Samraksh NETMF Emulator Tutorial

Author: Mukundan Sridharan

The Samraksh Company

Revision	Revision Date
1.0	May 20, 2012

Table Of Contents

1 Introduction

In this tutorial we will understand the design of the Samraksh's Micro Framework Emulator and will learn how to build and test application using it. We will first look at the hardware features and the design of the emulator and will subsequently go write two example applications, (i) a simple "hello world" kind of application where we will toggle LEDs using button presses and (ii) a more sophisticated application where we interact with a 'Physical Model' through the serial port to implement a Car and its Driver as two separate applications.

Note: The .Net Micro Framework is sometimes abbreviates to NETMF or sometimes just MF in this document.

1.1 Emulator Design Overview

The Samraksh's NETMF emulator is designed by extending Microsoft's .NET Emulator library. The two under lying design guidelines are:

1. Embedded applications interact with the physical work using sensors and actuators and hence for application developers to validate their program they must able to emulate both the embedded application and the environment in which it will work (the Physical World, or a small part of it)

2. The Interfaces of the embedded application should remain exactly the same as it would on the hardware thus validating its design

The unique feature of Samraksh's Emulator is, it lets the users write both .NET micro framework applications and the physical world model which interacts with the application, in order to validate the application. Thus the Emulator transparently reroutes the communication and sensing interfaces to the outside world, there by a developer can interact the MF application.

1.2 Emulator Hardware

The emulator currently supports the following hardware components

- 3 LEDS
    - Connected to CPU pins 0,1,2
    - Which can be accessed using Component Ids "led_0","led_1","led_2"
- 3 Buttons
    - Connected to CPU pins 3,4,5
    - Which can be accessed using Component Ids "button_0","button_1","button_2"
- 8 general purpose GPIO

    - Connected to CPU pins 6,7,8,9,10,11,12,13
    - Which can be accessed using Component Ids " gpio_0" through "gpio_7"
- 1 Serial Port
    - Which can be accessed using Component Ids "Emulator_COM1", this will be automatically routed to the Physical Model module.
- Timers
- Interrupts

1.3 Test Applications

In this tutorial we will write two test applications to demonstrate the two ways of interacting with the Emulator.

EmulatorTestApp: A simple application that toggles LED 1 when Button 1 is pressed, toggles LED 2 when Button 2 is pressed and toggles LED 3 when Button 3 is pressed. In this example, we will learn how to write a really simple Micro Framework application and to run it inside the emulator. We will also learn how to automate the input of the GPIO/Interrupts without manually pressing the button.

OpenLoopCarTest: Here we will learn how to built a physical model of the car as seperate .NET application and let it interact with the Micro Framework application running inside the Emulator thorough the serial port hardware interface. The MF application will act as the driver, which (somewhat blindly) drives the car implemented by the model application

The source code for the above applications can be found in the Release_0/SourceCode/MFApplications directory of Samraksh Release.

2 Installation

2.1 Requirements

Software Requirements: The only requirement for the emulator is a Microsoft Windows PC, running on a reasonably modern hardware

2.2 Software Installation

- Install C# visual studio 2008 express edition, from

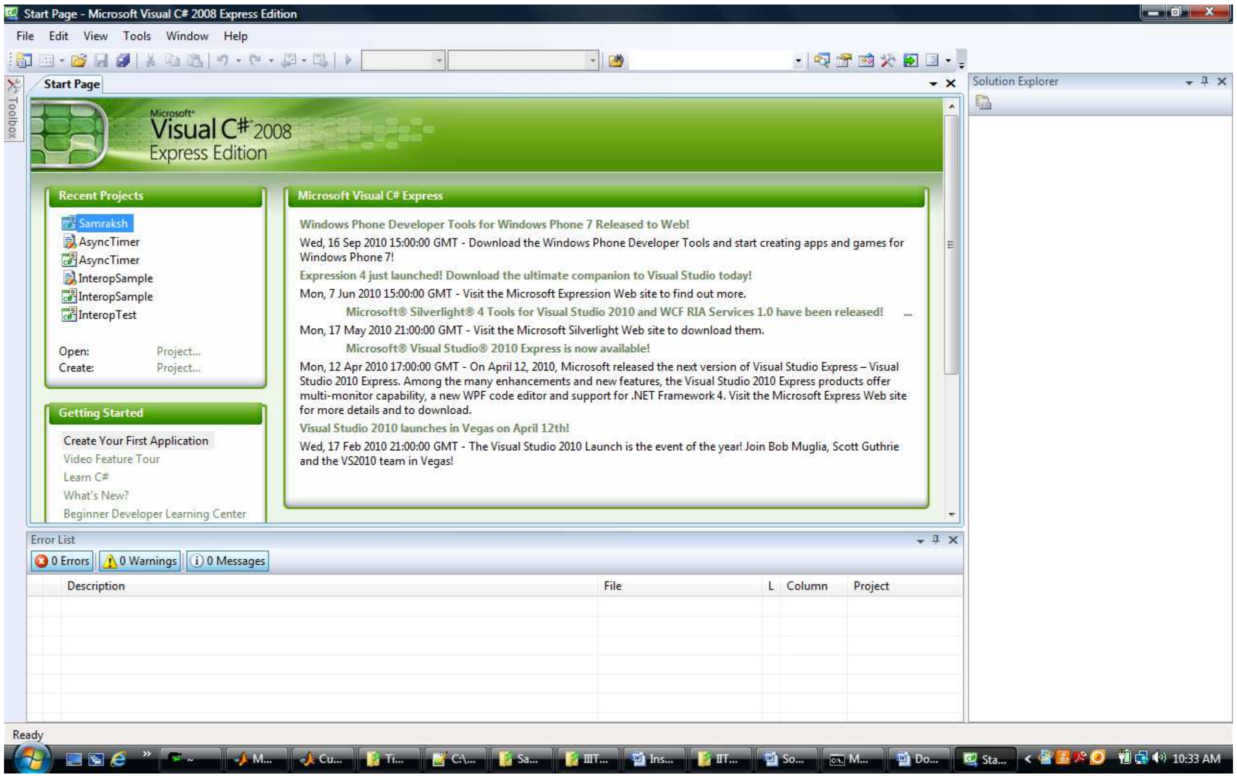
 http://www.microsoft.com/en-us/download/details.aspx?id=14597, or from http://download.microsoft.com/download/a/5/4/a54badb6-9c3f-478d-8657-93b3fc9fe62d/vcssetup.exe

