Samraksh NETMF Emulator Tutorial

Author: Mukundan Sridharan

The Samraksh Company

Revision	Revision Date
0.0	May 20, 2012
0.1	June 20, 2012
0.2	July 13, 2012

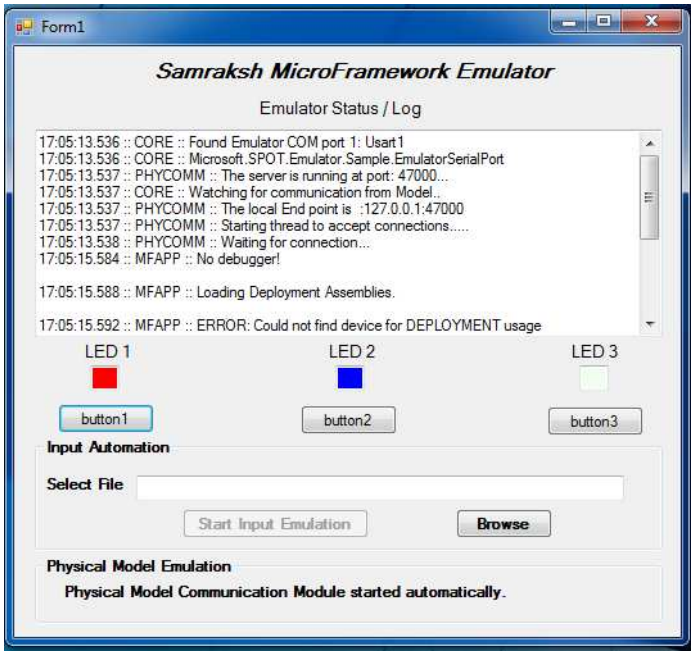
Table Of Contents

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. The users can control various features of the emulator from the Emulator Control Panel, a snapshot of which is show above .

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

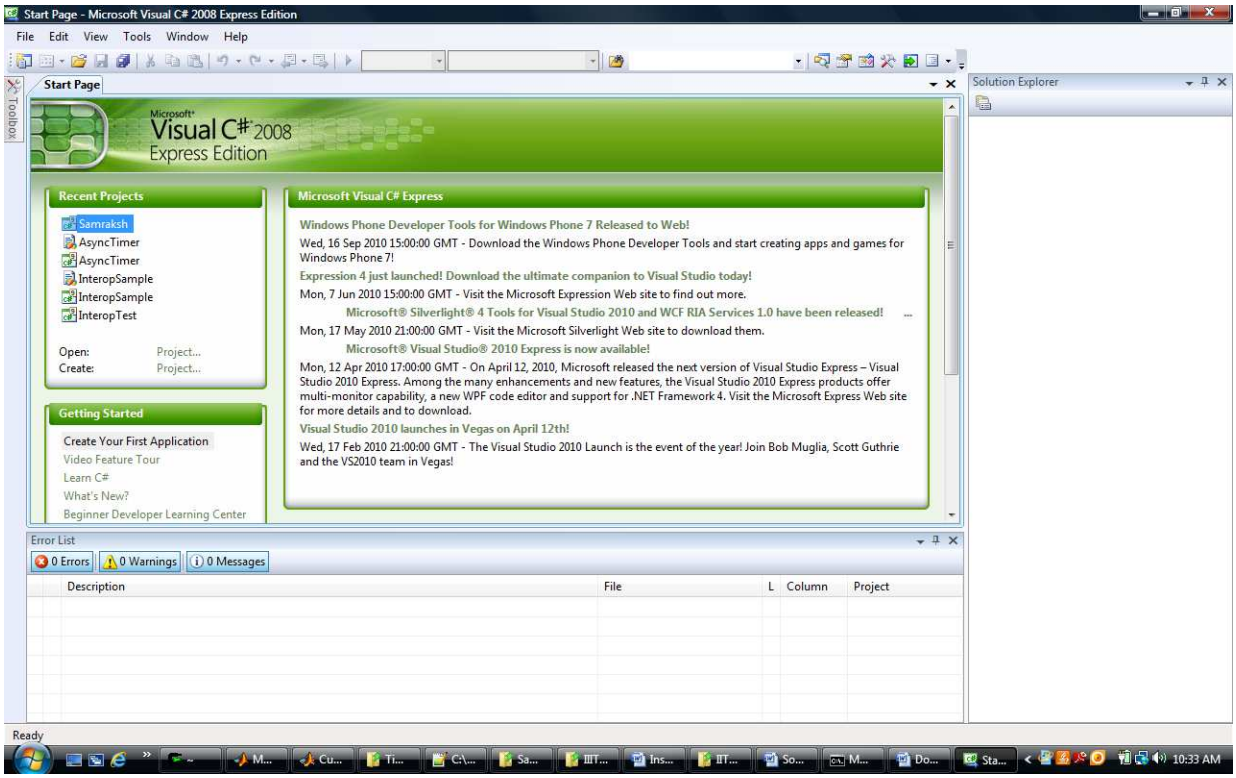
  http://www.microsoft.com/en-us/download/details.aspx?id=14597, or from http://download.microsoft.com/download/a/5/4/a54badb6-9c3f-478d-8657-93b3fc9fe62d/vcssetup.exe

