Using the eMote .NOW
Version 1.0

The Samraksh Company

Manual Version 0.9.9.5
February 3, 2013

http://www.samraksh.com/

ALERT: Appropriate for some young makers.

The eMote software platform is designed to be easier than traditional micro-controller (µC) programing environments, especially for users with less software experiences.

At present, however, the .NOW is provided with minimal purpose built accessories. Because the .NOW has lots of general purpose IO, users with experience at wiring up hardware should be able to use the .NOW in a wide range of projects. However, this may be inappropriate for makers with less experience at electrically connecting µCs to a range of general hardware.

Contents

1. Overview of the eMote .NOW Platform

1.1 Origins of the .NOW eMote Platform

The wireless sensing network community was among Samraksh's earliest customers. This group had specific needs, such as reliable wireless networks that would work when sensors were deployed with a variety of topological and power management configurations.

Samraksh developed robust built-in wireless and low power management solutions. Samraksh also noted that not every wireless sensing customer wanted to use the same set of sensors. To Samraksh, this meant an opportunity to develop a more general solution in which sensors could be integrated with the basic platform.

It's been a number of years now and the demand for wireless networking is only growing. Meanwhile, the maker / hobbyist community has also been growing. This community has a number of options for microcontrollers that support, well, various projects that we want to do. Recent entries include the Arduino and the Netduino.

The eMote .NOW is geared to supporting the maker/hobbyist community. While it delivers wireless networking out of the box, it also offers other features.

Terminology. ".NOW" refers to the physical board. "eMote" refers to the pre-installed software that lets you write programs in C#.

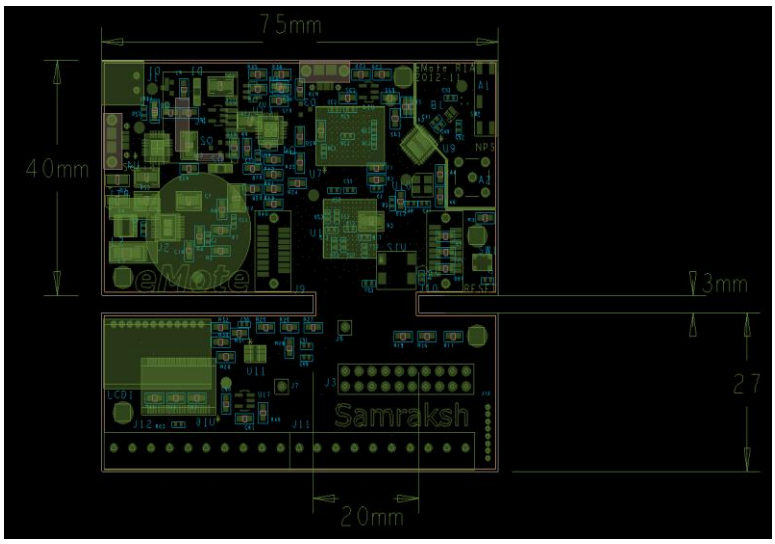


Figure 1. The eMote .NOW Layout.

1.2 Layout of the .NOW Platform

Figure 1 and **Figure 2** presents the layout of the .NOW platform, with an accompanying marked up version indicating primary features. The details of the pinouts are given in **Section 1.4**.

Note the *radio/wireless* area, the two high density connectors and the MCU on the main board area. The attached board area has the LCD, the JTAG and the low density connectors.

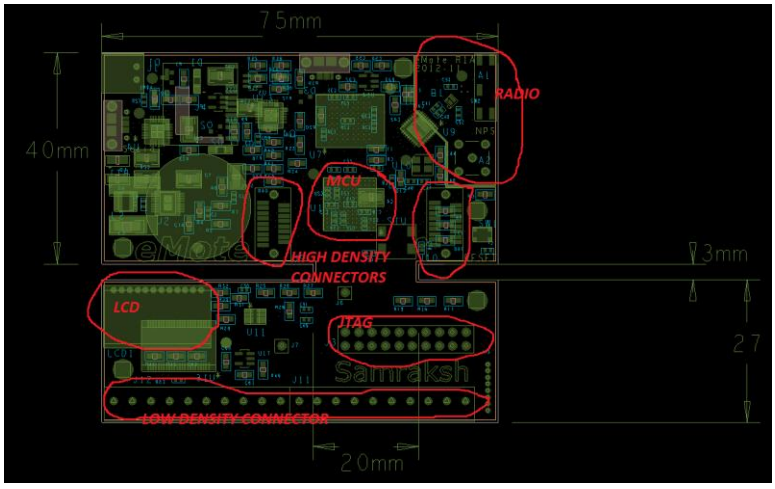


Figure 2. The .NOW layout with annotation.

1.3 Wireless and the eMote .NOW

Wireless Networking in the eMote .NOW

The .NOW offers built-in wireless capability. This wireless capability uses the IEEE 802.15.4 standard. eMote .NOW is not the only product making use of IEEE 802.15.4. In order to highlight some of the differences in capability offered by the eMote, it is useful to review the 802.15.4 standards. (IEEE is the Institute of Electrical and Electronics Engineers).

Background: Network and Network Stack

As part of this review, we consider the 'network stack' concept. The network stack is an abstraction that maps the functionality that needed to make networking a reality to different layers. Each layer offers a specific and related set of functionality and the layers are stacked on top of one another to provide the complete functionality for networking. Usually in any networking stack the lower layers provide simpler functions, which are in turn used by the higher layers to provide increasingly complex functionality and end-to-end semantics.

It is helpful to start with the conceptualization of a network stack for wired networking. The OSI (Open Systems Interconnection) standard model from the International Standards Organization (ISO 7498-1) is shown in the **Figure 3**. In implementation and practice, the layers 5 through 7 are folded together. The functionality at each layer is governed by the standards. The standards-compliant implementation at each layer, which supports the required functionality, can be achieved by different variants which are typically termed protocols. For example, at the transport layer, TCP/IP protocols suite includes TCP and UDP.

In the case of wireless networking, IEEE 802.15.4 is one of the standards designed for low-power short-range networking. The focus of this standard is that of a wireless personal area network (WPAN). Using the OSI as a reference model, the IEEE 802.15.4 standard addresses the functionality of the two bottom layers, the physical layer and the MAC layer.

Key functionalities of the physical layer include managing the radio frequency of the sender/receiver (usually most radio standards operate in a number of frequencies, sometimes also referred to as radio channels), synchronizing the sending and receiving radios, and encoding/decoding the data transmitted into physical layer symbols that are more suitable for transmission/reception. Key functionalities of the

OSI Model

	Data unit	Layer	Function
Host layers	Data	7. Application	Network process to application
		6. Presentation	Data representation, encryption and decryption, convert machine dependent data to machine independent data
		5. Session	Interhost communication, managing sessions between applications
	Segments	4. Transport	End-to-end connections, reliability and flow control
Media layers	Packet/Datagram	3. Network	Path determination and logical addressing
	Frame	2. Data link	Physical addressing
	Bit	1. Physical	Media, signal and binary transmission

Figure 3. OSI Model (Adapted from *Wikipedia*. See http://en.wikipedia.org/wiki/OSI_model.

MAC layer include managing access to the physical channel and enabling MAC frame transmission. The MAC frame is the term given to the unit of data that will be transported. A MAC can have several types of frames, such as the data frame, the MAC command frame, an acknowledgement frame and a beacon frame. The beacon frame is used by an implementation that includes beaconing, which is a process that allows a node to announce its presence. A brief discussion of the details of the physical layer and of the MAC layer can be found at Wikipedia:

http://en.wikipedia.org/wiki/IEEE_802.15.4.

A crucial point is that specifications for the upper layers of the network stack for wireless communication are not addressed by the 802.15.4 specs. This means that functionality that would be handled by the upper layers is open to different implementations and customized solutions.

MAC Implementation Approaches and Higher Level 'Solutions'

Product offerings may assume the needs of wireless applications and may provide their customized solution to implementations of the MAC as well as elements of upper layers. For example, in a wireless sensing application, the application infrastructure might include a way to organize the wireless nodes to allow them to pass their data from nodes to a collection point.

Zigbee: One of the integrated product offerings using the IEEE 802.15.4 standard for the physical and MAC layer is that of Zigbee solution. Zigbee is a networking standard for higher layers published by the Zigbee Alliance (a group of companies), that build on top of the 802.15.4 standard. This solution proposes distinct types of nodes: the full functioning nodes and those that are reduced function. The full functioning nodes can act as Coordinator nodes. The Coordinator concept plays a role in how the Zigbee solution sets up and maintains the network. The reduced function devices are only allowed to talk with one full function node.

This sets up a mesh network in which Coordinators talk to each other and each Coordinator or fully functioning device can have multiple reduced function devices which talk to it. Thus, passing a sensor reading from a reduced function device would involve a message to its associated fully functioning device. This device, if not a Coordinator, would be able to interact with a Coordinator and pass the message along the network as per the application-specific programming. For more information about Zigbee please see http://en.wikipedia.org/wiki/ZigBee

eMote Stack

Every node in the eMote network is a peer. The main difference between the eMote stack and the Zigbee stack is that the eMote stack considers the typical networking case to be peer-to-peer, whereas the Zigbee approaches uses Coordinator/full function/reduced function nodes. The eMote stack approach lets the user build whatever type of network structure they want on top of the peer-to-peer architecture. As one example, the peer-to-peer nodes could build routes through the network to a base station. As another example, the base station could host a GUI that allowed the user to process content from received messages and display resulting information.

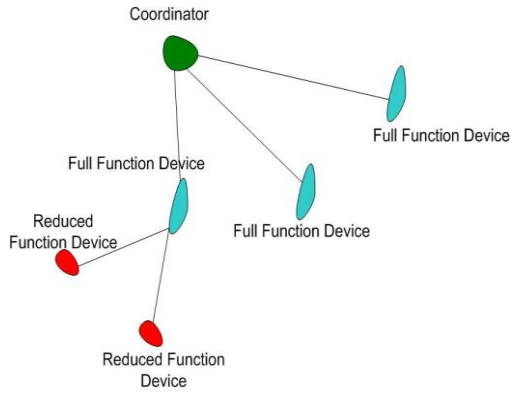


Figure 5. The *coordinator* network approach used by ZigBee.

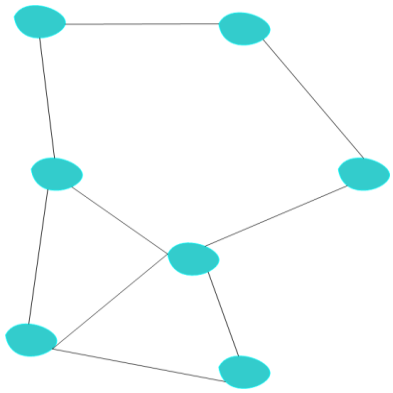


Figure 4. The Peer-to-Peer network used by the eMote.

Figure 4 illustrates the Zigbee Coordinator-based network architecture; while **Figure 5** illustrates eMote's peer-to-peer based architecture.

There exists more than one type of MAC protocol that can be used. eMote uses a 'simple MAC', based on CSMA (carrier sense, multiple access). A future write-up will discuss CSMA and how eMote would utilize it. For those wanting to take a look at the general description of CSMA, you may start with the description at http://en.wikipedia.org/wiki/Carrier_sense_multiple_access.

Table 1. The .NOW related specifications.

Quantity	Value
Operating Frequency	2.4 GHz
Data Rate	250 kb/s
Antenna	SMA connector or built in chip antenna
Encryption	Not supported in software yet; AES module in hardware
Outdoor Range	100 m (Line of Sight, without any hindrance like foliage)
Indoor Range	Roughly 3-10 m (Through one wall)
Radio Chip	Atmel RF231
MAC Protocols	CSMA [for now, may be expanded].

Access to the API for the MAC layer is provided by one of the Samraksh custom DLLs. There are additional MAC protocols being developed for the general eMote; selected candidates may be offered for the eMote .NOW platform in the future.

We close this note with **Table 1**, which shows some of the .NOW network-related specifications.

Radio and Networking Application Programming Interface (API)

Table 2. The Namespace for C# programs using the radio on the eMote.

Name Space	Details
Samraksh.SPOT.Net	All elements common to all layers of the networking stack, like error enums, basic link structures, etc.
Samraksh.SPOT.Net.Radio	Radio interface and modules. Each standard should have its own class implementing it. For example IEEE 802.15.4, is implemented as Radio_802_15_4
Samraksh.SPOT.Net.Mac	MAC interface and modules. Each MAC should be implemented as separate class.
Samraksh.SPOT.Net.Routing	Routing interface and modules.

This section presents the C# API for radio utilization. The namespace is shown in **Table 2**. The entire networking stack is published as single dynamic link library (DLL) called "`Samraksh_SPOT_Net.dll`".

Table 3 provides the API for the Radio and **Table 4** provides the API for the MAC. There is also an API for a callback when a message is received in the "`Samraksh.SPOT.Net`" namespace.

Before we present the networking API, we note that the following structs and classes are defined, and belong to the three namespaces that are indicated.