- **Note:** It is important to stick to C# 2008 Express edition, C# Visual studio express 2010 does not work for MF application development.

- Install NETMF 4.0 sdk, available at , http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=23546

3 Building a Simple NETMF Application

3.1 *Create a new application in visual studio*



Windows Phone Developer Tools for Windows Phone 7 Released to Web!

Wed, 16 Sep 2010 15:00:00 GMT - Download the Windows Phone Developer Tools and start creating apps and games for Windows Phone 7!

Expression 4 just launched! Download the ultimate companion to Visual Studio today!

Mon, 7 Jun 2010 15:00:00 GMT - Visit the Microsoft Expression Web site to find out more.

Microsoft® Silverlight® 4 Tools for Visual Studio 2010 and WCF RIA Services 1.0 have been released! ...

Mon, 17 May 2010 21:00:00 GMT - Visit the Microsoft Silverlight Web site to download them.

Microsoft® Visual Studio® 2010 Express is now available!

Mon, 12 Apr 2010 17:00:00 GMT - On April 12, 2010, Microsoft released the next version of Visual Studio Express – Visual Studio 2010 Express. Among the many enhancements and new features, the Visual Studio 2010 Express products offer multi-monitor capability, a new WPF code editor and support for .NET Framework 4. Visit the Microsoft Express Web site for more details and to download.

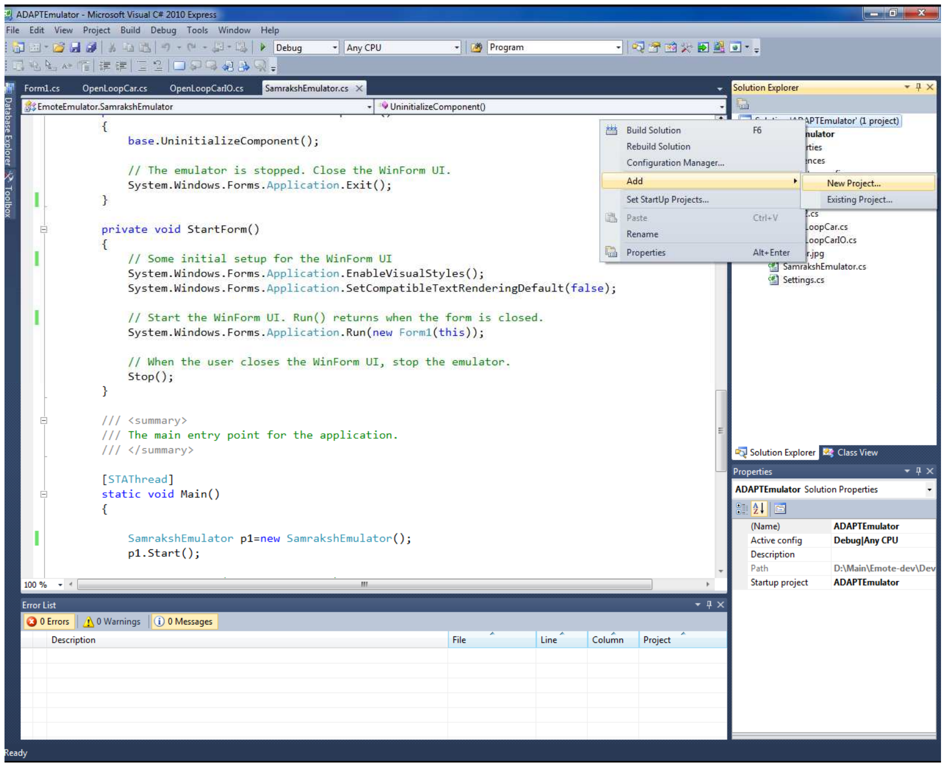
Visual Studio 2010 launches in Vegas on April 12th!

Wed, 17 Feb 2010 21:00:00 GMT - The Visual Studio 2010 Launch is the event of the year! Join Bob Muglia, Scott Guthrie and the VS2010 team in Vegas!