- **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

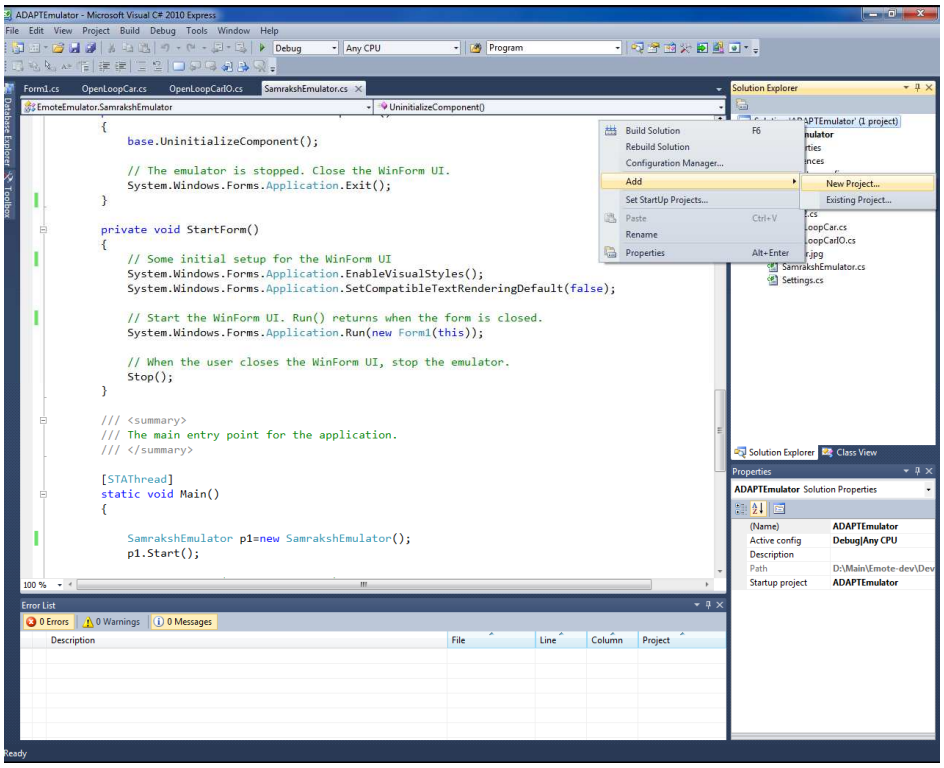
3 Building a Simple NETMF Application

3.1 Create a new application in visual studio

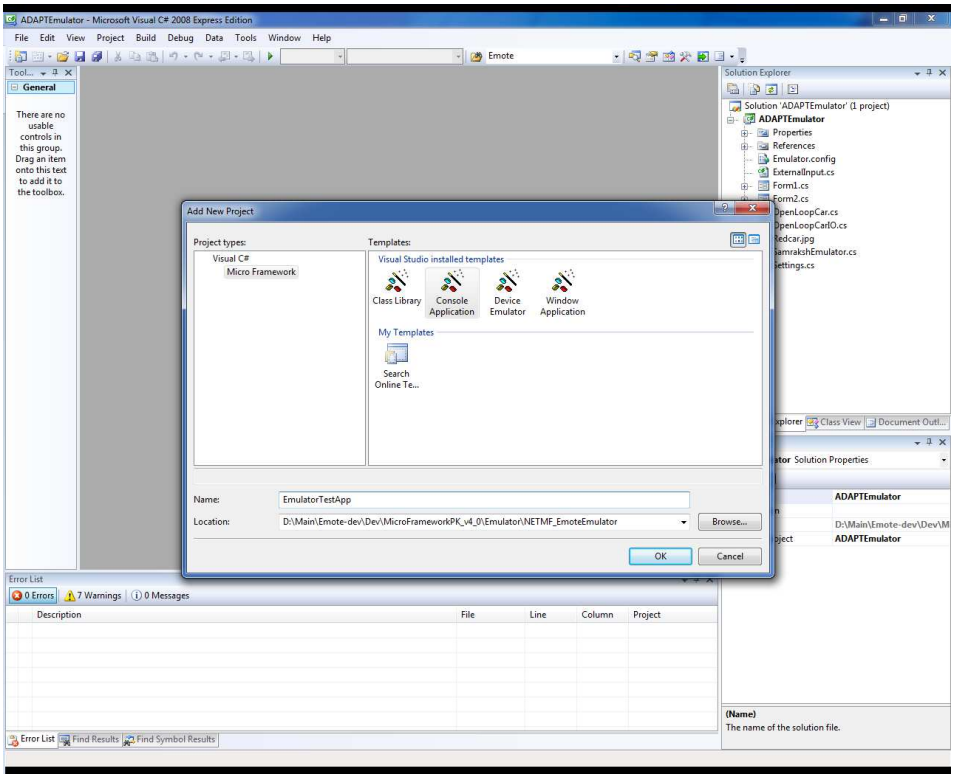


3.2 Add new project to the Emulator solution

- 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)