Namespace `Samraksh.SPOT.Net`

```
enum DeviceStatus
{
    Success,
    Fail,
    Ready,
    Busy,
};

enum NetOpStatus
{
    E_RadioInit,
    E_RadioSync,
    E_RadioConfig,
    E_MacInit,
    E_MacConfig,
    E_MacSendError,
    E_MacBufferFull,
    S_Success

};
```

```
public delegate void ReceiveCallBack(byte[] message, UInt16 size, UInt16 source, bool
unicast, byte rssi, byte linkQuality)
```

Namespace Samraksh.SPOT.Net.Radio

```
struct RadioConfiguration
{
    byte TxPower;
    byte Sensitivity;
    byte Channel;
    byte TimeStampOffset;
};
```

Class: Radio_802_15_4 implements interface IRadio (see **Table 3**)

Namespace: Samraksh.SPOT.Net.Mac

```
enum Addresses
{
    BROADCAST = 65635,
};

struct MacConfiguration
{
    bool CCA;
    byte NumberOfRetries;
    byte CCASenseTime;
    byte BufferSize;
    byte RadioID;
};



struct Link
{
    byte Quality;
    byte LossRate;
    byte AveDelay;
};
```

```
enum NeighborStatus
{
    Alive,
    Dead,
    Suspect
};

struct Neighbor
{
    UInt16 MacAddress;
    Link ForwardLink;
    Link ReverseLink;
    NeighborStatus Status;
    UInt64 LastHeardTime;
    byte ReceiveDutyCycle; //percent
    UInt16 FrameLength;
};

struct NeighborTable
{
    byte NumberValidNeighbor;
    Neighbor[] Neighbor;
};
```

Class: CSMA implements interface IMac (see **Table 4**)

Table 3. The C# APIs Interface IRadio for use on eMote platforms.

Function Call	Description
DeviceStatus Initialize (RadioConfiguration config, ReceiveCallBack callback)	Initializes the Radio with the parameters specified in config and the callback delegate will be called whenever a new packet is received by the Radio
DeviceStatus Configure (RadioConfiguration config)	Change the parameters of the Radio after Initialization
DeviceStatus UnInitialize()	UnInitialize the Radio. (Object and memory is not released)
byte GetID()	Return the ID of the radio
DeviceStatus TurnOn()	Turn the radio On (comes up in receive state by default)
DeviceStatus Sleep(byte level)	Put the radio to sleep. The level can specify the state off sleep and power consumption
bool SetAddress(UInt16 Address)	Set the identifier of the radio to preferred address. . Please note that changing the Radio identifier will also change the MAC layer identifier
UInt16 GetAddress()	Returns the Address of the Radio
NetOpStatus PreLoad (byte[] message, UInt16 size)	Load the message into the transmit buffer of the radio
NetOpStatus SendStrobe()	Send the packet already in the transmit buffer
NetOpStatus Send (byte[] message, UInt16 size)	Load and send the message
NetOpStatus SendTimeStamped (byte[] message, UInt16 size, UInt32 eventTime)	Load and Send the message, with radio layer time stamping. The offset for the timestamp in the packet is specified by TimeStampOffset member of the RadioConfiguration structure passed as parameter during radio module initialization.
bool ClearChannelAssesment()	Asses the channel activity (the default time is 140 us). Returns true if channel is free.

Note that the default address for the radio is based on the CPU identifier.

Table 4. The C# MAC APIs for use on the eMote platforms.

Function Call	Description
DeviceStatus Initialize (MacConfiguration config, ReceiveCallBack callback);	Initializes the MAC with the parameters specified in config and the callback delegate will be called whenever a new packet is received by the MAC
DeviceStatus Configure (MacConfiguration config);	Change the parameters of the MAC after Initialization
DeviceStatus UnInitialize()	UnInitialize the MAC. (Object and memory is not released)
bool SetAddress(UInt16 Address)	Set the MAC identfier to preferred address. Please note that changing the MAC identifier will also change the radio layer identifier.
byte GetID()	Returns the ID of the MAC
UInt16 GetAddress()	Returns the MAC Address of the MAC
NetOpStatus Send (UInt16 Address, byte[] message, UInt16 offset, UInt16 size)	Send a message to another node
bool GetNeighborStatus (UInt16 macAddress, ref Neighbor neigbor)	Get the details of the neighbor referred by the macAddress and the details are copied into the neighbor structure passed as parameter by reference. Returns true if operation is successful, else returns false.
byte GetBufferSize()	Return the size of the MAC buffer (send and receive buffers have the same size)
byte GetPendingPacketCount()	Return the number of packets pending in the MAC's send buffer
DeviceStatus RemovePacket (byte[] msg)	Remove a packet from the MAC's send buffer, referred to by the msg.

Note that both the radio and the MAC identifier will be the same. If you have a preferred address, set either or both (as the same value). The default MAC identifier is the same as the default Radio identifier.

Radio and Mac Initialization Example

One of the example programs involves the use of the radio; programs on two different eMote .NOWs send messages to each other. The full code for this example is shown in the section entitled **Application 1: Using the Radio with a Ping Program**. If you look at this code, there are specific uses of the radio and mac APIs, such as shown in **Table 7**.

1.4 Real Time Extensions, Precision Timing and the eMote .NOW

This section presents the approach taken by The Samraksh Company in developing a Real Time Extension to the .Net Micro Framework's CLR. The goal is to offer improved real-time support.

Background: Standard MF Scheduler Design and Ramifications

Before discussing the Real Time extensions, it is helpful to briefly review the standard .NET MF scheduler. This is a weighted priority non-preemptable scheduler. It does support multi-threaded applications similar to other .NET version. Each thread may be assigned a priority. Interrupt and Timer events run inside their own threads and have the highest priority. Once a thread starts running it cannot be preempted by other application threads and they usually run till the current activity is complete or their allocated *time quantum (20ms* by default*)* is over. This non-preemtable nature of the scheduler poses timing issues for asynchronous interrupt driven applications with strict timing requirements.

Samraksh Real Time Extension

In adding a real-time capability to the NETMF we have modified the CLR and Scheduler to make it a preemptable scheduler. This is needed in developing the real-time capability.

For the C# user, the RealTime extension provides a timer interface called RealTimeTimer; it is similar to the standard C# timer. But the execution of this timer handler will happen in interrupt mode; that is, the managed handler of the RealTimeTimer can interrupt all other threads including the regular Interrupt and Timer threads. The result is a timer with very accurate timing properties. The other difference between the RealTimeTimer and the standard timer is that the minimum timer period supported by the standard timer is 1ms, while we support a timer period of a few 100 micro-seconds.

The Samraksh Real Time Timer and Garbage Collection

In the standard NETMF if the Garbage Collector (GC) starts running, all threads belonging to the managed application are suspended until the GC finishes cleaning the heap, possibly compacting the heap (which can be a very time consuming operation). The GC can interrupt the application threads, including timer and interrupt threads.

The RealTimeTimer provides a way to the user to avoid this problem, *provided the user writes the C# handler of the RealTimeTimer is a certain way.* If each and every single variable and object used in the RealTimeTimer handler is allocated either statically or globally, then the code in the RealTimeTimer handler will continue to execute even when Garbage Collector is active. This feature makes the RealTimeTimer very accurate, in most cases with an average Jitter of a few micorseconds. See the performance graphs of the next sub-section, in which the .NET MF standard timer performance is compared to that of the Samraksh RealTimeTimer.

Performance Comparison of the Samraksh Real Time Timer and .NET MF Standard Timer

The RealTimeTimer in general has vastly improved jitter performance to less than 3 micro seconds. While running the CPU at 48Mhz on an eMote we guarantee predictable performance for timer values above 500 microseconds. For less than 500 microseconds the performance is not predictable.

However for values between 500-750 microseconds the average delay is slightly higher than the timer value set; see Figure 6 and Figure 7. For 500 microseconds it's about 54 microseconds above the set value and this additional delay goes to zero for values about 750 microseconds.

However, the real performance of the RealTimeTimer is its ultra-low jitter values compared to the standard timer. A standard timer running at 5000 microseconds has an average jitter of 450 microseconds, while the RealTimeTimer, running at 500 microsecond timer (a value 10 times smaller) has a mean jitter of about 2 microseconds, as shown in Figure 8 and Figure 9. Maximum jitter for the standard timer is in the range of 2.5 to 3 milliseconds, which is the average time taken by the Garbage Collector to do a Mark & Sweep operation. In contrast, the maximum jitter for the RealTimeTimer is 12 microseconds, a vast improvement. Jitter histograms are shown In Figure 10 and Figure 11.

All results in this section were collected on a eMote.Now running at 48MHz (which is the default frequency), running a (modified) .NET 4.3 CLR.

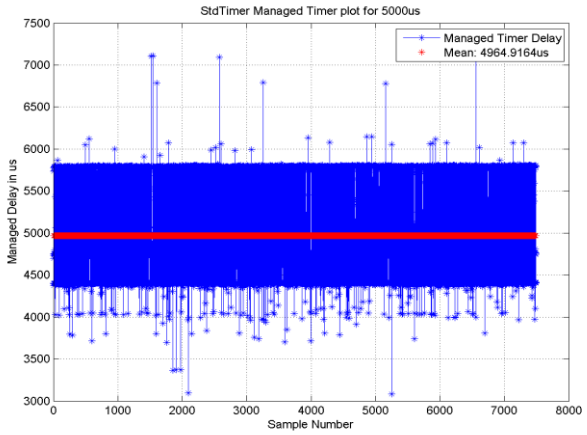


Figure 6. Mean delay for the Standard Timer for 5ms (5000 microseconds) timer

Figure 7. Mean delay for the RealTimeTimer for 0.5ms (500 microseconds) timer.

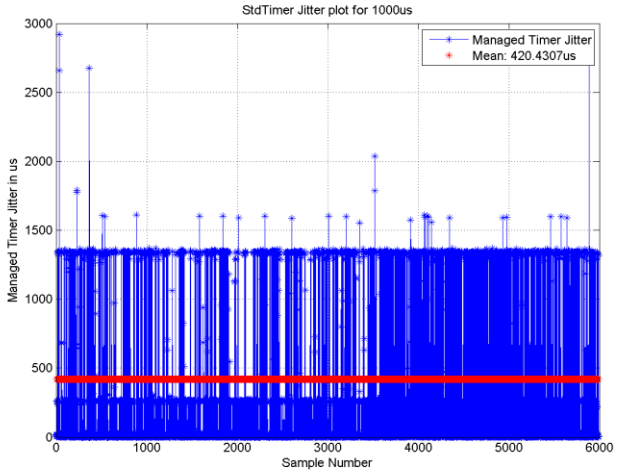


Figure 8. Mean Jitter for the Standard Timer for 5ms (5000 microseconds) timer.

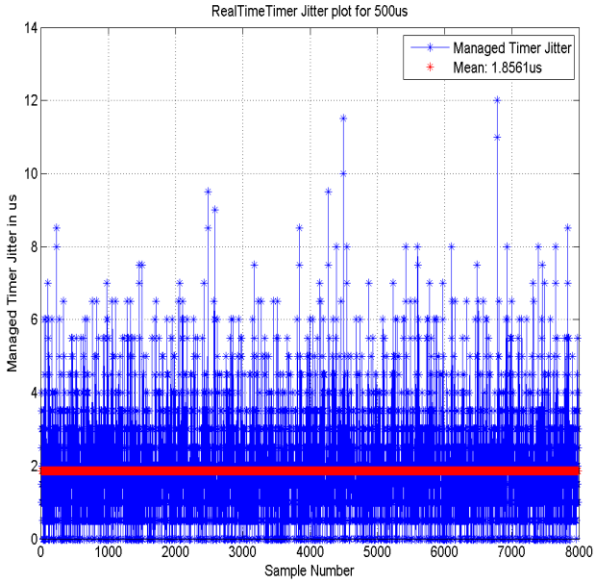


Figure 9. Mean Jitter for the RealTimeTimer for 0.5ms (500 microseconds) timer

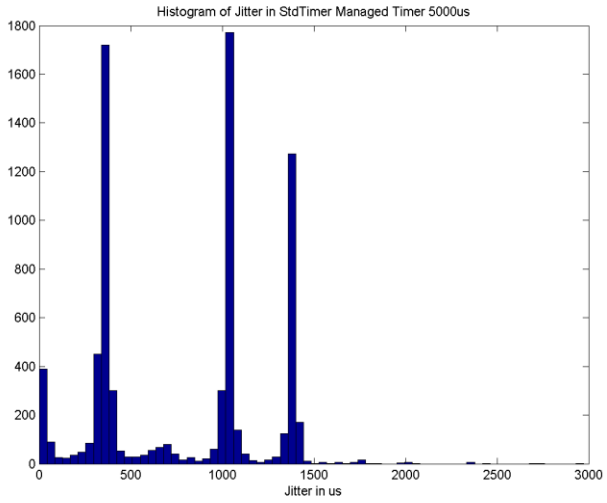


Figure 10. Histogram of Jitter for the Standard Timer for 5ms (5000 microseconds) timer.

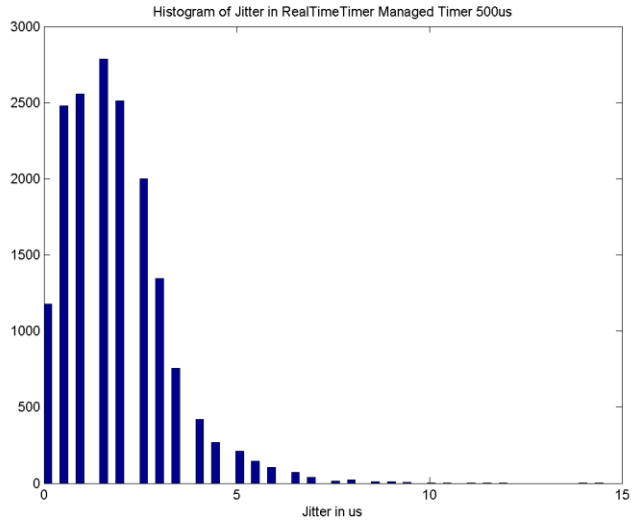


Figure 11. Histogram of Jitter for the RealTimeTimer for 0.5ms (500 microseconds) timer.

Guidelines for C# Application Developers Using the Real Time Timer

The following are guidelines for the developers interested in using the RealTimeTimer API to extract the maximum real time guaranty and without getting affected by the Garbage Collector.

1. **Create only a single RealTimeTimer:** If you create a second one, only the second one will be operational. That is, the handler of the first timer will not be called.