3.2 *Add new project to the Emulator solution*

- Right click on the solution and choose Add > New Project

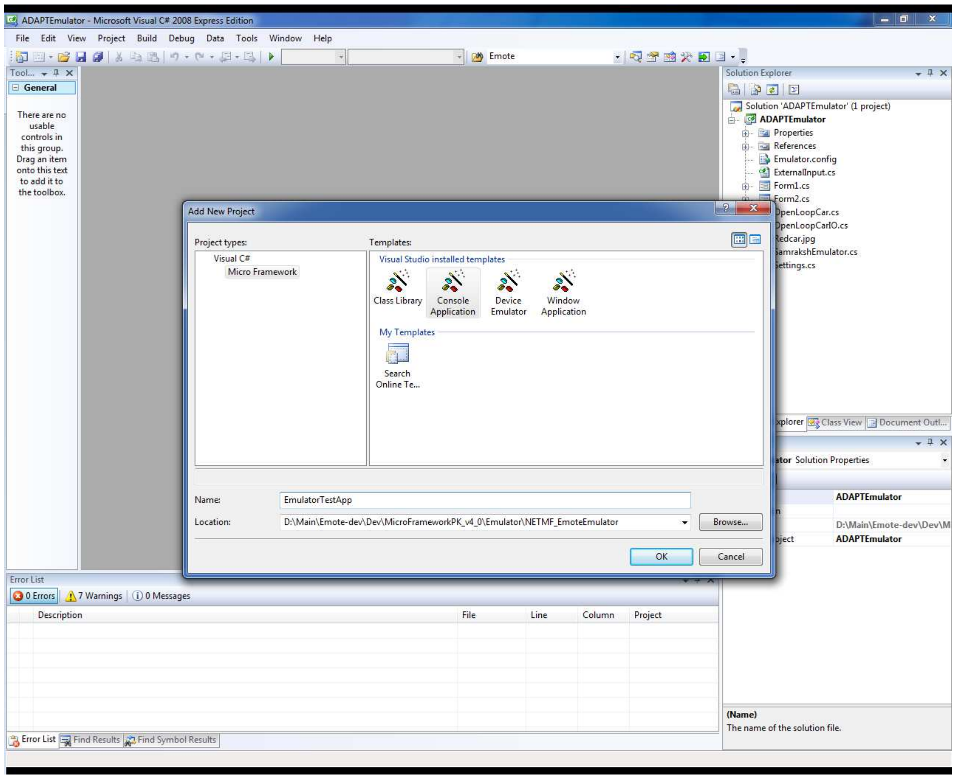
- Alternately you can open the ADAPTEmulator solution from source code under Release_0/SourceCode/ADAPTEmulator directory and add a new project to the solution. (This is option that is shown in the following Figures)



```
    {
        base.UninitializeComponent();

        // The emulator is stopped. Close the WinForm UI.
        System.Windows.Forms.Application.Exit();
    }

    private void StartForm()
    {
        // Some initial setup for the WinForm UI
        System.Windows.Forms.Application.EnableVisualStyles();
        System.Windows.Forms.Application.SetCompatibleTextRenderingDefault(false);

        // Start the WinForm UI. Run() returns when the form is closed.
        System.Windows.Forms.Application.Run(new Form1(this));

        // When the user closes the WinForm UI, stop the emulator.
        Stop();
    }

    /// <summary>
    /// The main entry point for the application.
    /// </summary>

    [STAThread]
    static void Main()
    {

        SamrakshEmulator p1=new SamrakshEmulator();
        p1.Start();
```

Properties	ADAPTEmulator Solution Properties
(Name)	ADAPTEmulator
Active config	Debug|Any CPU
Description	
Path	D:\Main\Emote-dev\Dev
Startup project	ADAPTEmulator

Choose the 'Micro Framework' under Project type and choose 'Console Application', and provide a name and a folder to the project . We will use EmulatorTestApp as the project name in this example.

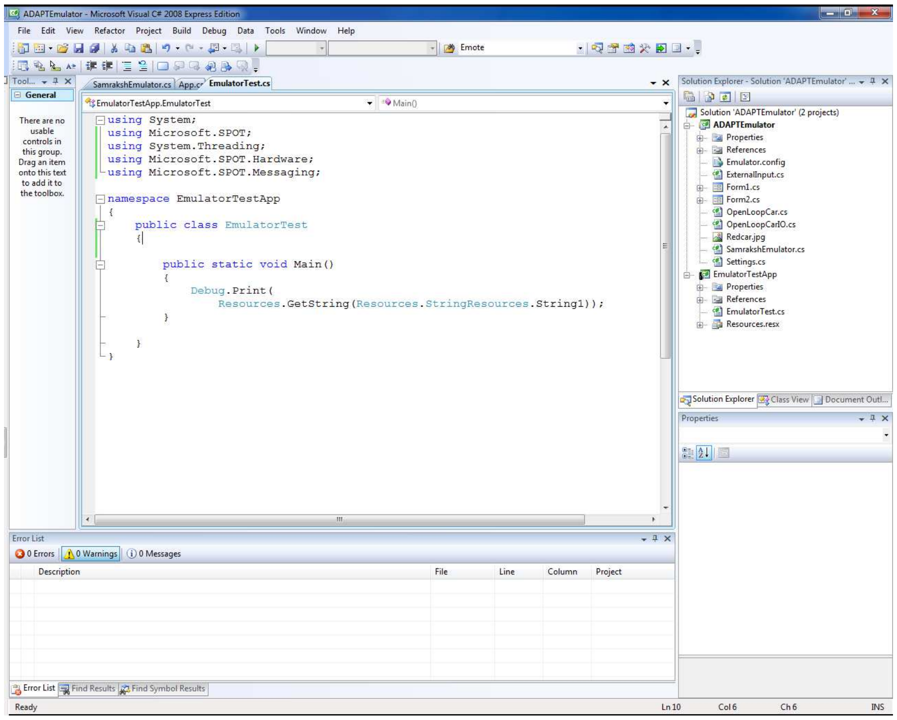


Name: EmulatorTestApp
Location: D:\Main\Emote-dev\Dev\MicroFrameworkPK_v4_0\Emulator\NETMF_EmoteEmulator

(Name)
The name of the solution file.