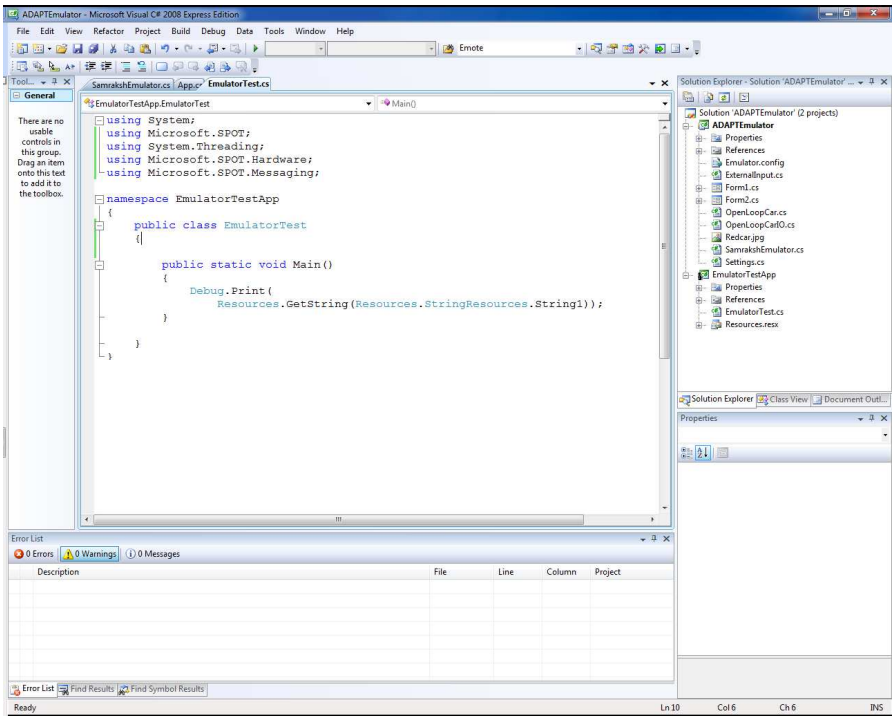


Choose the 'Micro Framework' under Project type and choose 'Console Application', and provide a name and a folder to the project . We will use EmulatorTestApp as the project name in this example.



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.



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 EmulatorTest()
{
    //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
    button1_state = !button1_state;
    _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
    button2_state = !button2_state;
```

```
    _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
    button3_state = !button3_state;
    _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 now 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.

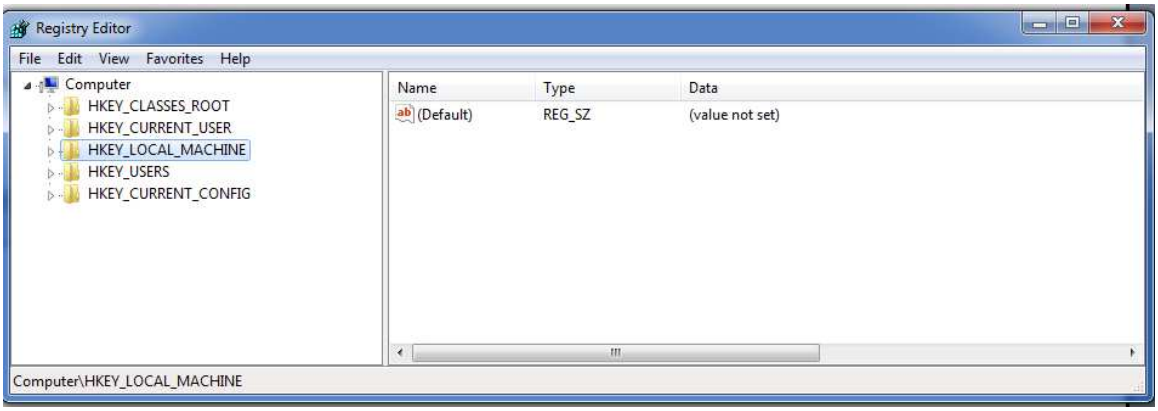
4 Running the Emulator Application

4.1 Registering you Emulator Binary

Note: This step is need only if you are going to launch emulations from Visual Studio, using a prebuilt emulator binary. Else (for example using command line as explain in Section 4.4) you can skip this step.

