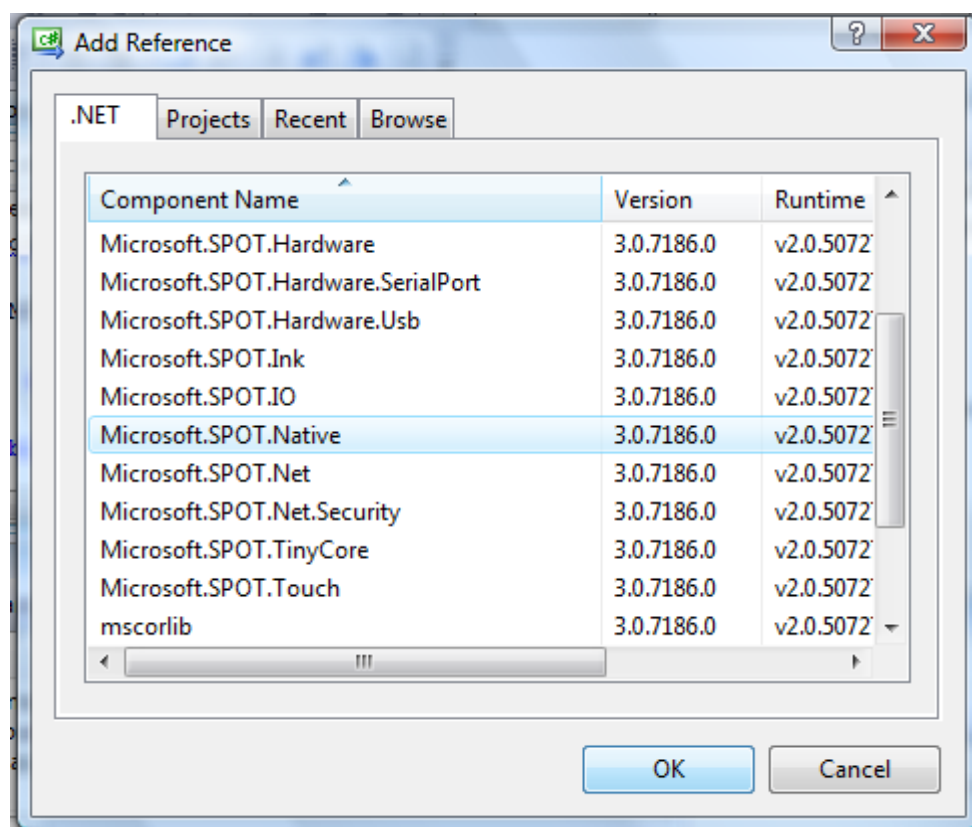
Our program will still be exactly the same as before, but now it's missing a very important assembly. Try to run it and you will see something like this



Let's add it back and make sure our program still runs. Right click on the “References” folder and select “Add Reference...”



In the new window, select “.NET” tab and then select “Microsoft.SPOT.Native” and click OK.



Run the program to make sure it's working again. If you have any errors, please go back and repeat the steps 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. These assemblies do not include the ones needed for networking or graphics.

Microsoft.SPOT.Hardware

Microsoft.SPOT.Native

Microsoft.SPOT.Hardware.SerialPort

Microsoft.SPOT.Hardware.OneWire

Microsoft.SPOT.Hardware.PWM

Microsoft.SPOT.IO

mscorlib

System

System.IO

If a sample code did not compile with an error about a missing assembly then please check the library documentation for the name of the assembly (DLL) needed.

Threading

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

Processors/programs only run one instruction at a time. 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 at once! What it is doing is running every program for a short time, 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 upcoming 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.