3.3 Write the Application Logic

Program.cs file is created for you which contains the application code. Lets rename this file as EmulatorTest.cs and also make sure the class name inside the file is changed.



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

namespace EmulatorTestApp
{
    public class EmulatorTest
    {

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

    }
}
```

Add References:

Import the necessary libraries and also right click on 'References' under the EmulatorTestApp and add references to these libraries

```
using System;
using Microsoft.SPOT; //Provides basic framework
using System.Threading;  //Needed for threads
using Microsoft.SPOT.Hardware; //Needed for accessing hardware
using Microsoft.SPOT.Messaging; //
```

Add the hardware interfaces and variables needed by the application:

Under the EmulatorTest class, lets first add the InterruptPorts and OutputPorts and three boolean variables to keep track of the state of the buttons.

```
InterruptPort _button1_InterruptPort, _button2_InterruptPort,
_button3_InterruptPort;
static OutputPort _led1_port, _led2_port, _led3_port;
//Boolean variables to keep tract of the state of the buttons
```

```
static bool button1_state, button2_state, button3_state;
```

Create a Constructor:

Next lets add a constructor method to the EmulatorTest class and instantiate 3 InterruptPort objects and 3 OutputPort objects as shown below. Make sure that the output/LEDs are connected to cpu pin 0,1,2 and the button/interrupt ports are connected to cpu pins 3,4,5

```
public EmulationTest()
{
    //Instantiate the Output ports
    _led1_port = new OutputPort((Cpu.Pin)0, false);
    _led2_port = new OutputPort((Cpu.Pin)1, false);
    _led3_port = new OutputPort((Cpu.Pin)2, false);

    //Instantiate the interrupt ports
    _button1_InterruptPort = new InterruptPort((Cpu.Pin)3, false,
Port.ResistorMode.PullDown, Port.InterruptMode.InterruptEdgeBoth);
    _button2_InterruptPort = new InterruptPort((Cpu.Pin)4, false,
Port.ResistorMode.PullDown, Port.InterruptMode.InterruptEdgeBoth);
    _button3_InterruptPort = new InterruptPort((Cpu.Pin)5, false,
Port.ResistorMode.PullDown, Port.InterruptMode.InterruptEdgeBoth);

    //Connect the interrupts to their handler methods
    _button1_InterruptPort.OnInterrupt += new
NativeEventHandler(button1_OnInterrupt);
    _button2_InterruptPort.OnInterrupt += new
NativeEventHandler(button2_OnInterrupt);
    _button3_InterruptPort.OnInterrupt += new
NativeEventHandler(button3_OnInterrupt);
}
```

Write the Interrupt Handlers:

Next let us add the three methods for the button's interrupt handlers. The interrupt handlers toggles the state of the boolen variables keeping track of the button state and then write the button state to the corresponding LEDs. Thus as the state of the button toggles, the LEDs also toggle their state.

```
static void button1_OnInterrupt(uint data1, uint data2, DateTime time)
{
    //Toggle button1_state
    if (button1_state) button1_state = false; else button1_state = true;
    _led1_port.Write(button1_state); //write the state to LED1
    Debug.Print("Button 1" + ((button1_state) ? "Down" : "Up"));
}

static void button2_OnInterrupt(uint data1, uint data2, DateTime time)
{
    //Toggle button2_state
    if (button2_state) button2_state = false; else button2_state = true;
    _led2_port.Write(button2_state); //write the state to LED2
    Debug.Print("Button 2" + ((button2_state) ? "Down" : "Up"));
}

static void button3_OnInterrupt(uint data1, uint data2, DateTime time)
{
    //Toggle button3_state
    if (button3_state) button3_state = false; else button3_state = true;
    _led3_port.Write(button3_state); //write the state to LED3
    Debug.Print("Button 3" + ((button3_state) ? "Down" : "Up"));
}
```

Write the Main:

And finally lets instantiate the class in the main and put the main thread to a sleep forever, so that the application will never quit and will keep awaiting for user input through the buttons.

```
//Application starts here
public static void Main()
{
    EmulatorTest e = new EmulatorTest();
    Thread.Sleep(Timeout.Infinite);
}
```

Your application is not complete and you are ready to execute your application (Section 4).

3.4 Automating the input to GPIO pins

The Samraksh's Emulator has a built-in input automation module using which inputs to any of the GPIO pins can be sent automatically based on a timer. The input automation module reads a text file specified in a particular format and sends the input to appropriate GPIO pins at the write time. The format of the input file is:

Time (in ms), Node_ID, GPIO_pin, Input

Apart from the input, the automation module also expects a first line to be a configuration input, which specifies if the time values are absolute or relative. If time values are absolute, the input values are sent at those specified times, starting from the moment when input automation is started. If they are relative, the values are treated as an delay before the injection of that input.

In our case we have the buttons are connected to GPIO pins 3,4 and 5 and are configured as interrupts. Thus in order to automate button presses we want to send input signals to the pins 3,4 and 5 in whatever order we want them to be activated with a particular delay between actuations. Below is an example input file, which sends a 1 and 0, alternately to the cpu pin 3 (connected to button1) with a 2 second delay between acuations.

Note: The second field Node_ID is ignored currently, has no effect on the automation since network emulation is not supported yet.