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 do not 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. The registry can be edited using the *regedit* tool in windows. To do this follow the steps below:

1. Open a command prompt in Administrator Mode (by right clicking on the command) and run the command regedit (Alternately you can directly run it from the runcmd window)



2. You will a window similar to the one shown above. Navigate by clicking on the arrow next to HKEY_CURRENT_USER and successively to key shown below:

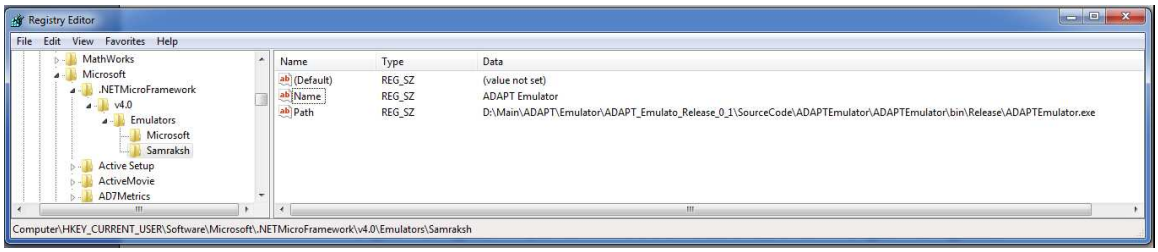
```
HKEY_CURRENT_USER\Software\Microsoft\.NETMicroFramework\v4.0\Emulators
```

Note: In some Windows systems this key is under HKEY_LOCAL_MACHINE instead of HKEY_CURRENT_USER. If you dont kind the Emulators key under HKEY_CURRENT_USER, please check under HKEY_LOCAL_MACHINE

3. You will see a key called Microsoft, which is the default emulator that comes with the .NET Micro framework SDK

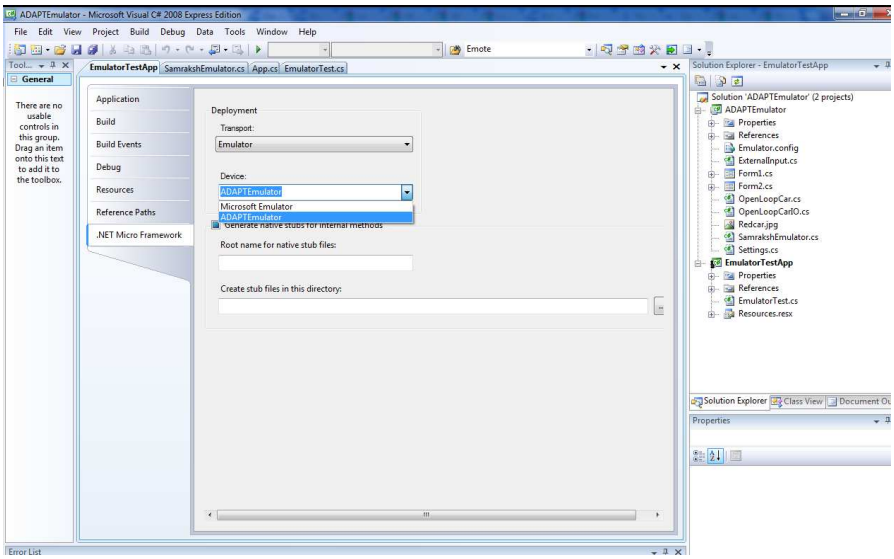
4. Add a second Key called Samraksh: Right click on the Emulator key and select New|Key. Name the key as 'Samraksh'. Next right click on Samraksh and then select new| String Value. Give the String Name as 'Name'. Right click on Name, select Modify and enter the value as ADAPT Emulator. Add another String Value, called Path and provide the path to the binary of the emulator

5. The Emulator registry keys after adding the second key is shown in the snapshot below



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.



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.

5.

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

6. The first parameter for the ADAPTEmulator.exe is the embedded application that you want to run on the emulator. In the above case it is the 'EmulatorTestApp.pe'. The rest of the parameters are the libraries needed by the MF application. For running other applications simply replace the 'EmulatorTestApp.pe' with say 'CarDriver.pe' (the next example we are going to write in Section 6), save it or save it to a new bat file and then run it.

5 Logging and Status Display

The Samraksh Emulator automatically logs all important events inside the emulator to a log file and also displays the same on the status box of the emulator. The logs are preceded by a timestamp with millisecond precision. Also any Debug.Print statement that is used by the user inside the MF application is also captured by the emulator and displayed in the status box and logged to the file.