2. **Preallocate all variables/objects:** All variables and objects used inside the RealTimeTimer handler must be allocated as static or global. A realtime timer handler written in such a way uses minimum heap space while executing.
3. **Keep the code inside the RealTimerTimer handler small and simple:** This timer is not designed for doing large amounts of computation. Remember that none of the other threads can interrupt this operation. While hardware interrupts during this time will be serviced, their managed code counterparts will not be and if the CLRs interrupt queue overflows you could lose interrupts.

Table 5. The C# RealTime Timer APIs for use on eMote platforms.

Function Call	Description
Timer(string strDrvName, ulong Period, int Delay)	Constructor: Used to create a new Realtime Timer. The strDrvName is the name of the interop driver and it should be "RealTimeInteropTimer". The Period specifies the timer period in microseconds and the last parameter Delay specifies the time in microseconds before the first time the timer fires
public static bool Change(uint32 dueTime, uint32 period)	Used to change the period of an existing timer. dueTime specifies the amount of time in microseconds to delay before changing the period. The period specifies the new period in microseconds.
public static void Dispose()	Stops the timer and disposes the object.

C# Real Time Timer API

The namespace is Samraksh.SPOT.RealTime. It consists of a single class called Timer. While the operation of this class is semantically same as a regular Timer, it is actually derived from the Microsoft.SPOT.Hardware.NativeEventDispatcher class and need to be instantiated using its syntax. **Table 5** presents the API methods associated with the class Timer.

```
Samraksh.SPOT.RealTime.Timer RT_Timer;
RT_Timer = new Timer("RealTimeInteropTimer", 500 , 0);
NativeEventHandler RT_EventHandler = new NativeEventHandler(RT_TimerCallback);
RT_Timer.OnInterrupt += RT_EventHandler;

static void RT_TimerCallback(uint data1, uint data2, DateTime time){
   //Do realtime stuff
}
```

Listing 1. Using the Realtime Timer on eMote .NOWs.

In order to attach a callback to the handler use the Microsoft.SPOT.Hardware .NativeEventHandler syntax as shown in **Listing 1**.

1.5 Pinout of the .NOW

The pinouts of the .NOW are shown in **Figure 12**.

Pinout of the first High Density Connectors is shown in **Figure 13**. Note in High Density Connector #2 pinout, if JTAG is used, then pin 15 (gpio pin) cannot be used.

Pinout of the second High Density Connectors are shown in **Figure 14**.

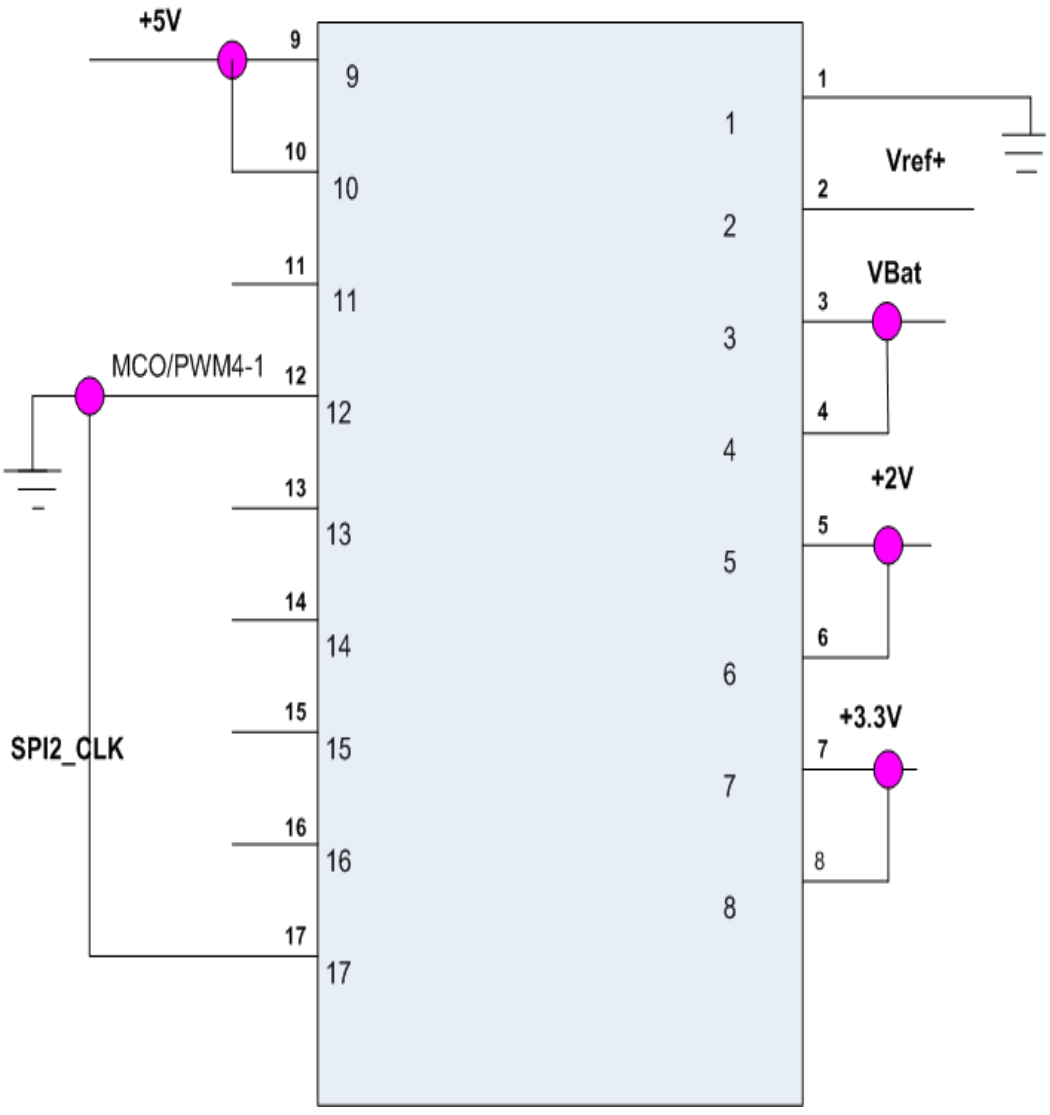


Figure 12. Pinout for J9.

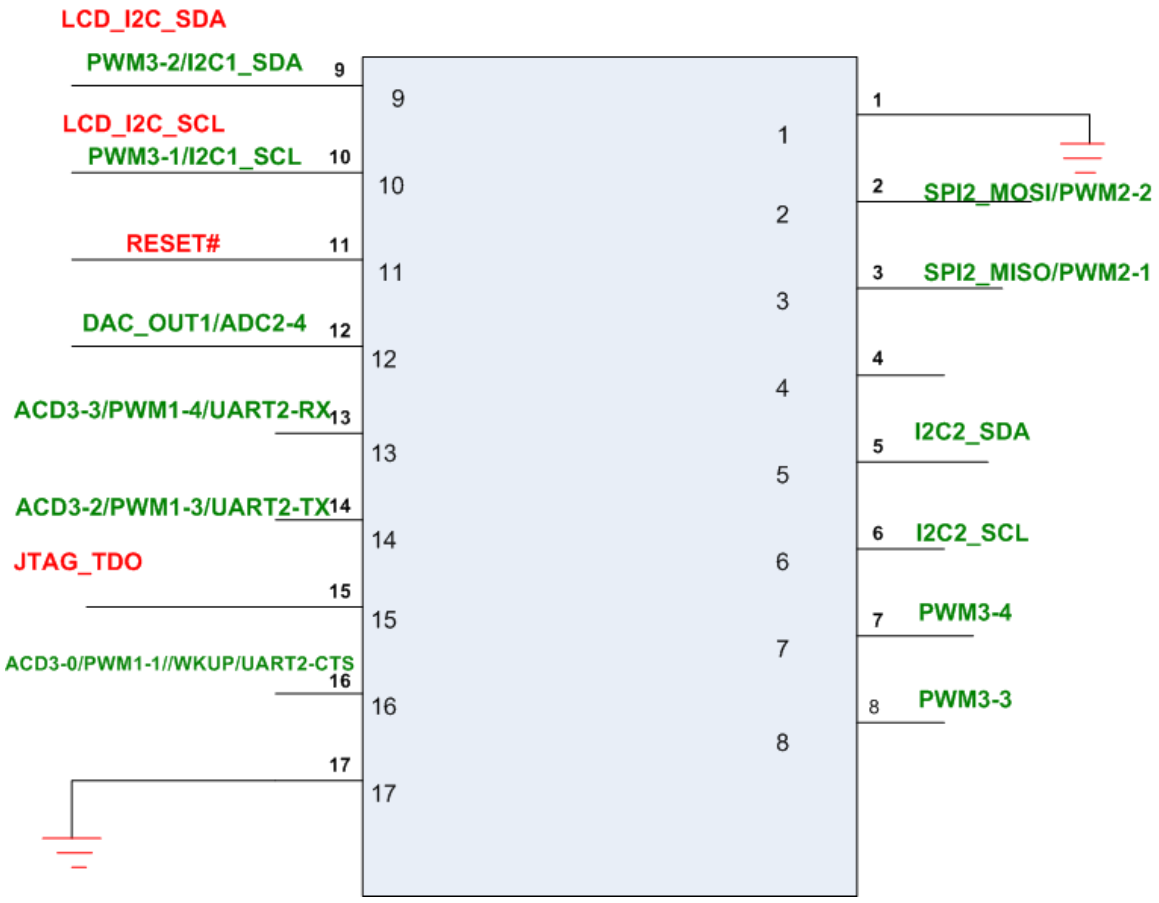


Figure 13. Pinout for the second high density connector.

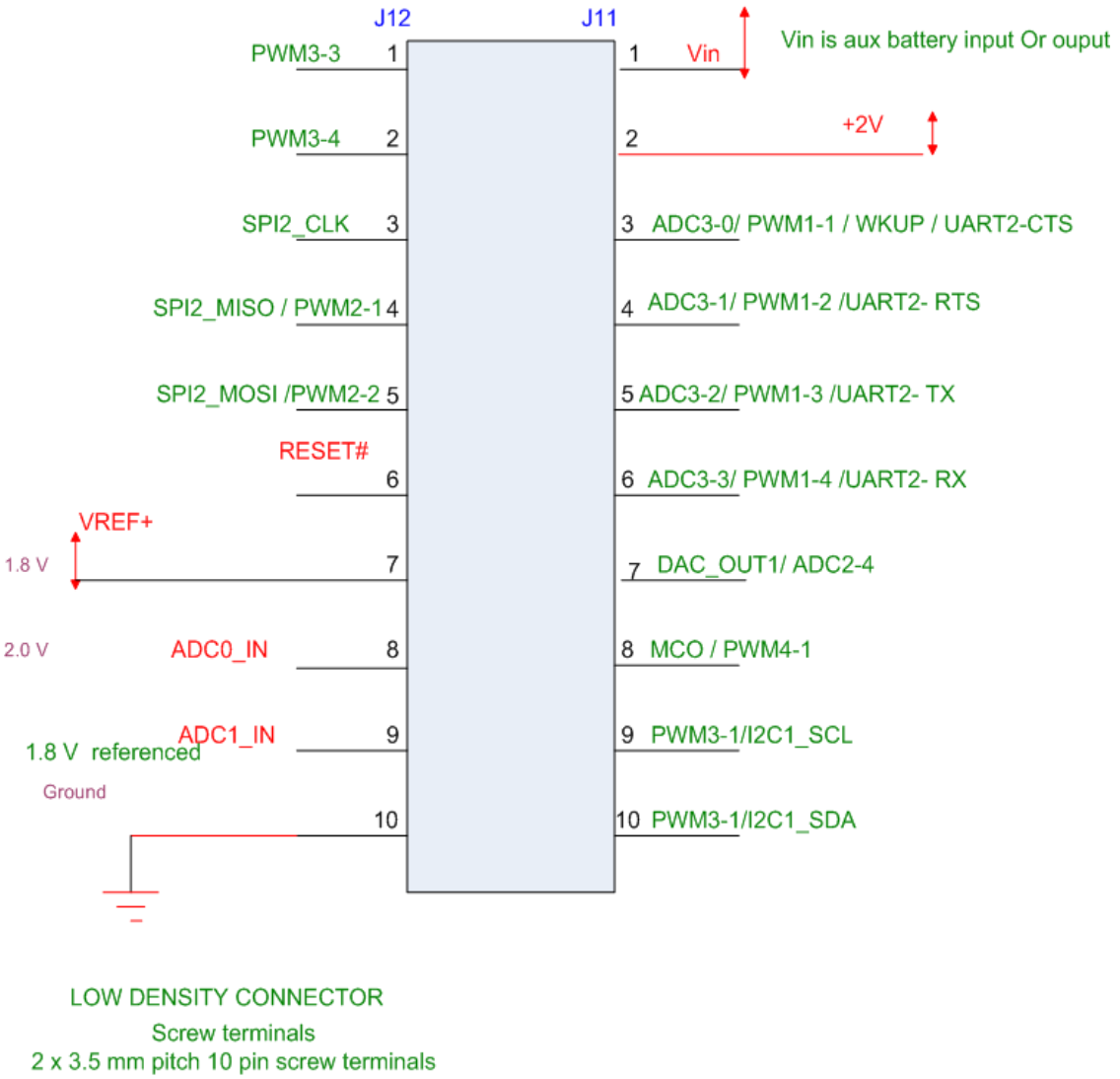


Figure 14. Pinout of the low density connector.