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);
```

To our While-loop.

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 an endless sleep.

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

```
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 second
                Thread.Sleep(1000);
            }
        }
        public static void Main()
        {
            // create a thread handler
            Thread MyThreadHandler;
            // create a new thread object
            //and assign 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);
        }
    }
}
```

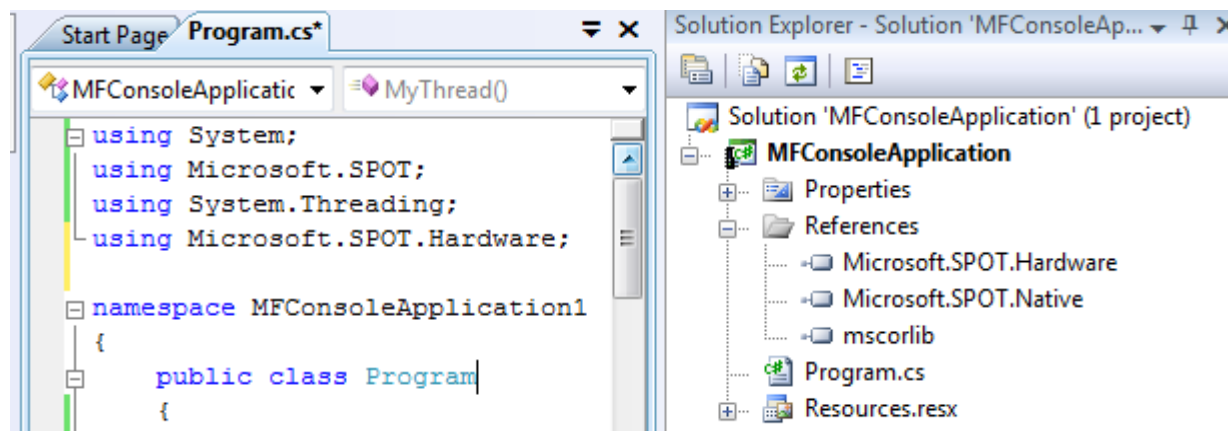
6. Digital Input & Output

On processors, there are many “digital” pins that can be used as inputs or outputs. by “digital” pins we mean the pin can be “one” or “zero”.

Important note: Static discharge from anything including the 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 equipment and take precautions handling the static charges in their body. You may not have such equipment so just try to stay away 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 the “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.

6.1. 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 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's 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's 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 controlled through an `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 pin number 4, the FEZ Domino on-board LED.

```
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);
        }
    }
}
```

The GHI SDK ships with special assemblies containing the pin mapping of your device. For example, “`FEZMini_GHIElectronics.NETMF.FEZ`” is needed for FEZ Mini.

Now, modify the code by adding “`using GHIElectronics.NETMF.FEZ`” at the top of your program. Your device may have a different DLL.

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

```
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 easier that is? 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 again. 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 it was off 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 the Write method on the OutputPort object. Note that you can't use "OutputPort.Write" This is very wrong because what output ports are you referring to? Instead, use "LED.Write" which makes complete sense.

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, a simpler way, to blink a LED.

```
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's 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 a long time and then it is off for a 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 a LED.

6.2. Digital Inputs

Digital inputs sense if the state of its pin is high or low. There is a 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 a pin, must be less than the

processor's power source voltage. All GHI Electronics boards use processors that run on 3.3V so the highest voltage the pin should ever see is 3.3V. This is true for ChipworkX and FEZ Hydra but not for EMX. EMX is 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 with are 5V. Being 5V tolerant allows us to use any of those digital circuits with our processor.

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

The `InputPort` object is used to handle digital input pins. Any pin on the processor's GHI use can be an input or an 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 an internal weak pull-down or pull-up resistor, 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. Many FEZ boards 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 the 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:

```
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);
        }
    }
}

```

6.3. Interrupt Port

If we want to check the status of a pin, we will always have to check its state periodically. This wastes the processor's 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 the 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: Most but not all pins on the processor support interrupts.

6.4. Tristate Port

If we want a pin to be an input and output, what can we 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 an input and read it.

NETMF supports better options for this, through a Tristate port. Tristate means three states; that is input, output low and output high. One minor issue about tristate pins is; if a pin is set to output and then you set it to output again, we will receive an exception. One way to get around this is by checking the direction of the pin before changing it. The direction of the pin is its “Active” property 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
{
    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 and 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 until you are comfortable with digital circuits.

7. C# Level 2

7.1. 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 statement

```
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 equality in C#. This is different from = which is used to assign values.

7.2. if-statement

An important part of programming is checking some state and taking 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). The LED needs to be checked repeatedly so we will do it once every 10ms.

```
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);
            }
        }
    }
}
```

7.3. if-else-statements

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

```
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 the if-statement and the else-statement in this example as a demonstration purpose. We can write the code this way.