The messages also display the module which generated the message. Currently there are four modules in the Emulator:

1. CORE: These are debug messages from the emulator core, which is written by Samraksh
2. DASHBOARD: There are messages that are generated as a result of the users interaction with the dashboard/control panel
3. PHYCOMM: There are messages that are generated by the Physical Model and Emulator Communication Library (discussed below in Section 6) written by Samraksh.
4. MFAPP: There are messages generated by the MF application using the Debug.Print statement

Log filename format: The messages are automatically logged to a text file with the name EmulatorLog_* , where the * represents the timestamp when the emulator was started. The format of the file name is EmulatorLog_yyyy_MM_dd_HH_mm.txt , for example a emulation started on June 15, 2012, at 5:10 pm will be logged to a file EmulatorLog_2012_06_15_17_10.txt

6 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.

While writing the MF application, the application should wait for a few seconds (that is sleep for a few seconds) before starting the execution in the main, to allow the physical model to connect to the emulator and initialize the communication. If the Physical Model application actually does not connect to emulator in that time, the emulator will continue to run the MF application, but every time the application tries to communicate with the outside world the emulator will display a warning message in the status window and the log, and the emulator will simply drop those messages destined for the physical model.

6.1 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

ADC: 1
USART: 2
SPI: 3
I2C: 4
USB: 5
```

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.)

6.2 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);
```

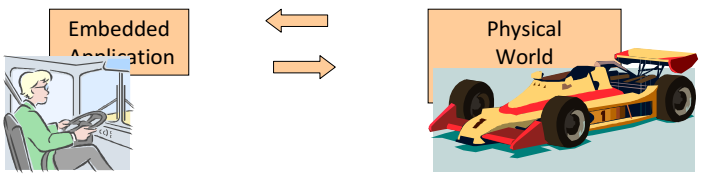
Public Methods:

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 SendToADC(byte[] buffer)`	Send the message in the byte array buffer, to the emulator, to be sent to the MF application through the ADC port. **Note:** There is no corresponding InitializeADC method, since ADC in a uni-directional interface, from Model to Emulator/MF Application
`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

6.3 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.



6.3.1 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 acceleration, 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.

6.3.2 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 'State' 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 States 'START' and 'STOP'

```
enum CarCommands
{
    START, STOP
};

class CarModelInput
{
    public short SteeringAngle; //-128to127
    public sbyte Acceleration;//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.Acceleration;
    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.State = (byte)CarCommands.STOP;
    SendModelInput()
}
void Stop()
{
    carInput.State = (byte)CarCommands.STOP;
    SendModelInput();
}
void Turn(short degrees)
{
    carInput.SteeringAngle = degrees;
    carInput.State = 0;
    carInput.Acceleration = 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 5 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.

Note: If the Physical Model application actually does not connect to emulator, the emulator will continue to run the MF application, but every time the application tries to communicate with the outside world the emulator will display a warning message in the status window and the log and will simply drop those messages.

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(5000); //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.

7 Analog to Ditigal Convertor

The Samraksh Emulator supports the ADC module. The ADC module provides up to 10, 16-bit ADC channels. Currently the module supports only sampling frequency upto 1khz, mainly due to the resolution of the time parameter in the .NET Timer APIs. To use the ADC interface, the users

need to use the Samraksh.SPOT.Emulator interface provided by Samraksh inside the MF App (A Samraksh.SPOT.Emulator.dll is provided in the bin directory of the Emulator)

After the ADC is initialized with the number of channels and the sampling time, the ADC can be operated in the following three modes:

1. Single Value Mode: A 'GetData' method samples the current value of the ADC channels and returns the values in an array.

2. Batch Mode: A 'ConfigureBatchMode' method, lets the user use the ADC in a batch mode. The user can provide the number of samples required (to be sampled at the sampling time provided as a parameter of the Init method) and a callback function, to be called once the samples ready. But the ADC stops sampling after finishing that batch. For every ConfigureBatchMode call, the callback will be called only once. It is assumed that the buffer provided is at the least of size (Number Of Samples * Number Of Channels).

3. Continuous Mode: A 'ConfigureContinuousMode' method lets the users use ADC continuously. The method's parameters and the semantics are very similar to the batch mode, except once configured the ADC keeps sampling the Channels and keeps calling the callback when the buffer is full. Again the size of the buffer is assumed to be at least as big as (Number Of Samples * Number Of Channels). To stop the ADC from sampling the channel (and signaling the callback function) the 'Stop' method, can be used

7.1 Samraksh's ADC API

Namespace: Samraksh.SPOT.Emulator

```
public enum AdcSampleTime
{
        ADC_SampleTime_1_5_Cycles,
   ADC_SampleTime_7_5_Cycles,
   ADC_SampleTime_13_5_Cycles,
   ADC_SampleTime_28_5_Cycles,
   ADC_SampleTime_41_5_Cycles,
   ADC_SampleTime_55_5_Cycles,
   ADC_SampleTime_71_5_Cycles,
   ADC_SampleTime_239_5_Cycles,
}
```

Class: ADC

Members:

Public Constructors:

Name	Description

ADC()	Creates a new ADC object

Delegates:

```
delegate void AdcCallback (bool Status);
```

Public Methods:

Name	Description
`void Init (AdcSampleTime SampleTime, int NumberOfChannels)`	Initializes the ADC channels. The SampleTime parameter specifies the number of cycles for which the ADC channel needs to sampled, and the NumberOfChannels indicated the number of ADC channel to be used.
`int GetData(ushort[] CurrSample, uint StartChannel, uint NumChannels)`	Sample ADC now and return the data in the CurrSample array. The NumChannels specifies the number of channels to be sampled, starting from the StartChannel.
`int ConfigureBatchMode(ushort[] SampleBuff, uint StartChannel, uint NumChannels, uint NumSamples, uint SamplingTime, AdcCallback Callback)`	Configures the Batch Mode of operation and data is returned in the SampleBuff array when ready. The 'Callback' method is called with a 'Status' set to true if the batch mode was successful and when the SampleBuff has the number of samples specified by NumSamples. NumChannels and StartChannel means the same as above. The SamplingTime specifies the time interval between 2 ADC samples in microseconds (or conversely the ADC sampling frequency). **Note:** The current minimum sampling time supported by the Emulator is 1000 microseconds.
`int ConfigureContinuousMode(ushort[] SampleBuff, uint StartChannel, uint NumChannels, uint NumSamples, uint SamplingTime, AdcCallback Callback)`	Configures the Continuous Mode of operation and returns data in the SampleBuff array. The 'Callback' method is called with a 'Status' set to true if the ADC sampling was successful and when the SampleBuff has the number of samples specified by NumSamples. NumChannels and StartChannel means the same as above. The SamplingTime specifies the time interval between 2 ADC samples in microseconds (or conversely the ADC sampling frequency). **Note:** The current minimum sampling time supported by the Emulator is 1000

	microseconds.
`void Stop()`	Stop the ADC when operating in the Continuous Mode. No more AdcCallbacks will be received after the Stop call.

7.2 A Example Micro Framework ADC Application

In this example, we will write an ADC test application, that samples a 2-channel ADC, Channels 0 and 1, at a sampling rate of 1Khz. The Samraksh Emulator supports of 10 ADC channels of 16-bits each. we will write an application that uses the ADC in a Continuous Mode, where it configures the ADC to read the channels at an interval of 1ms. The application will provide the ADC interface with a buffer of size 2000, to reach 1000 samples (for 2 channels) at one time, then call the Callback when the samples are available. Thus the ADC interface will call the applications Callback once every second with 1000 samples. After 10 such callbacks or 10000 samples we will stop the ADC interface from sampling the channels.

Note 1: The code for this example can be found under the SourceCode\MFApplications\ADCTestApp directory in our Emulator distribution

NOTE 2: Currently the maximum ADC sampling time supported by the emulator is 1 milli second

1. Create new project: The steps for creating a new project for Micro Framework ADC application is similar to the what has been explained in Section 3 for creating a basic application. We will assume that the users have created an project called the ADCTestApp and has renamed the Program.cs file to ADCTest.cs.

2. Add the Samraskh library to the project: To use the ADC interface you need to import the 'Samraksh.SPOT.Emulator' library, by using the 'using' keyword and also add the reference to the library. Right click on the 'References' in the Solution Explorer, click on the 'Browse' tab and browse to the 'bin' directory of Samraksh's Emulator and choose the Samraksh.SPOT.Emulator.dll

3. Create an instance of the ADC: Create a global instance of ADC in the ADCTest class. Next add a constructor for the ADCTest class.

```
AdcSampleTime SampleTime = AdcSampleTime.ADC_SampleTime_1_5_Cycles;
ADC Adc;
const int NumberOfSamples = 1000;
uint SamplingTime = 1000; //micro seconds. 1KHz frequency.
ushort[] Adc_buffer= new ushort[NumberOfSamples*2];
int BatchNumber=0;

ADCTest()
{
    Adc = new ADC();
}
```

4. Create a AdcCallback function: Next let us create a function for the ADC Interface to Callback when samples are ready. Let us call this function 'ContinousModeCallback'. This is the function where the processing of the samples should take place. In this example, we will simply calculate the mean of 1000 samples, for each of the 2 channels and will simply print them to the debug channel. After 10 Callbacks which we keep track in a BatchNumber variable, we will call the ADC.Stop method to stop the ADC from sampling. The code for the callback function is shown below