1.6 Using Low Density Connectors in Applications on the eMote .NOW

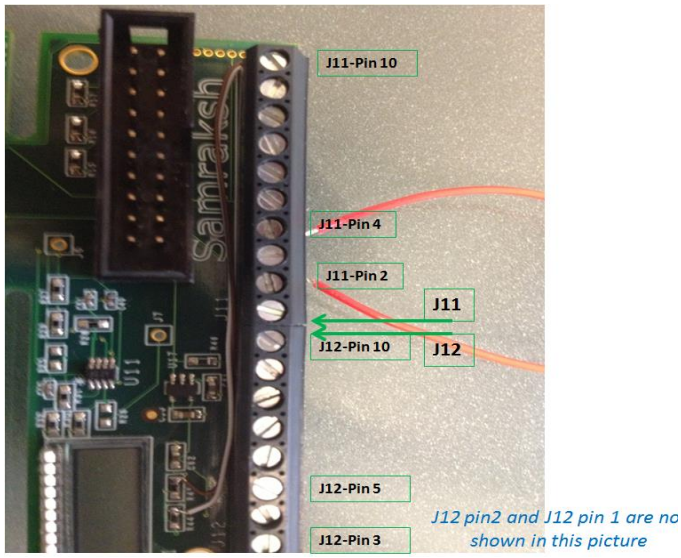
Section 2 provides the information on the software installation needed for using the .NOW, while Section 3 looks at applications that could be hosted on the eMote .NOW. Before proceeding to Section 2, we provide a table on the mapping of the low density connector pins to the software expression needed to use them in a C# application.

This information will be useful in building C# applications, as presented in Section 3, but is presented here to emphasize the connection to the pin out in Figure 14. See Table 5 below. Also, Figure 15 shows the pin locations on a photo of the eMote .NOW.

Table 6 Mapping of Programmable Pins on the Low Density Connector to Their C# Software Representation.

Programmable Hardware Pin (Digital)	Software Designation
J11	
Pin 3	(Cpu.Pin)0
Pin 4	(Cpu.Pin)1
Pin 5	(Cpu.Pin)2
Pin 6	(Cpu.Pin)3
Pin 7	(Cpu.Pin)4
Pin 8	(Cpu.Pin)8
Pin 9	(Cpu.Pin)22
Pin 10	(Cpu.Pin)23
J12	
Pin 1	(Cpu.Pin)24
Pin 2	(Cpu.Pin)25
Pin 3	(Cpu.Pin)29
Pin 4	(Cpu.Pin)30
Pin 5	(Cpu.Pin)31

Figure 15 Pin Locations on the physical .Now low density connector. Shown are J11 and J12 pins.



2. Getting Started Using the eMote .NOW

2.1 Software Installation

The eMote software has been built using the Micro Framework Version 4.3 porting kit.

The eMote already comes with the firmware installed; this includes the TinyCLR that was developed for the eMote firmware.

On samraksh.com, we have the Samraksh-provided managed code dynamic linked libraries (DLLs) for using with your C# applications. These libraries provide custom APIs that Samraksh has built for capabilities such as networking, "real time" timers and A-to-D converters. [Real time timing capability is to within an average of 4 microseconds jitter.] While the Micro Framework SDK 4.3 offers the standard Micro Framework APIs, and you can program eMote using these, you will be able to use the added features & advanced functionality of the eMote only with the Samraksh DLL. In particular, three DLLs will be available:

1. `Samraksh.SPOT.Hardware`
2. `Samraksh.SPOT.RealTime`
3. `Samraksh.SPOT.Net`

The LCDs and ADC API are addressed by the first DLL. The wireless communication is supported by the third DLL. Real time activity is supported by the second DLL.

The LCDs and ADC API are addressed by the first DLL. The wireless communication is supported by the third DLL. Real time activity is supported by the second DLL. We have already presented the API offered in the Samraksh.SPOT.Net. A sub-set of the LCD API is given in the section that discusses writing applications using the LCD. The real time API will be given in the future along with an App Note in that area.

The software environment for the user on the eMote .NOW platform is the Micro Framework SDK 4.3. The following are the installation steps.

2.2 Getting Started

Step 1: Visual Studio 2012 Windows Desktop is used as a C# development environment for the eMote. If you do not have it installed, then you can install the free Express version from

http://www.microsoft.com/visualstudio/eng/downloads

(The paid version is offered first; please scroll down to find the Visual Studio 2012 Express for Windows Desktop version.)

Step 2: Next, install the Micro Framework SDK 4.3 from

http://netmf.codeplex.com/releases/view/81000

Step 3: Install the Serial over USB driver. Connect a standard Micro-B USB cable to the connector marked "Serial" on the eMote to your machine. Typically, the operating system should be able to download the necessary driver and install them. If there are problems with the driver installation, you can get them from

http://www.silabs.com/products/mcu/Pages/USBtoUARTBridgeVCPDrivers.aspx

The driver should now appear under Ports in Window's Device Manager (which you can access from Control Panel).

Step 4: You are now ready to write C# applications for the .NOW platform using the Visual Studio 2012 Windows Desktop. We assume the Express version, but procedures are similar for the paid versions.

Step 5: Click on Start, Open Visual Studio Express 2012 for Desktop and Select New Project. A screenshot of Step 5 is shown in **Figure 16**.

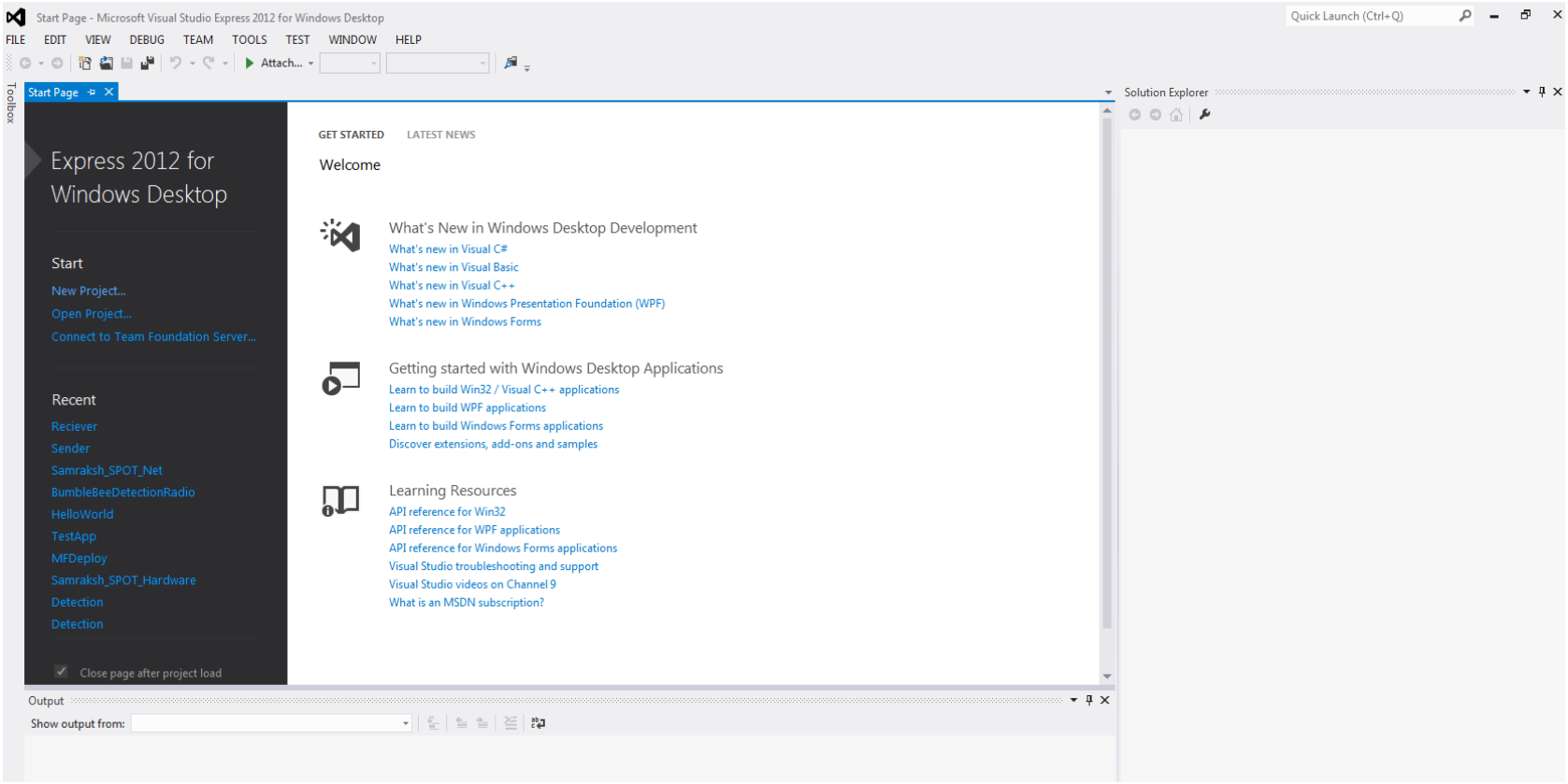


Figure 16. Screenshot of Visual Studio Express 2012 for Windows Desktop when first opened.

2.3 Building A C# Application

Step 6: Under Templates towards the left, Select Visual C# -> MicroFramework and then Select Console Application. See Figure 17.

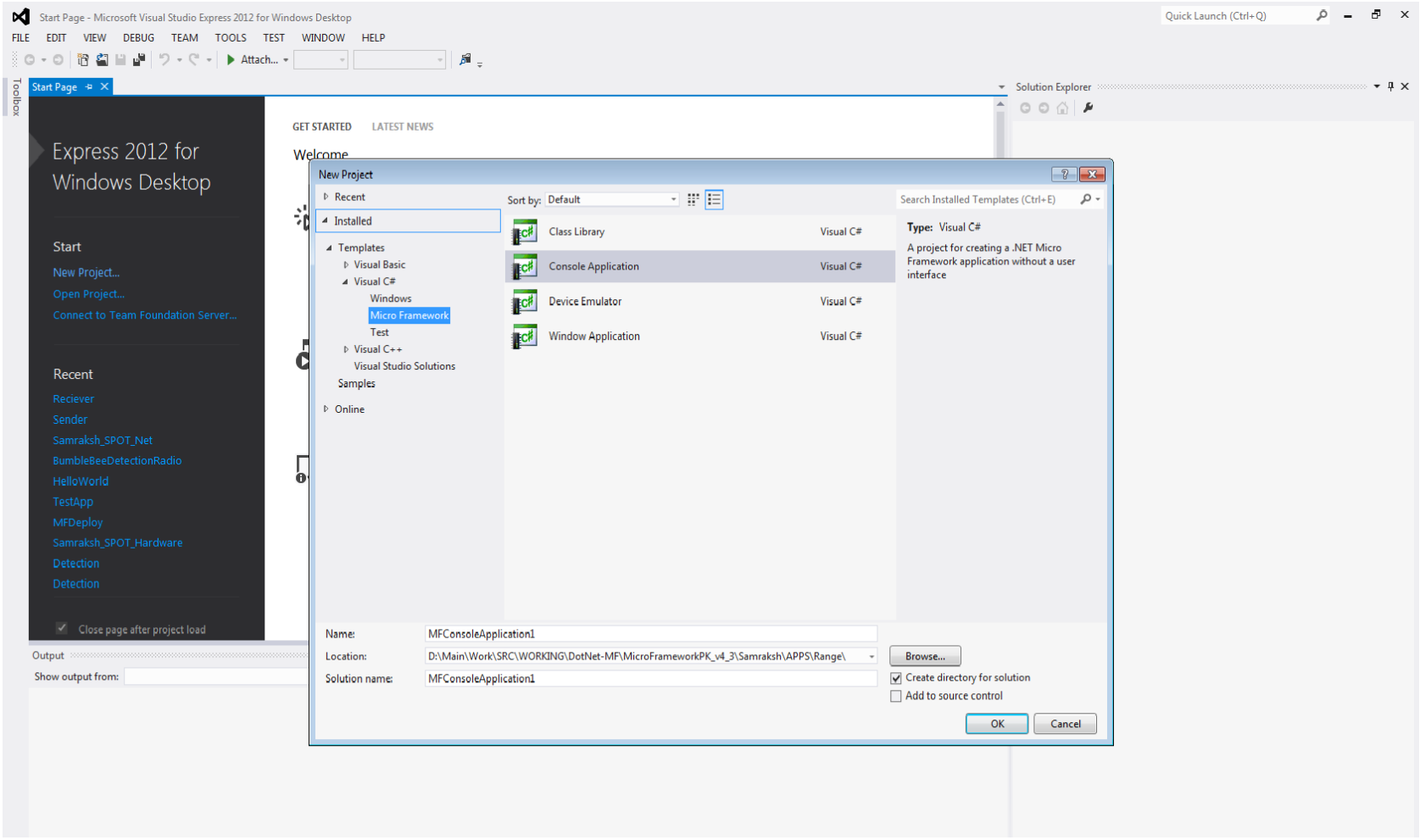


Figure 17. Screenshot of Step 6. Select the console application.

Step 7: Select a name for your application and select the location where you wish to store the application and select OK. This should open the project as shown in the figure below. See Figure 18 and Figure 19.