```
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!

```
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.

7.4. 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 the object Debug which has a 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 the Debug object only takes one string argument. Other methods 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 the temperature.

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

The returned value can be an optional variable type. If there is no returned value 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, var3. Local variables are created inside a method and die once we exit the method. Now, 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 the number 123 is not the same as an integer containing 123. An integer is a number but a string is an array of characters that represent text or numbers. To humans these are the same thing, but in the computer world, this is totally different.

Here is the code to return a string.

```
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 changed to a string. We couldn't return var3 because it is an integer variable, 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 why we called a "ToString" method on a variable of type integer? 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.

```
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, until 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.

7.5. Classes

All objects we talked about so far are 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 the “walk” method in C# it is not clear if the cat or the human will walk but using `human.walk` or `cat.walk` makes it more clear.

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

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

7.6. Public vs. Private

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

Here is a quick example

```
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()
    {
    }
}
```

7.7. 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 this reference at this point is not referencing 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” of 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 value and 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 way.

7.8. 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;
```

7.9. Enumeration

An Enumeration is very similar to a constant. Let's say we have a device that accepts four

commands, those are MOVE, STOP, LEFT, RIGHT. This device is not a human, 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 what the command numbers are. 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);
```

8. Assembly/Firmware Matching

NETMF devices usually include extended features provided by the manufactures. It is very important that the firmware loaded on the device matches the assemblies being loaded from Visual Studio. **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 a very common issue that users run into when updating the firmware where the application just stops working and debugging seems to fail. You may see “checksum mismatch” in the output window.

Here is what happens:

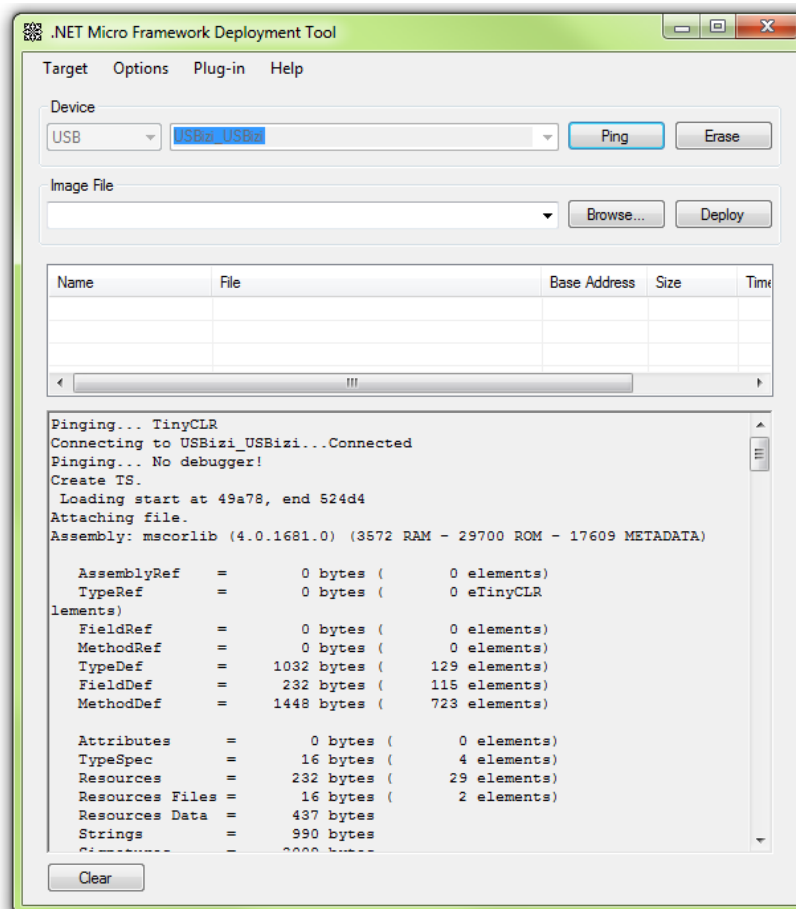
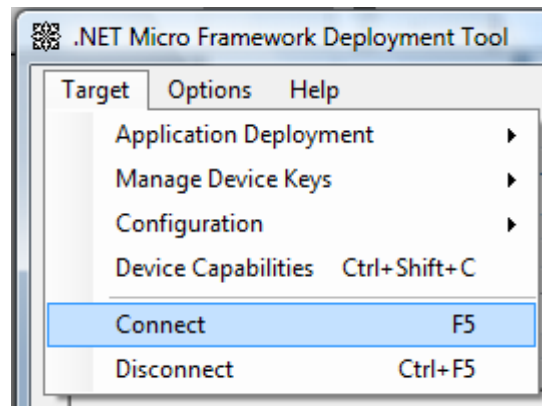
Scenario #1: A developer has received a new device. This device happens to have firmware version 1.0 on it. Then the developer went to the website and downloaded the latest SDK. The SDK has firmware version 1.1 in it. When trying to upload a project, VS2010 will fail to attach to the device with no indication why! The developer will now think the new device is not functioning, but actually, the device is just fine. 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 is still using 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.



9. 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 runs out of memory and probably crashes. Those bugs are usually very difficult to find in code.

Modern languages have garbage collector's that keeps track of used objects. When the system runs low on memory resources, the garbage collector jumps in and searches through all objects and frees the ones 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.