```
public void ContinuousModeCallback(bool Status)
        {
            if (Status) {
                double[] Mean=new double[Adc_config.NumberOfChannels];
                for(int i=0; i< Adc_buffer.Length; i++){
                    Mean[i%Adc_config.NumberOfChannels]+=Adc_buffer[i];
                }
                for (int i=0; i<Adc_config.NumberOfChannels; i++){
                    Mean[i]/= (adc_buffer.Length/Adc_config.NumberOfChannels);
                    Debug.Print("BatchNumber: "+ BatchNumber+ ", ADC[" + i + "]
Mean:" + Mean[i]);
                    Mean[i] = 0;
                }
                BatchNumber++;
                if (BatchNumber == 10) Adc.Stop();
            }
            else { Debug.Print("Error in ADC Continous Mode"); }
        }
```

5. Initialize and Configure ADC: Next we will initialize the ADC using the ADC.Init method and will configure the ADC to run in the Continuous Mode using the ConfigureContinuousMode method. We will write a StartCotinuousModeTest function to configure the ADC in continuous mode , so that it is simpler to invoke from the Main. We will first call the Init method with 'AdcSampleTime' and 'NoOfChannels'. The AdcSampeTime Enum specifies the number of CPU cycles the channel should sampled. This parameter has no effect inside the Emulator. The 'NumberOfChannels' parameter specifies that number of ADC channels we are going to use our application. Also create an ushort buffer of size (NumberOfChannels* NumberOfSamples), in our case of size 2000.

Next we will sleep for couple seconds to let the ADC module initialize. Finally we will configure the ADC in Continuous mode. The ConfigureContinuousMode method takes 6 parameters; 1. The buffer in which the sample will be filled up; 2. the number of the starting number of the channel; 3. The number of channels to be samples; 4. The number of samples to be sampled and copied to the buffer before the callback is activated; 5. The SamplingTime between consequtive samples (Or alternately 1/frequency); 6. The Callback function. The SamplingTime in our case will be 1000 microseconds (i.e. 1ms or 1KHz sampling frequency).

The code for the Callback function is shown below.

```
void StartContinuousModeTest()
{
    Debug.Print("Starting Continuous Mode Test");
    try
    {
        Adc.Init(SampleTime, NoOfChannels);
        Thread.Sleep(2000); //Let the ADC system initialize
        Adc.ConfigureContinuousMode(Adc_buffer, 0, 2, NumberOfSamples,
SamplingTime, ContinuousModeCallback);
    }
    catch (Exception e)
    {
         Debug.Print(e.ToString());
    }
}
```

6. Write the Main: Finally lets write the main of the test app. First lets sleep for couple seconds for the rest of the Emulator to boot up (for example the physical model app). Next lets create an instance of the ADCTest application. Next we will call the StartContinuousModeTest method to start the test application and then we will go into an indefinite sleep. The code for the main is shown below.

```
public static void Main()
{
    Thread.Sleep(2000);
    ADCTest _test = new ADCTest();
    _test.StartContinuousModeTest();//Start a Continuous mode test
    Thread.Sleep(Timeout.Infinite);
}
```

7. Compile and Run: Compile and Run the application as shown in Section 4 (along with the Microphone Physical Model App described in Section 7.3 Next).

7.3 A Example Microphone Physical World Application to provide Samples

In this section we will write a Physical World Application that can generate the samples to be read by the ADCTest application running in the Micro Framework (that we wrote in Section 7.2). The Microphone physical world application will simulate the values generated while sampling a 2-channel ADC at 1Khz. For simplicity, we will simulate 12-bit random noise on both the channels. That is for each sample we will generate two uniform random numbers, one for each channel, between 0 and 4095. And the samples generated will be feed to the MF application using the PhysicalModelEmulatorComm library described in Section 6.1/6.2 .

Note 1: The code for this example can be found under the SourceCode\PhysicalModels\MicrophonePhysicalApp directory in our Emulator distribution

1. Create a new windows application: Create a new C# windows console application and name it MicrophonePhysicalApp. Rename the Program.cs to MicrophoneSamples.cs. Import the Samraksh.PhysicalModel library (using the 'using' keyword) and add a reference to it by right

clicking on the 'References' on the SolutionExplorer, click on the 'Browse' tab, browse to the bin directory of our Emulator distribution and selecting the 'PhysicalModelEmulatorComm.dll' . Also import (and add reference to) System.IO which you will need to use a logger to write to a file.

2. Create a constructor and Connect to the Emulator: Create a constructor which takes the sample time and the number of channels as parameter. In the constructor create an instance of the PhysicalModelEmulatorComm object and connect to the Emulator. Also create global variables (as shown below) that we will need.