```
Time:Absolute
1000,1,3,1
3000,1,3,0
5000,1,3,0
7000,1,3,1
```

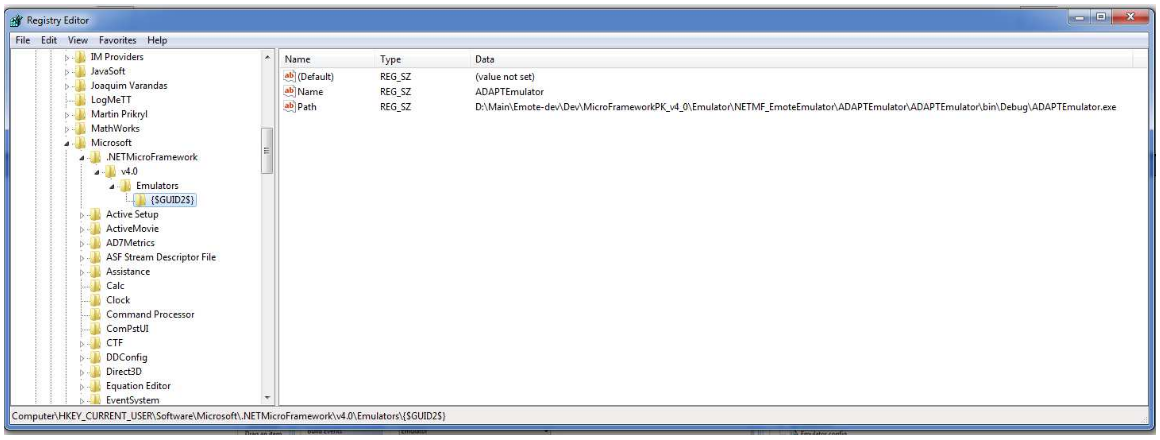
In order to automate the gpio input, all you need to do is to click the browse button in the Input Automation section in the control panel and select the input text file, and then click the 'Start Input Automation' button.

4 Running the Emulator Application

4.1 Registering you Emulator Binary

When you build Emulator application in Visual Studio or Visual Express, the Visual Studio automatically registers the Emulator with the OS, so that when you build MF applications Visual Studio will automatically pickup the list of Emulator available in the system. But sometime you dont want to build your own Emulator and use directly the binary (or the source code is not available to you). In that case you might want to register the Emulator binary. To do this just add the following registry keys as shown in the images below:

```
HKEY_CURRENT_USER\Software\Microsoft\.NETMicroFramework\v4.0\Emulators
```

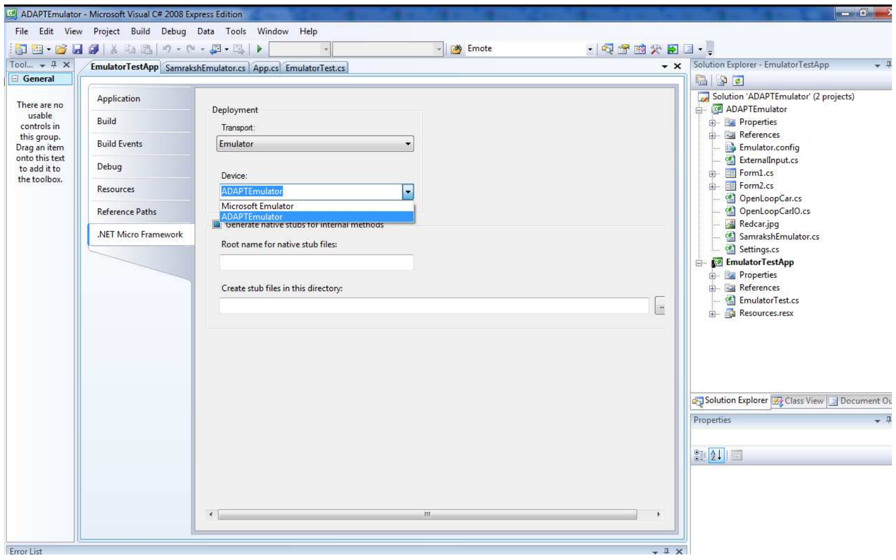


Name	Type	Data
(Default)	REG_SZ	(value not set)
Name	REG_SZ	ADAPTEmulator
Path	REG_SZ	D:\Main\Emote-dev\Dev\MicroFrameworkPK_v4_0\Emulator\NETMF_EmoteEmulator\ADAPTEmulator\ADAPTEmulator\bin\Debug\ADAPTEmulator.exe

Computer\HKEY_CURRENT_USER\Software\Microsoft\.NETMicroFramework\v4.0\Emulators\{$GUID2$}

4.2 Configuring the Emulation Device

Before running the emulation you want to make sure to select the correct device on which to run your application. The NETMF platform comes with a default emulator called the 'Microsoft emulator', you however want to make sure to select 'ADAPT Emulator' developed by Samraksh. To do this Right click on the EmulatorTestApp project and go to properties. Under properties go to .NET Micro Framework and under device the default option will be set to 'Microsoft Emulator'. Change that to 'ADAPT Emulator' as shown in the below Figure.



Deployment

Transport: Emulator

Device: ADAPTEmulator

Root name for native stub files:

Create stub files in this directory:

4.3 Running Emulation from Visual Studio

To run you application just press F5 or the run button in the Visual Studio/Express IDE. A Emulator Control Panel will open up, from which you can control the execution.

4.4 Running Emulation from command line

You can also run the emulator and the application from the command line. This involves the following steps

1. Copy framework dlls: Copy the following framework dll from the .NET SDK installation to the execution directory. The required dlls are Microsoft.SPOT.CLR.dll, Microsoft.SPOT.Emulator.dll, Microsoft.SPOT.Emulator.Interface.dll.

2. Next you will need to copy the ADAPT Emulator executables to the execution directory, which include the ADAPTEmulator.exe, ADAPTEmulator.exe.emulatorconfig and Microsoft.SPOT.Emulator.Sample.SampleEmulator.exe .

3. Next you need to copy the *.pe files of the MF application and applications refernce libraries to the directory. For example, for the EmulationTestApp, we need to copy the

EmulationTestApp.pe, its references mscorlib.pe, Microsoft.SPOT.Hardware.pe and Microsoft.SPOT.Native.pe

4. Finally you can start the emulator from the command line using the following command or put the command to a batch file (add the command to text file and rename the file extension to *.bat). A example batch file called 'RunTest.bat' can be found under the bin directory in the Samraskh Emulator Release.