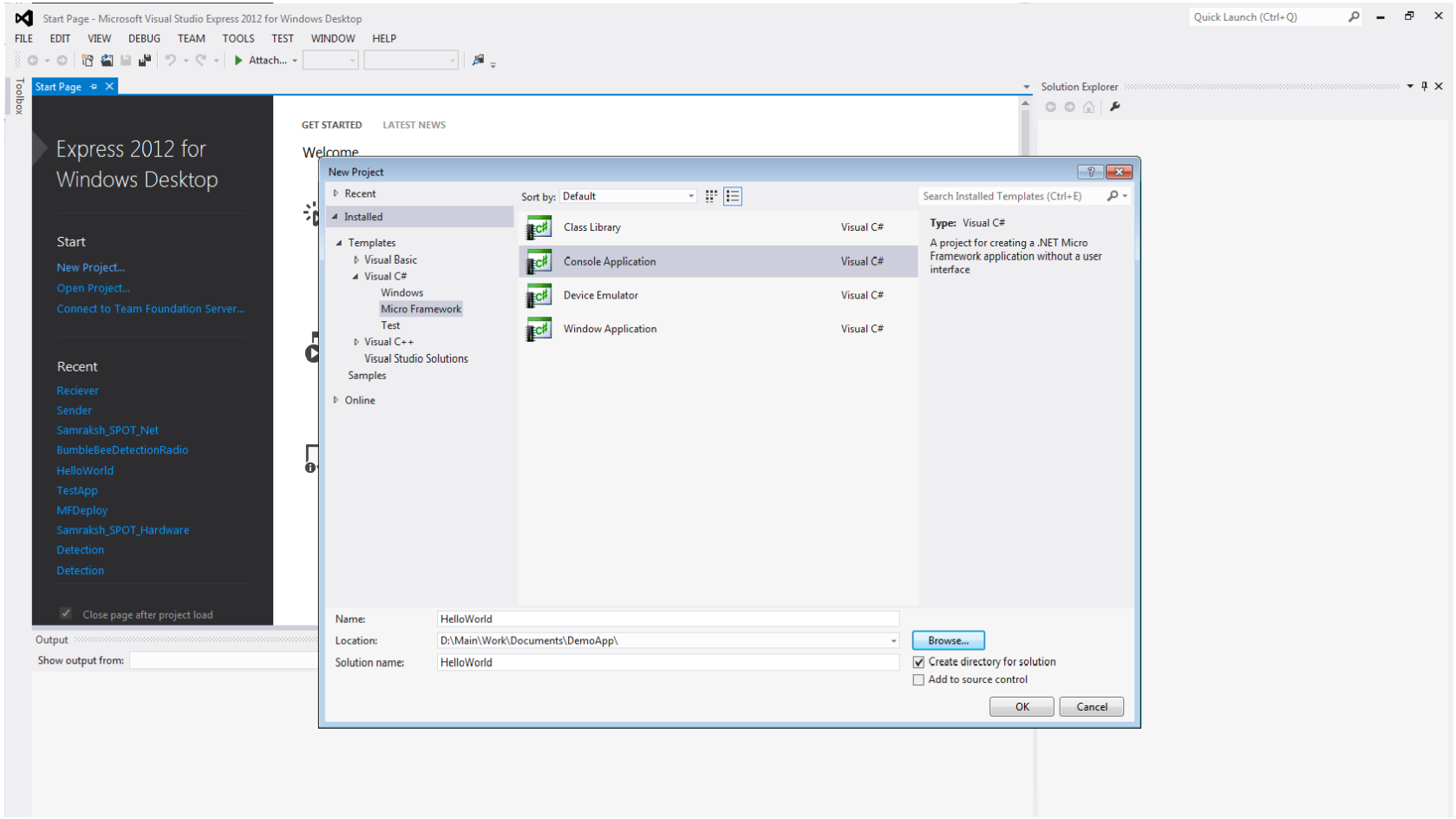


Figure 18. Screenshot 1 of Step 7: Setting up the project using Visual Studio Express 2012 for Windows Desktop.

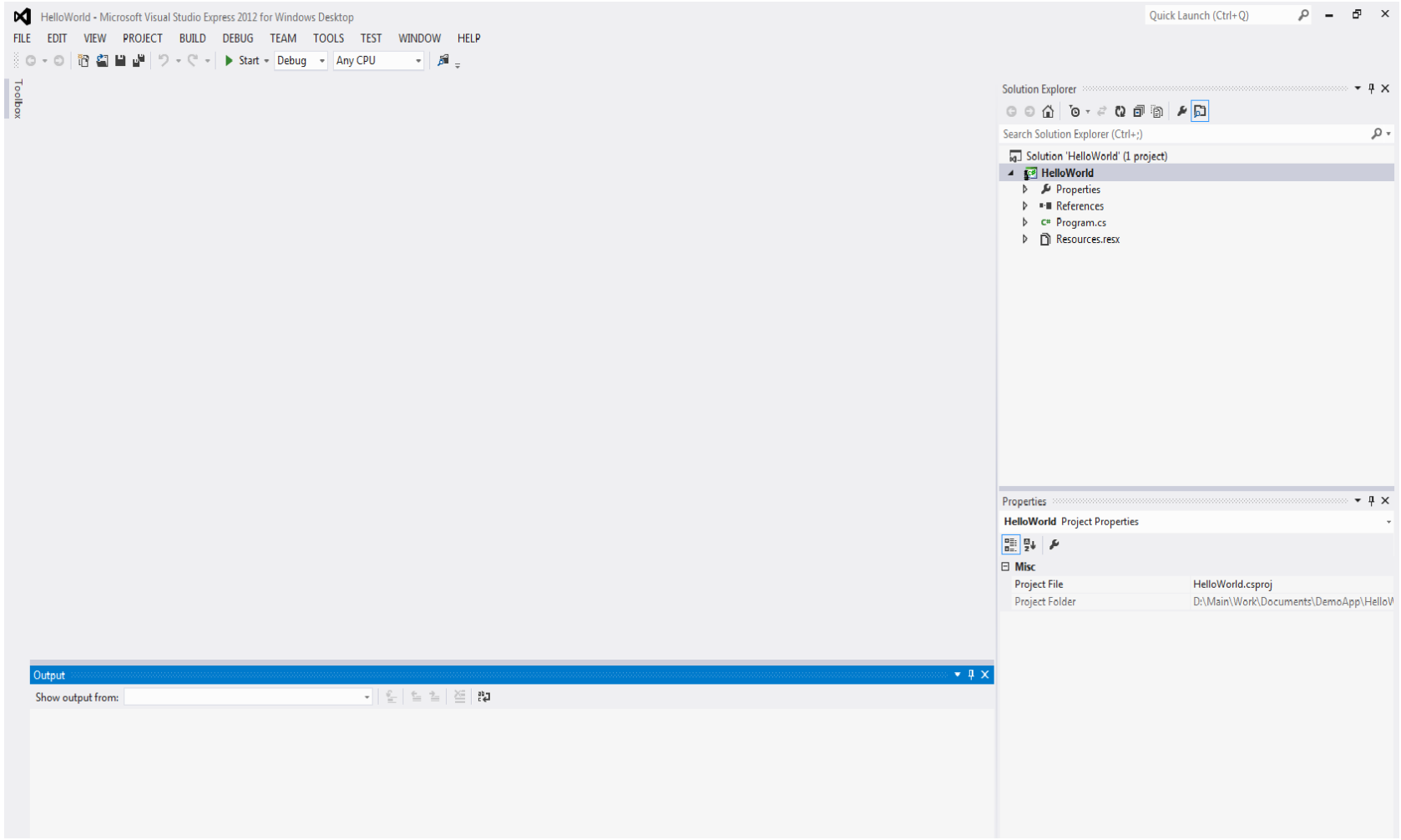


Figure 19. Screenshot 2 of Step 7. Note the right hand side.

Step 8: Click on Program.cs on the right menu and write the application you wish to deploy on the eMote .NOW. In Figure 20 we have a simple HelloWorld application.

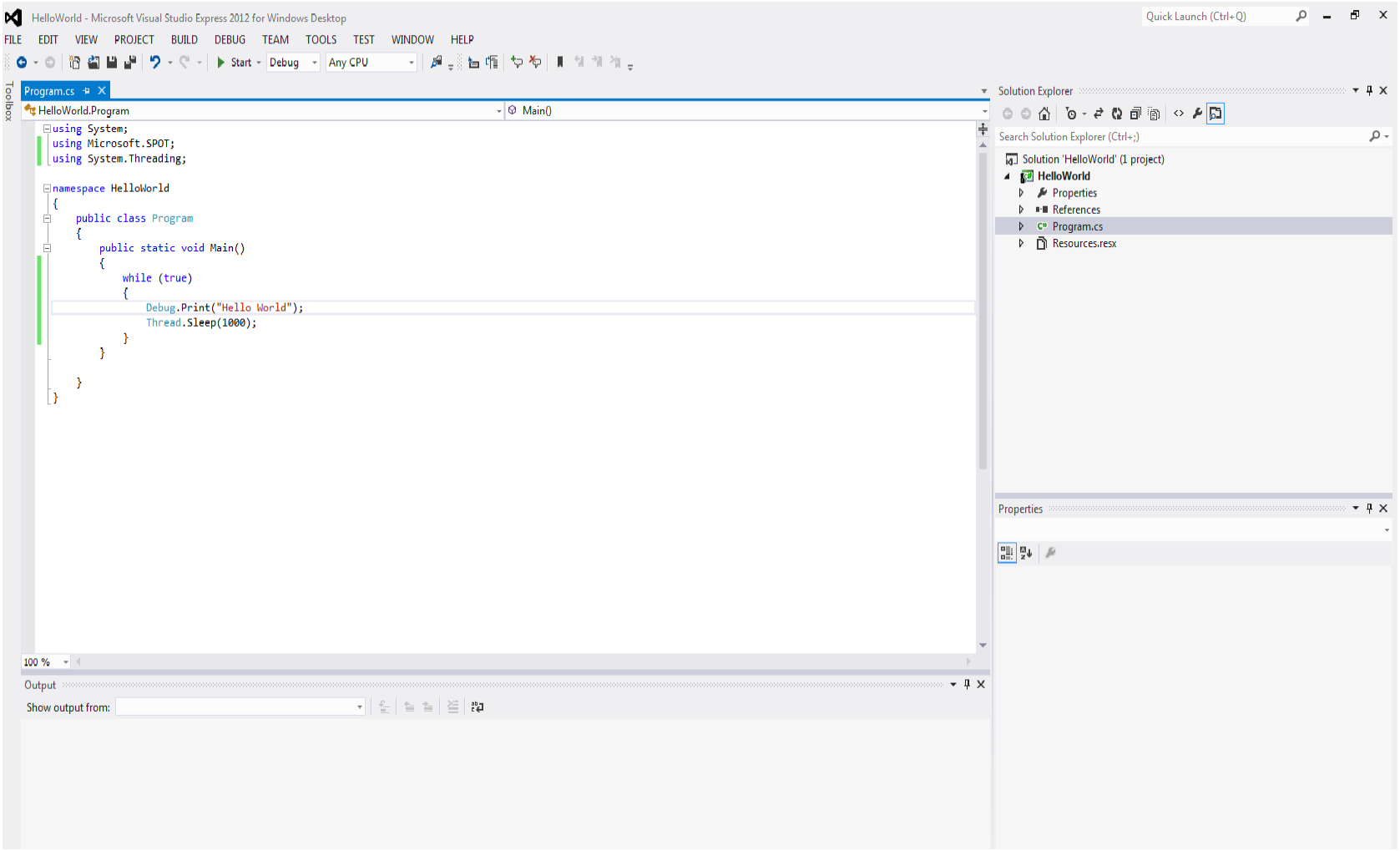


Figure 20. Screen-shot of Step 8. Starting *Hello World* code details.

Step 9: Once app development is complete, click on Build Solution under BUILD to compile the application. See Figure 21.

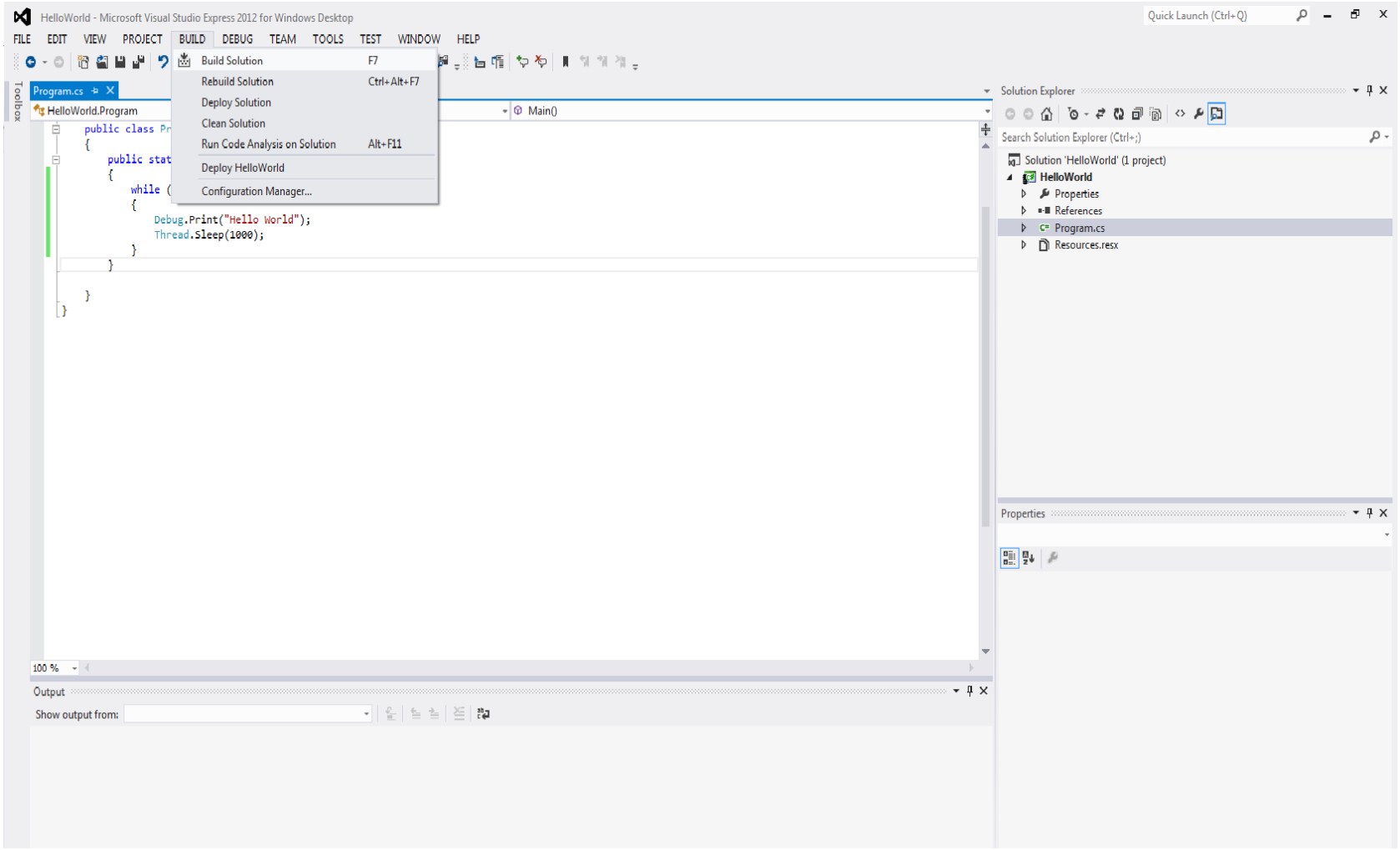


Figure 21. Screen-Shot of Step 9. Building the Application.