```
// 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.

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

9.1. Losing Resources

The garbage collector ease's object allocation but it can also cause problems if we are not careful. A good example would be on using digital output pins. Lets say we need a pin to be high. We create an `OutputPort` 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 a few minutes, the garbage collector kicks in and it finds this unreferenced object, so it will be removed. Freeing an `OutputPort` will cause the pin to change its state to input. Now, the pin is not high anymore!

```
// 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

```
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

```
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 provides a way to create timers that handle work after a determined time. If the reference for the timer is lost and the garbage collector runs, the timer is now lost and it will not run as expected.

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

9.2. Dispose

The garbage collector will free objects at some point but what 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” of it.

Disposing an 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);
```

9.3. 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);
```

10. C# Level 3

This section will cover all the 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>

10.1. Byte

We learned how int is 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 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.

10.2. 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 is called the 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 use 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 is good to store numbers from 0 to over 64,000. This 2-byte 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.

10.3. 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 a reference but it doesn't have an object yet, it is null. If you forgot what is a reference then go back and read more in the C# Level 2 chapter.

To create the object we use the “new” keyword and then we need to tell it the size of our array. This size is the count of how many elements of the type we will have in an array. Our type is a byte, 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.

```
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

```
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.

```
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
```

10.4. 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 human readable text. Because this is useful and a 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.

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

Whatever is 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 quotes (red text) but only takes it as it is; a string.

You may still have confusion on what the difference between an integer variable that has 5 in it and a string that has 5 in it is. Here is 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");
```

10.5. 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.

```
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 (“<=”)

```
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 = i + 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.

10.6. Switch Statement

You probably won't use the switch statement for beginner applications 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
{
    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("Wednesday");
                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-statements 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 shows:

```
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("Wednesday");
            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.

11. Additional Resources

Tutorials and Downloads

This page includes plenty of tutorials covering about everything NETMF. It also includes all needed links for downloads, references and any other links needed by NETMF users. You may want to bookmark this page.

<http://www.ghielectronics.com/support/dotnet-micro-framework>

Community: Forum, Codeshare and more

<http://www.tinyclr.com/> is the most active NETMF and gadgeteer website on the web. Its forum is very active with users ranging from expert system architects to hobbyists with very limited knowledge. Check it out today <http://www.tinyclr.com/forum>.

The website also include code share section which includes hundreds of code snippets and complete projects provided by the community. The source of the files is visible right on the website for those wanting to look for a “cheat sheet” or you can download the complete files as well. We look forward to see your own contributions <http://www.tinyclr.com/codeshare>

eBooks

- Beginners' NETMF Guide:
<http://www.ghielectronics.com/downloads/FEZ/Beginners%20guide%20to%20NETMF.pdf>
- Beginners' NETMF Porting Guide:
<http://www.ghielectronics.com/downloads/FEZ/Beginners%20Guide%20to%20Porting%20NETMF.pdf>
- Internet of things:
http://www.ghielectronics.com/downloads/FEZ/FEZ_Internet_of_Things_Book.pdf

Books

- Getting started with .NET Gadgeteer Book, by Simon Monk, O'Reilly
- Expert .NET Micro Framework, by Jens Kuhner, APress