```
PhysicalModelEmulatorComm EmulatorComm; //Emulator Communication
int SampleTime; //in microseconds
Thread SampleThread; //thread to create samples
static int NumberOfChannels = 2; //Default number of channels
ushort[] CurrentValues = new ushort[NumberOfChannels]; //Current value of the
channels
ushort[] Adc_buffer = new ushort[1000 * NumberOfChannels];
double[] Mean = new double[NumberOfChannels];
static Random RandomNumberGen = new Random(NumberOfChannels); //Random number
generator
int BatchNumber;  //Current Batch number
int CurrSampleNumber = 0;
uint NumberOfSamples = 1000*12; //Number of samples to generate
StreamWriter logFile; //Log file to write the samples

MicrophoneSamples(int _SampleTime, int _numberOfChannels)
{
    EmulatorComm = new PhysicalModelEmulatorComm();
    EmulatorComm.ConnectToEmulator();
    SampleTime = _SampleTime;
    NumberOfChannels = _numberOfChannels;

    //Create a file with current datetime to use as log
    DateTime now = DateTime.Now;
    String nowString = now.ToString("yyyy_MM_dd_H_mm");
    logFile = new System.IO.StreamWriter("MicroPhoneApp_" + nowString +
".txt");
}
```

3. **Write the Start and Stop functions:** Next we will write the Start and Stop functions. In the Start function we will instantiate the SampleThread, which will use the GenerateRandomSamples (which we will write next) function as its ThreadStart and then we will start this thread. In the Stop function we will simply flush and close the log file. The code for the functions are shown below.

```
void Stop()
{
    logFile.Flush();
    logFile.Close();
}
void Start()
{
    SampleThread = new Thread(new ThreadStart(GenerateRandomSamples));
    SampleThread.Start();
}
```

4. Write a function to generate random samples: Next we write the GenerateRandomSamples function that will generate 12-bit uniform random numbers for each of the channels. We will generate a certain number of samples, specified by the 'NumberOfSamples' variable (a sample in this case is 2 12-bit number), while sleeping for 'SampleTime' amount of time between each samples. Please note that SampleTime is specified in microseconds, as in the MF application in Section 7.2. Once the values are generated we will use the 'SendToADC' method of the PhysicalModelEmulatorComm library to send the values to the Emulator and through the Emulator to the MF application. We will also simultaneously compute the mean for each channel for each batch of 1000 values, the same continous batch size that the MF application in Section 7.2 requests from the ADC interface. The code is shown below.

```
void GenerateRandomSamples()
{
    BatchNumber = 0;
    for(int j=0; j<NumberOfSamples; j++)
    {
        for (int i = 0; i < NumberOfChannels; i++)
        {
            CurrentValues[i] = (ushort)RandomNumberGen.Next(4095); //12-bit
number
            Adc_buffer[j * NumberOfChannels + i] = CurrentValues[i];
            Mean[i] += CurrentValues[i];
        }
        CurrSampleNumber++;
        EmulatorComm.SendToADC(CurrentValues, NumberOfChannels);
        logFile.WriteLine(j + ", " + CurrentValues[0] + ", " +
CurrentValues[1]);
        //A Batch of 1000 samples done, compute mean for each channel and
display
        if (CurrSampleNumber== 1000)
        {
            CurrSampleNumber = 0;
            for (int i = 0; i < NumberOfChannels; i++)
            {
                double[] Mean = new double[NumberOfChannels];
                Mean[i] /= (Adc_buffer.Length / NumberOfChannels);
                Console.WriteLine("BatchNumber: " + BatchNumber + ", ADC[" + i
+ "] Mean:" + Mean[i]);
                logFile.WriteLine("BatchNumber: " + BatchNumber + ", ADC[" + i
+ "] Mean:" + Mean[i]);
                Mean[i] = 0;
            }
            BatchNumber++;
        }
        Thread.Sleep(SampleTime / 1000);
    }
    Stop();//Done generating samples, Stop.
}
```

5. Write the Main: Finally we will write the Main. Here we will create an instance of the MicropheSample Class for 1000 microsecond (i.e. 1 millisecond or 1Khz frequency) SampleTime and 2 channels. We will wait for 2 seconds for the system to stabilize and then we will start the Sample generating thread, by calling the Start method and finally the main will be put so sleep indefinitely. The code is shown below.

```
public static void Main()
{
    MicrophoneSamples Samples = new MicrophoneSamples(1000,2); //1 millisecond
interval and 2 channels
    Thread.Sleep(2000);
    Console.WriteLine("Starting Sample Generation...");
    Samples.Start();
    Thread.Sleep(Timeout.Infinite); //Go to sleep
}
```

6. Compile and Run: When you compile and run the MF application in Section 7.2 and the Microphone sample application in Section 7.3 you see that the Mean for each batch of 1000 values displayed on the consolde in both the applications are the same.

8 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.

9 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 currently run a network emulation by running multiple instances of the emulator and networking them on their own.

10 Some of the quirks in the Emulator

1. SerialPort.DataReceived Event Handler does not work inside the Emulator. This is probably something to do with the virtual nature of the serial port inside the Emulator. Use polling instead.

2. The max period currently supported in timers and ADC is 1ms.

11 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

12 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