Step 10: If the application build successfully, you should be able to see the following output, shown in Figure 22. Note the information in the lower left hand side.

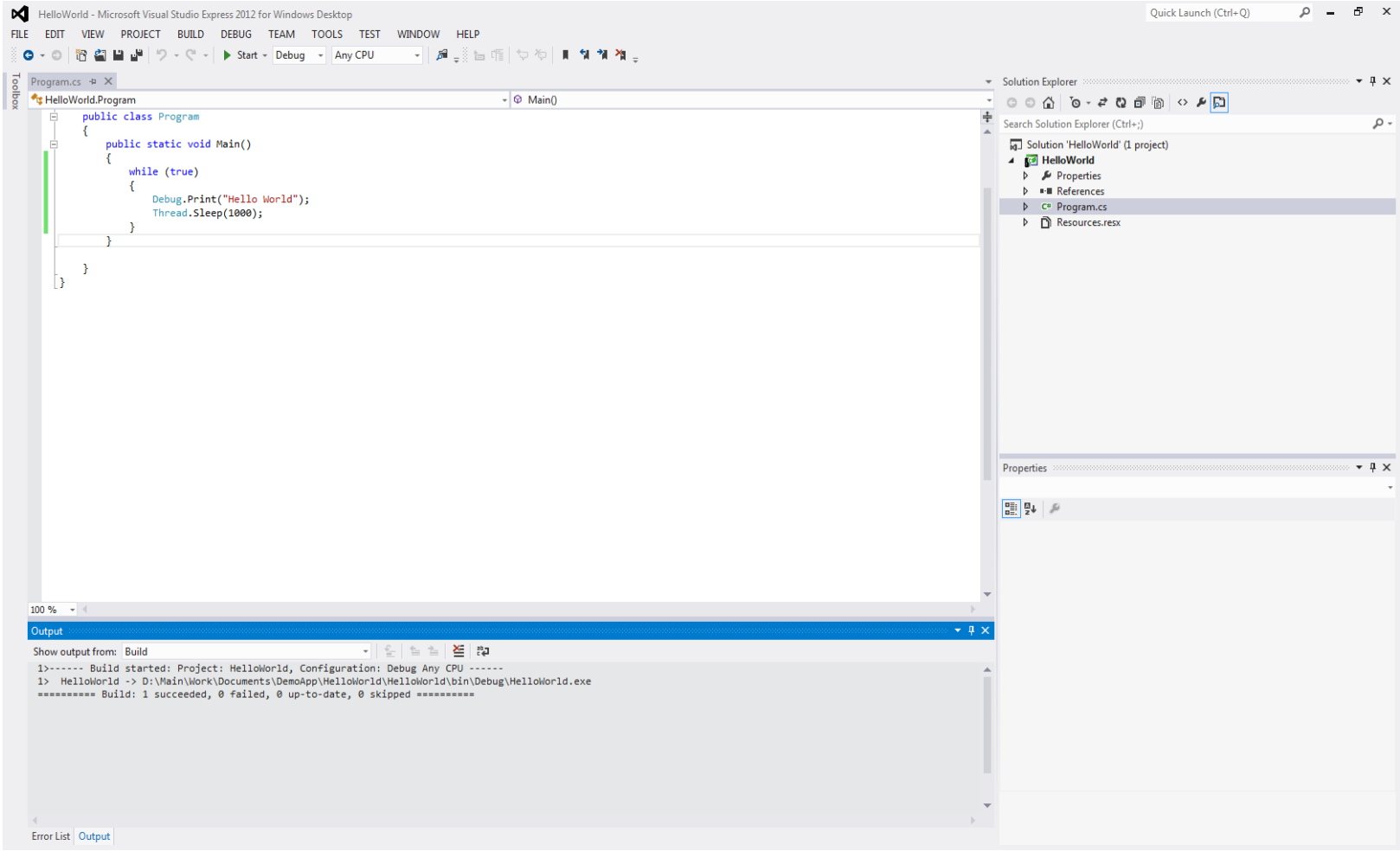


Figure 22. Screen-shot of Step 10. Success at building the application.

2.4 Deploy the C# Application on the eMote

Step 11: The next step is to deploy the application on the eMote. Select PROJECT-> HelloWorld Properties… as shown in Figure 23 below.

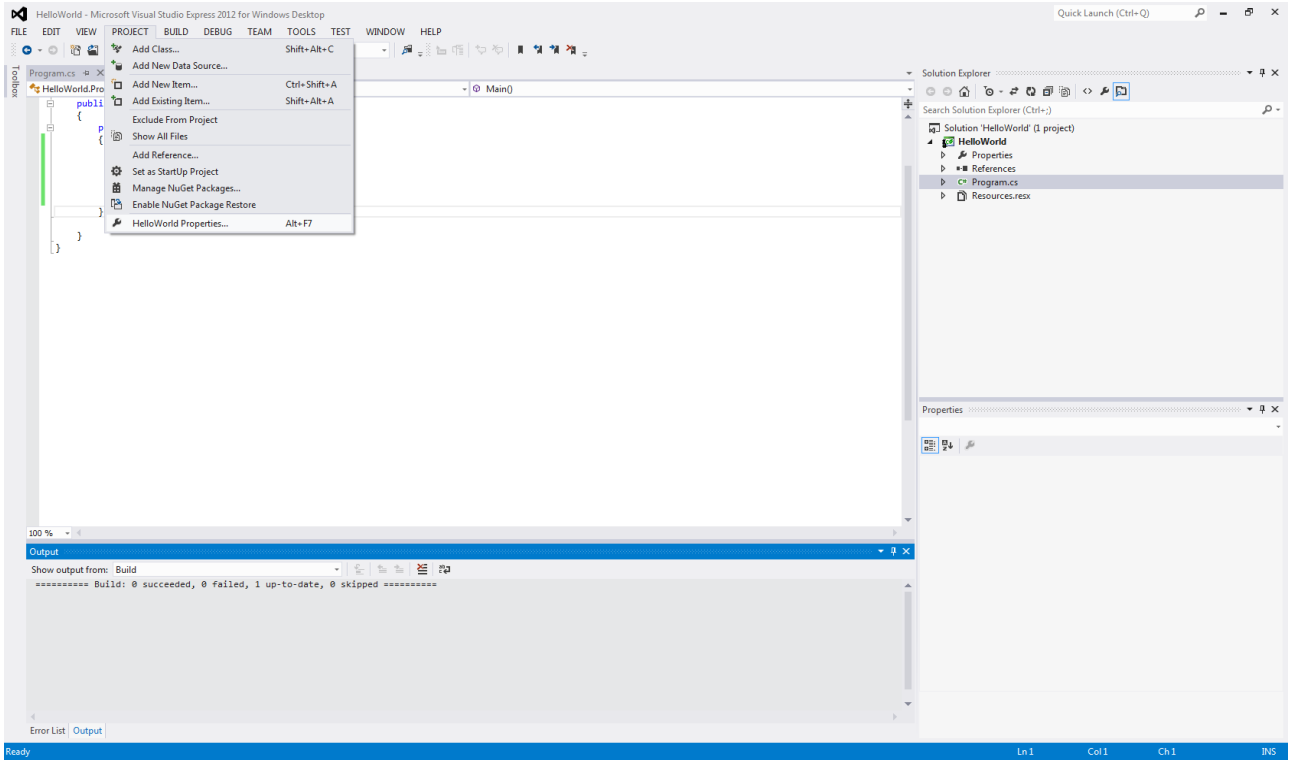


Figure 23. Screen shot of Step 11: The Initial Step in Deploying the Application to the eMote.

Step 12: The next step is to deploy the application on the eMote. Select the .Net Microsoft Emulator shown in Figure 24 below.

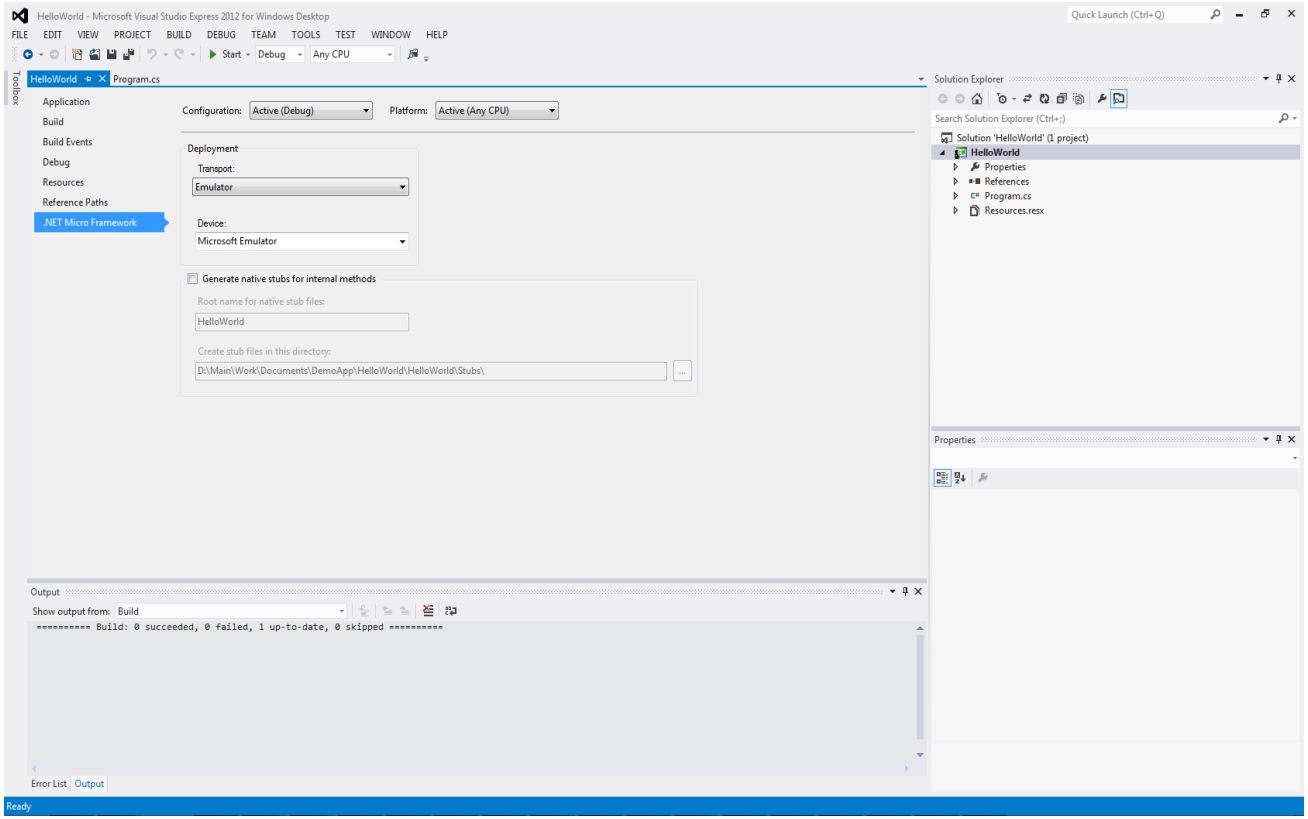


Figure 24. Screen shot of Step 12: The Initial Step in Deploying the Application to the eMote: Select the .Net MICRO Framework.

Step 13: Under Transport, Select Serial as shown in Figure 25 below.