```
"ADAPTEmulator.exe"  "/load:EmulatorTestApp.pe" "/load:mscorlib.pe"
"/load:Microsoft.SPOT.Native.pe"  "/load:Microsoft.SPOT.Hardware.pe"
```

5 Emulating the Real World using a Physical Model

The idea of an emulator is to verify the program logic of a particular application. But an embedded application interacts closely with the external world through its sensors and actuators. Hence, emulating the real world which interacts with the Microframework Application becomes a nessacity. The Samraksh's MicroFramework Emulator is designed to enable such an interaction possible. The Emulator exposes key hardware interfaces of the application to a Physical Model application, which is written as a seperate application interacting with the MF application. The interaction between the Emulator and the Physical Model application is through regular TCP/IP sockets, which means the Physical Model can be in any language and can run even on a different machine connected to a network.

Further more, the Emulator is designed in such a way that there need to be no changes to the way the Micro Framework application is written for the Emulator. Communication interfaces such as Serial Port, SPI, I2C ,USB and Radio are routed transperently to the Physical Model application running as a seperate application.

5.1 *Physical Model <-> Emulator Interface*

At its simplest the interface between the Model and the Emulator is simply a packet interface with a single byte header, which indicates what is the communication interface for which the rest of the payload is intended. The following Figure shows the values of the header/interfaces that are currently supported.

```
InterfacePacket{
    Header, //1bytes
    Payload
}

InterfaceType: Header Values
USART: 1
SPI: 2
I2C: 3
USB: 4
```

A user can use the above packet format to send and receive messages to Emulator and there by interact with a MF application running inside the Emulator. The Emulator runs a TCP/IP server on the port (default port number is **45000**) passed as a parameter to the Emulator. A user can write a Physical Model application which connects to the Emulator port and sends and receives messages to the MF application. Depending on the first byte of the messages received by the header the Emulator, the emulator will route the messages to appropriate hardware interface of the application. (**Note:** We currently support only one port of each type of interface. This restriction will be removed in the next release.)

5.2 PhysicalModelEmulationComm Class

Even though write a socket communication application can b The interface between the Physical Model and Emulator is provided by a Interface called In order to make writing Physical Models easy, we provide the users with a C# class called the 'PhysicalModelEmulatorComm' which provides simple APIs to communicate with the Emulator. The details of this class are given below.

Namespace: Samraksh.PhysicalModel

Class: PhysicalModelEmulationComm

Members:

Public Constructors:

Name	**Description**
`PhysicalModelEmulationComm ()`	Creates a new a new communication object and connects to the Emulator Server on the default port (45000) and IP address (127.0.0.0)
`PhysicalModelEmulationComm (int port, String IpAddress)`	Creates a new a new communication object and connects to the Emulator Server port and IPAddress provided as parameters

Delegates:

```
delegate void InterfaceCallback(byte[] Message);
```

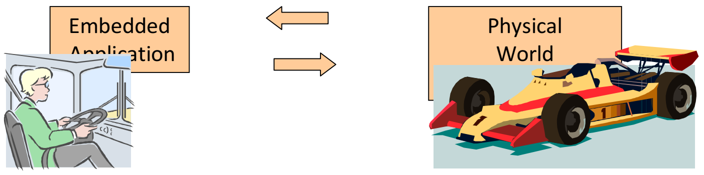
Public Methods:

Name	**Description**
`void InitializeSerial( InterfaceCallback Handler)`	Initializes the serial port communication with the Emulator with the method Handler is called anytime a message arrives on the serial port.

`void InitializeSPI( InterfaceCallback Handler)`	Initializes the SPI communication with the Emulator with the method Handler is called anytime a message arrives on the serial port.
`void InitializeI2C( InterfaceCallback Handler)`	Initializes the I2C port communication with the Emulator with the method Handler is called anytime a message arrives on the I2C interface.
`void InitializeUSB( InterfaceCallback Handler)`	Initializes the USB port communication with the Emulator with the method Handler is called anytime a message arrives on the USB interface.
`void SendToSerial(byte[] buffer)`	Send the message in the byte array buffer, to the emulator, to be sent to the MF application through the serial port
`void SendToSPI(byte[] buffer)`	Send the message in the byte array buffer, to the emulator, to be sent to the MF application through the SPI interface
`void SendToI2C(byte[] buffer)`	Send the message in the byte array buffer, to the emulator, to be sent to the MF application through the I2C interface
`void SendToUSB(byte[] buffer)`	Send the message in the byte array buffer, to the emulator, to be sent to the MF application through the USB interface

5.3 *The Open Loop Car Model Example*

In this Section we will write a physical model for a simple open-loop car and then we will write a corresponding Micro Framework application which acts as the Driver for the car. We will use the Samraksh's 'PhysicalModelEmulationComm' class to while writing the physical model application to enable the interaction between the interaction between the model and the Emulator. The Figure below gives a pictorial representation of the Physical Model and the MF Application.



5.3.1 Physical Model Application

We will develop a Open Loop Car model, with a 2D animation to visualize the movement of the car. The model takes the car steering wheel angle (with respect to the rest of the car) and the acceleration of the car as input parameters and calculates the location of the car in a 2D plane, its velocity and direction (with respect to the plane). The car has a initial velocity and the steering wheel direction is zero, hence it keep moving straight at a constant velocity unless there is an input from the MF Driver application. The velocity varies linear when there is a change in the accelaration, untill it reaches a maximum velocity or zero velocity. The physical model computes the trajectory of the car either on a straight line (when the wheel angle is zero) or on a circle when the wheel angle is non-zero.

1. **Create a new project:** Open Visual C# Express and start new project | Select Visual C# under Project Types and Select Windows Forms Application under Templates. Lets call our project OpenLoopCarModel

2. **Rename files and save:** A Form called Form1 will created for you and a Class called Program (in file Program.cs) will also be created for you. Lets rename the file Program.cs to OpenLoopCar.cs (and also make sure the class name is changed to OpenLoopCar). Next its save the solution as OpenLoopCarModel.sln

3. **Add the PhysicalModelEmulatorComm class:** Right click on the project | Add | Existing Item| Select tht PhysicalModelEmulatorComm.cs file (provided by Samraksh) and click ok.

3b. Alternately right click on the References and add the PhysicalModelEmulatorComm.dll provided by Samraksh.

4. **Initialize emulator communication:** Create a function called void HandleSerialInput(byte[] input), which will be called by the communication module anytime a message is sent by the MF application through the serial port.

Next lets declare an instance of the communication module called emulatorCom in the OpenLoopCar class. Lets also create a constructor for the class and inside that lets instantiate the communication module. Next lets register the callback function for the serial port with the emulator communication object using the 'InitializeSerial' method and passing the 'HandleSerialInput' method as the parameter. And finally lets connect to the emulator by calling the ConnectToEmulator method as shown in the code section below.

```
PhysicalModelEmulatorComm emulatorCom;

public void HandleSerialInput(byte[] input)
{
}

public OpenLoopCar()
{
   emulatorCom = new PhysicalModelEmulatorComm();
   emulatorCom.InitializeSerial(HandleSerialInput);
   emulatorCom.ConnectToEmulator();
}
```

5. **Write the model:** Next lets write the actual model for the car.

Application Logic: We will start a free-running timer for (lets say) 1sec, which will fire once every second. Each time when the timer fires the next location of car will be recomputed, given the cars current location, wheel angle and velocity. And after the location is computed we will call the visualization app to plot the new course of the car on screen. The function to start the model is shown below, which initializes the timer and also instantiates the form for visualization.

```
public void Start()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    f1 = new Form1();
    curX = f1.Width / 2;
    curY = f1.Height - 50;
    locationUpdateTimer = new System.Windows.Forms.Timer();//declared
            //under the class
    locationUpdateTimer.Interval = 1000;
    locationUpdateTimer.Tick += new
         EventHandler(locationUpdateTimer_Tick);
    locationUpdateTimer.Start();
    Application.Run(f1);
}
```

We will skip the rest of the actual model in the tutorial and the implementation of the animation to visualize the car, but the code for the model and animation is provided under the 'PhysicalModels' directory of the under SourceCode. (After all, the user is supposed to write the model).

Even though in this example we are not sending any (sensing) messages to the application, all the necessary plumbing to send a message through the serial port already exist. For example in order to send a message to the MF application, we can use the 'SendToSerial' method of the communication library as shown in code segment below.

```
void SendMessageToEmulator(byte[] message)
{
    emulatorCom.SendToSerial(message);
}
```

Exercise: One exercise for the user to learn the emulator could be to extend our Open Loop Car model to give feedback to the MF application about the boundaries and obstacles in the field through the serial port, so that the driver program can avoid the boundaries and obstacles.

5.3.2 Micro Framework Application

In this Section we will write the Micro Framework application that will act as the driver for the car. The driver application will user the serial port to send any changes in the steering wheel angle or the acceleration.

1. **Create new project:** Open a new project in the C# Visual Studio Express, Select Micro Framework under 'Project Types' and choose 'Console Application'. Lets name our project

'CarDriver' and click ok. A class called Program will be created for you in a file called Program.cs, which contains the main application.

1a. Alternately open the ADAPTEmulator solution and add a new project to the solution.

2. Import necessary libraries: Lets add the following libraries that we will need for this application

```
using System;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using Microsoft.SPOT.Messaging;
using System.Threading;
using System.IO.Ports;
```

3. Input and Output packet formats: Lets first define the input and output interfaces for the application.The application will send the SteeringAngle, the Acceleration and any special commands called the 'Command' to the model and can read the models output which are the location X,Y, the speed of the car and the direction of the car. (Note: we do not read the output of the model in this example, but all necessary plumbing is available in the emulator). Currently we have defined just two special commands 'START' and 'STOP'