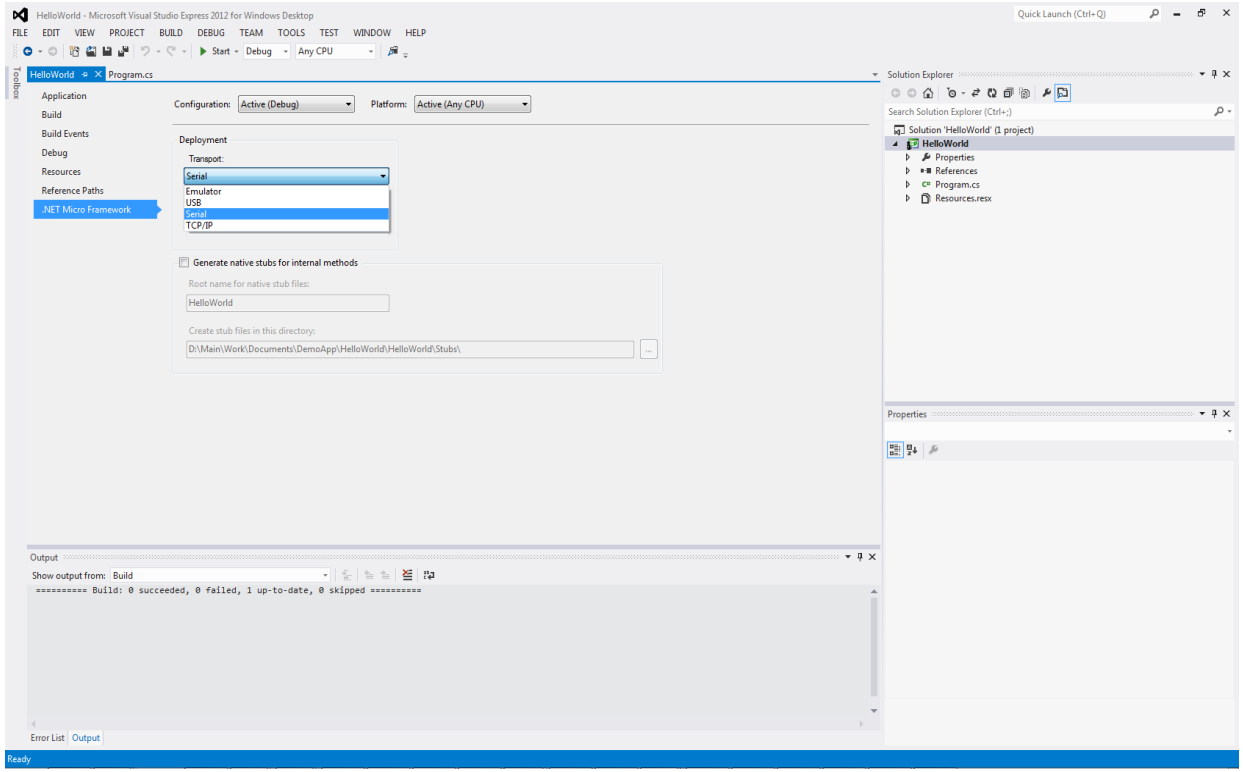


Figure 25. Screen shot of Step 13: The Initial Step in Deploying the Application to the eMote: Select the .Net MICRO Framework.

Step 14: Under device, in the Visual Studio 2012 for Windows Desktop view as shown in Figure 26, select the appropriate COM port that the eMote is connected on. (These may be viewed directly by checking under Device Manager. The Device Manager is accessible – in Windows 7 – by accessing the Control Panel and then selecting 'System and Security' and then selecting 'System' and finally select 'Device Manager' (and look under Ports)).

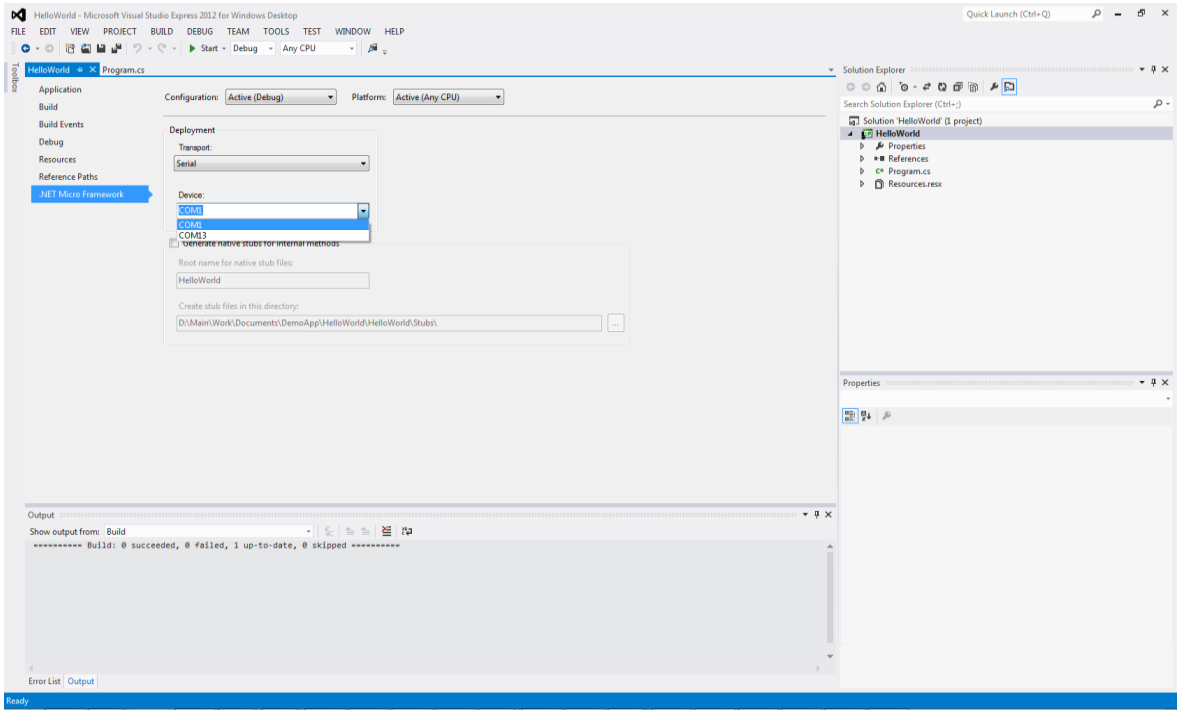


Figure 26. Screen shot of Step 14: Selecting the COM port; part of the application deployment process.

Step 15: Once this is selected, click on FILE-> Save ALL, to ensure that our new configuration has been saved.

Step 16: The next step is to deploy the application, click on BUILD -> Deploy HelloWorld. This is shown in Figure 27.

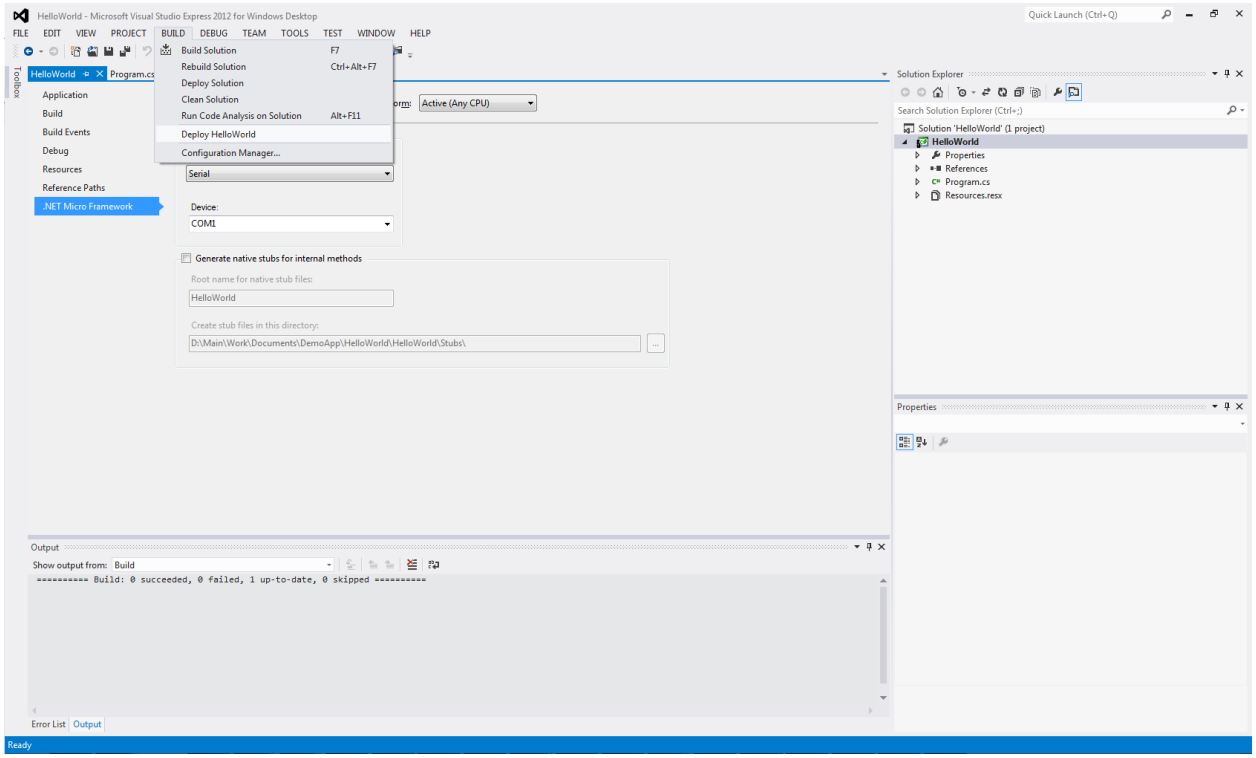


Figure 27. Screen shot of Step 16: Deploying 'Hello World'

Step 17: If the deployment is successful, you should be able to see the output below in Figure 28. If there are errors at this point, please try rebooting the eMote .NOW by power cycling it (disconnecting and reconnecting the power source), and then try step 16 again.

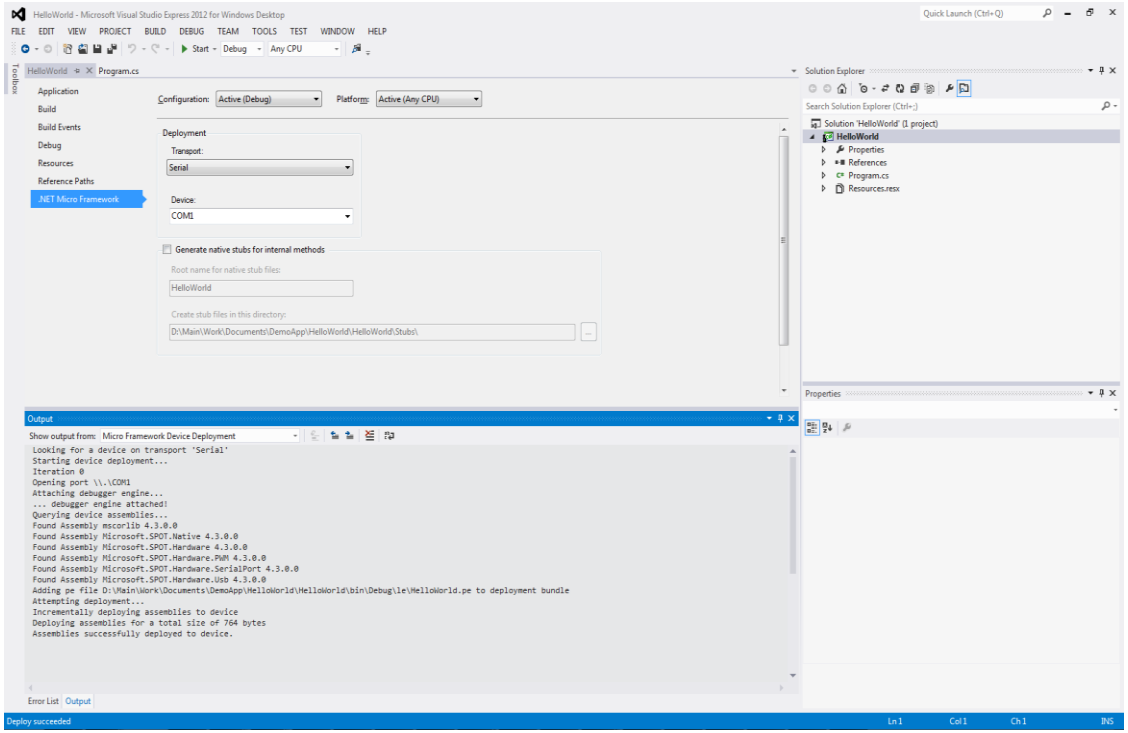


Figure 28. Screen shot of Step 17: Success in Deploying the Application

Step 18: Once deployment is complete, reboot the emote by power cycling and you should be able to see the output on any terminal program. In this case we show the output on MFDeploy. To do this, open MFDeploy by typing MFDeploy under start menu. Select Serial under Device and Select the appropriate COM port from the device manager. Choose Target-> Connect. At this point you should be able to see the output. See Figure 29. (If you don't see it, just click on Ping and you should be able to see "Hello World" being printed on the screen.)

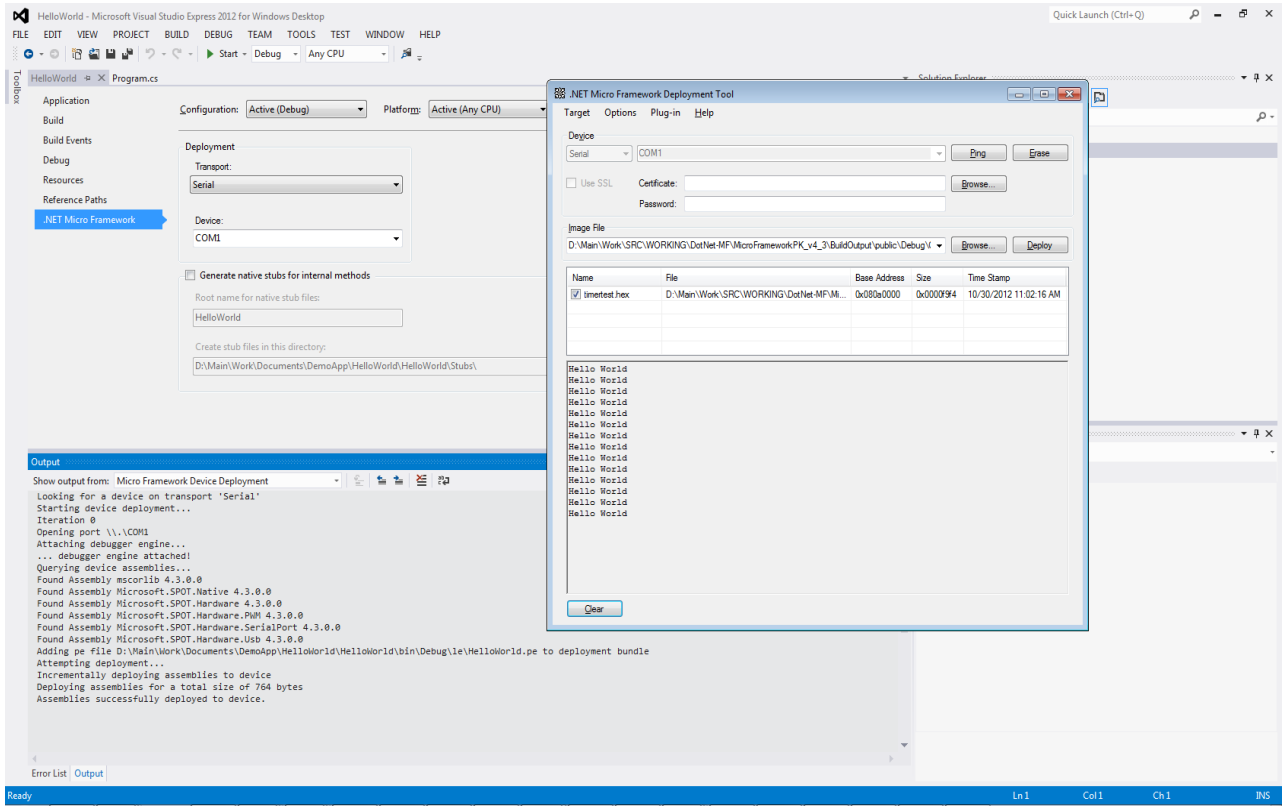


Figure 29 Screen shot of Step 16: Deploying 'Hello World'.