```
enum CarCommands
{
    START, STOP
};

class CarModelInput
{
    public short SteeringAngle; //-128to127
    public sbyte Accelaration;//int8 -63 to 63
    public byte State; //Special commands
}

class CarModelOutput
{
    public ushort X; //0-65535
    public ushort Y; //0-65535
    public byte speed;//0-255
    public short direction; //degrees
}
```

4. Declare variables: Lets declare the global objects we will use in the application. We will need a serial port to communicate with the model and a carInput object to be sent over the serial port

```
SerialPort phyiscalModelPort;
CarModelInput carInput;
```

5. Initialize serial communication: Next, lets create a contructor for the Program class in that we will instantiate and initialize the SerialPort

```
Program()
{
    carInput = new CarModelInput();
    try
    {
        phyiscalModelPort = new SerialPort("COM1");
    }
    catch (Exception e)
    {
        Debug.Print(e.ToString());
    }
}
```

6. Sending and receiving through the serial port: The code segment below shows how to send and receive messages through the serial port, which we initialized in the previous step. The ReadModelOutput method needs to called periodically from a timer to check if there are packets to be handled. (Even dont have this piece of code, since in our application we dont read from serial port). If there are any messages you can subsequently process them using the ProcessInput method (Again not shown in the code).

```
Void SendModelInput()
{
    byte[] buffer = new byte[4];
    ShortToBytes(carInput.SteeringAngle, buffer);
    buffer[2] = (byte)carInput.Accelaration;
    buffer[3] = carInput.Command;

    if (!phyiscalModelPort.IsOpen)
    {
        phyiscalModelPort.Open();
    }

    int n = phyiscalModelPort.Write(buffer, 0, buffer.Length);
    if (n <= 0)
    {
        Debug.Print("Failure: " + n.ToString());
    }
    else
    {
        Debug.Print("Success, sent: " + n.ToString());
    }
}

void ReadModelOutput(object state)
{
    if (!phyiscalModelPort.IsOpen)
    {
        phyiscalModelPort.Open();
    }
    byte[] readBuffer = new byte[100];
    int bytes_read = phyiscalModelPort.Read(readBuffer, 0, 100);
    if (bytes_read > 0)
        {
            ProcessInput(readBuffer,bytes_read);
        }
}
```

7. Fill the application logic: Now we start filling out the actuall application logic which is simply sending steering wheel angles at appropriate times. We will write 3 simple methods for Starting, Stoping and Turning (as shown below in the code segment) which we will use to write a TestDrive method. The 'Turn' method takes as input the number of degress (remember 360 degrees is a full circle) by which to turn the steering wheel. A positive angle means turning left and a negative angle means turning right. In each of above driving methods, we will fill up the carInput packet with the right information and will call the SendModelInput method to send it via the serial port.

```
void Start()
{
    carInput.Command = (byte)CarCommands.STOP;
    SendModelInput()
}
void Stop()
{
    carInput.Command = (byte)CarCommands.STOP;
    SendModelInput();
}
void Turn(short degrees)
{
    carInput.SteeringAngle = degrees;
    carInput.Command = 0;
    carInput.Accelaration = 0;
    SendModelInput();
}
```

Now lets write the TestDrive method, which will drive the car (somewhat blindly) for about a minute and will then stop.

```
void TestDrive()
{
    Start();
    Thread.Sleep(8000); //Drive Straight for 5s

    Turn(20);
    Thread.Sleep(5000); //Slight Left for 3s
    Turn(-20);

    Thread.Sleep(5000); //Drive Straight for 5s

    Turn(-45); //Turn right (almost u-turn) 13s
    Thread.Sleep(7000);
    Turn(45);

    Thread.Sleep(5000); //Drive Straight for 10s

    Turn(45); //Turn left for 5s
    Thread.Sleep(9000);
    Turn(-45);

    Thread.Sleep(4000); //Drive Straight for 5s

    Turn(-45);//Turn right for 10s
    Thread.Sleep(7000);
    Turn(45);

    Thread.Sleep(6000);//Drive Straight for 10s
    Stop();
}
```

7. Write the Main: Finally, lets write the main. In the main lets first instantiate the application Program. We will wait for 3 seconds for the physical model to connect to the emulator. In the current version of the emulator it is assumed that the physical model is connected by this time. We donot do any sophisticated staging check to see if it actually connects. The emulator is likely to hang if the physical model does not connect within this time. Next lets call the 'TestDrive' method to drive the car and will finally sleep for 10 secs before exiting the application.

```
public static void Main()
{
    Program P = new Program();
    Thread.Sleep(3000); //Wair for 3secs for the model to connect to
                        //the emulator
    P.TestDrive();      //Start the driving
    Thread.Sleep(10000); //Sleep for 10 secs then exit.
}
```

8. Run the emulation: Run the application as discussed in Section 4.

6 Time inside the Emulator

Functions like Timers and Sleep do run in real time, that if you Sleep for 1000ms you will exactly sleep for 1000ms. However commutations inside the emulator happen "as fast as possible" and might not confirm to the commutation on the real hardware.

7 Network Emulation

The current version of the emulator does support any at network-level emulation. Our next release will have support for a Radio hardware interface and a networking framework. However users could possibly run a network emulation by running multiple instances of the emulator and networking them on their own.

8 Support

Please email support@Samraksh.com with the subject "Emulator:" followed by a optional subject title. This tutorial and a manual detailing the design of the Emulator is available online from www.Samraksh.com

9 References

1. Microsoft Micro Framework Emulator Overview and Reference.
   http://msdn.microsoft.com/en-us/library/ee433256.aspx

2. Running Emulator from Command line.
   http://bloggingabout.net/blogs/jens/archive/2008/12/08/standalone-net-micro-framework-emulator.aspx