Next steps:

Step N1: Obtain the custom DLLs from the Samraksh web-site at https://samraksh.com/support/downloads .

Step N2: Install the DLLs in the appropriate directory on your computer, according to the IDE you are using.

Step N3: Use the API that are supported by the DLL as per your application needs.

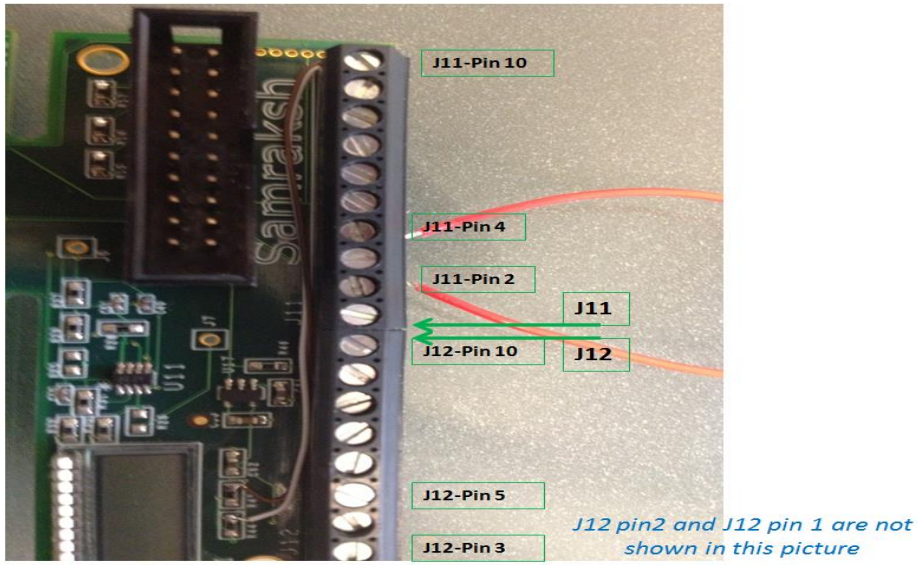
Step N4: Compile your application; it should compile (absent any programming errors).

You can then deploy your application to the eMote .NOW by using the Visual Studio Express 2012 capability as shown previously in Figure 28. Your application may be debugged (using debugging statements) using the MFDEPLOY tool, as shown in Figure 29.

3. Using the TinyBooter

The TinyBooter enables the user to install TinyCLR updates and recover from crashes where the Emote does not respond to pings from MFDeploy. Use the following steps to boot into the TinyBooter.

1. Short (via a connecting wire) pins 2 and 4 on J11 as shown in the image below.
2. Reset the emote using the reset button or by disconnecting the power source and connecting it back.
3. Ping the mote from MFDeploy.
4. You should be able to see Pinging ….. TinyBooter on the MFDeploy output.



3.1 Updating the CLR

After doing the above steps, If you are attempting to apply an update to the CLR,

1. Navigate to the folder containing the update by hitting the Browse button.
2. Once you have selected the new version of the CLR, hit Deploy.

   This should deploy the new version of the CLR. If the update fails please try a few more times. In the event that the device refuses to deploy the CLR, please contact support@samraksh.com

3.2 Recovering from a crash or RealTime App deployment.

As a result of special needs of the real time extensions in Micro framework, there are instances where you find yourself unable to connect to the emote or deploy from Visual Studio after deploying an app exercising the real time extension. This is also applcable to scenarios where an application not using the real time extension has caused the mote to not respond. In both these cases,

1. Once you connect to the TinyBooter, Hit Erase on MFDeploy
2. Select the deployment sectors and hit Erase

   This should erase the application causing the CLR to not respond. In event that the erase fails or the problem persists even after erasing the application, please contact support@samraksh.com

4. Application 1: Using the Radio with a Ping Program

The code for the Ping application is shown in Table 7. This application uses the Samraksh.SPOT.Net library.

Table 7. Code for Ping Application

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

using Samraksh.SPOT.Net;
using Samraksh.SPOT.Hardware.EmoteDotNow;

namespace Samraksh.SPOT.Net.Mac.Ping
{
    public class PingMsg
    {
        public bool Response;
        public ushort MsgID;
        public UInt16 Src;
        public UInt16 dummySrc;

        public PingMsg()
        {
        }
        public static  int Size(){
            return 5;
        }
        public PingMsg(byte[] rcv_msg, ushort size)
        {
            Response = rcv_msg[0] == 0 ? false : true;
            MsgID = (UInt16)(rcv_msg[1] << 8);
            MsgID += (UInt16)rcv_msg[2];
            Src = (UInt16)(rcv_msg[3] << 8);
            Src += (UInt16)rcv_msg[4];
        }
        public byte[] ToBytes()
        {
            byte[] b_msg = new byte[5];
            b_msg[0] = Response ? (byte)1: (byte)0;
            b_msg[1] = (byte) ((MsgID >> 8) & 0xFF);
            b_msg[2] = (byte)(MsgID & 0xFF);
            b_msg[3] = (byte)((Src >> 8) & 0xFF);
            b_msg[4] = (byte)(Src & 0xFF);
            return b_msg;
        }
    }

    public class Program
    {
        UInt16 myAddress;
        UInt16 mySeqNo = 0;
        Timer sendTimer;
```

```
        EmoteLCD lcd;
        PingMsg sendMsg = new PingMsg();

        //Radio.Radio_802_15_4 my_15_4 = new Radio.Radio_802_15_4();
        //Radio.RadioConfiguration radioConfig = new Radio.RadioConfiguration();
        //int myRadioID;
        static OutputPort SendPort = new OutputPort((Cpu.Pin)30, true);
        static OutputPort ReceivePort = new OutputPort((Cpu.Pin)31, true);

        static Mac.CSMA myCSMA = new CSMA();

        Mac.MacConfiguration macConfig = new MacConfiguration();
        ReceiveCallBack myReceive;

       void Initialize()
        {
            Debug.Print("Initializing:  EmotePingwLCD");
            Thread.Sleep(1000);
            lcd = new EmoteLCD();
            lcd.Initialize();
            lcd.Write(LCD.CHAR_I, LCD.CHAR_N, LCD.CHAR_I, LCD.CHAR_7);

            macConfig.CCA = true;
            macConfig.BufferSize = 8;
            macConfig.NumberOfRetries = 0;
            //macConfig.RadioID = (byte) myRadioID;
            macConfig.RadioID = (byte)1;
            macConfig.CCASenseTime = 140; //Carries sensing time in micro seconds

            myReceive = HandleMessage; //Assign the delegate to a function

            Debug.Print("Initializing:  CSMA...");
            try{
                  myCSMA.Initialize(macConfig, myReceive);
            }
            catch (Exception e)
            {
                Debug.Print(e.ToString());
            }
            Debug.Print("CSMA Init done.");
            myAddress = myCSMA.GetAddress();
            Debug.Print("My default address is :  " + myAddress.ToString());

            /*myCSMA.SetAddress(52);
            myAddress = myCSMA.GetAddress();
            Debug.Print("My New address is :  " + myAddress.ToString());
             */
        }

        void Start()
        {
            Debug.Print("Starting timer...");
            sendTimer = new Timer(new TimerCallback(sendTimerCallback), null, 0, 2000);
            Debug.Print("Timer init done.");
        }
```

```
        void sendTimerCallback(Object o)
        {
            if((mySeqNo % 50) == 0)
               Debug.Print("Sending broadcast ping msg:  " + mySeqNo.ToString());
            try
            {
                Send_Ping(sendMsg);
            }
            catch (Exception e)
            {
                Debug.Print(e.ToString());
            }
        }

        void HandleMessage(byte[] msg, UInt16 size, UInt16 src, bool unicast, byte
rssi, byte lqi)
        {
            try
            {
                if (unicast)
                {
                    Debug.Print("Got a Unicast message from src: " + src.ToString() +
", size: " + size.ToString() + ", rssi: " + rssi.ToString() + ", lqi: " +
lqi.ToString());
                }
                else
                {
                    Debug.Print("Got a broadcast message from src: " + src.ToString() +
", size: " + size.ToString() + ", rssi: " + rssi.ToString() + ", lqi: " +
lqi.ToString());
                }
                if (size == PingMsg.Size())
                {
                    PingMsg rcvMsg = new PingMsg(msg, size);

                    if (rcvMsg.Response)
                    {
                        //This is a response to my message
                        Debug.Print("Received response from: " + rcvMsg.Src.ToString()
+ "for seq no: " + rcvMsg.MsgID.ToString());
                        lcd.Write(LCD.CHAR_P, LCD.CHAR_P, LCD.CHAR_P, LCD.CHAR_P);
                    }
                    else
                    {
                        Debug.Print("Sending a Pong to SRC: " + rcvMsg.Src.ToString() +
"for seq no: " + rcvMsg.MsgID.ToString());
                        lcd.Write(LCD.CHAR_R, LCD.CHAR_R, LCD.CHAR_R, LCD.CHAR_R);
                        Send_Pong(rcvMsg);
                    }
                    ReceivePort.Write(true);
                    ReceivePort.Write(false);
                }
            }
            catch (Exception e)
            {
                Debug.Print(e.ToString());
            }
        }
```

```
        void Send_Pong(PingMsg ping)
        {
            UInt16 sender = ping.Src;
            ping.Response = true;

            ping.Src = myAddress;

            byte[] msg = ping.ToBytes();
            myCSMA.Send(sender, msg, 0, (ushort)msg.Length);

        }

        void Send_Ping(PingMsg ping)
        {
            ping.Response = false;
            ping.MsgID=mySeqNo++;
            ping.Src = myAddress;

            SendPort.Write(true);
            SendPort.Write(false);

            byte[] msg = ping.ToBytes();
            myCSMA.Send((UInt16)Mac.Addresses.BROADCAST, msg, 0, (ushort)msg.Length);
            int char0 = (mySeqNo % 10) + (int)LCD.CHAR_0;
            lcd.Write(LCD.CHAR_S, LCD.CHAR_S, LCD.CHAR_S, (LCD)char0);
        }

        public static void Main()
        {
            Debug.Print("Changing app");
            Program p = new Program();
            p.Initialize();
            p.Start();
            Thread.Sleep(Timeout.Infinite);
        }
    }
}
```

Using this application, two or more eMote.NOWs which are within radio transmission range can discover each other. We term each eMote .NOW that is participating as a node. Each node will send a broadcast 'Ping' message when its timer fires. The timer in this case is set to fire every 2 secs. Each time a node sends a 'Ping' message it increments the 'seqNo' field inside the packet. When any node in the sender's transmission range receives the 'Ping' message, it responds back with a 'Pong' message to the source. The 'Pong' message is sent as Unicast and only the intended node can receive the message.

When a node receives a message intended for it (either unicast or broadcast) the ReceiveCallBack delegate is called by the TinyCLR. The initialization of the mac layers for each node is found in API which are offered through the Samraksh.SPOT.Net library. The settings that are shown in the code sample above may be used. The radio layer is automatically initialized when you initialize the Mac layer. No separate

radio initialization is required. Radio layer initialization is required only if you are writing an application directly on top of radio layer and not using the CSMA Mac implementation provided by Samraksh.

Note that the node's address is the MAC address, which is a UINT16. When the network layer is initialized the MAC address is by default initialized to a number based on the STM32 CPU's serial number. However you can reset this address to any 16-bit number using the MAC APIs.

The MAC configuration is shown below.

```
Mac.CSMA myCSMA = new CSMA();
 Mac.MacConfiguration macConfig = new MacConfiguration();
macConfig.CCA = true;
macConfig.BufferSize = 8;
 macConfig.NumberOfRetries = 0;
 macConfig.RadioID = (byte) myRadioID;
//Carrier sensing time in micro seconds
macConfig.CCASenseTime = 140;
//C# delegate is assigned to a fcn
 myReceive = HandleMessage;
 myCSMA.Initialize(macConfig, myReceive);
```

There are different types of MAC layer protocols, generally speaking. What is offered here is a 'CSMA' or a 'carrier sense multiple access' protocol. The CSMA protocol senses the physical carrier or channel for 140 microseconds before a message is sent over the airwaves; this is done to avoid collision of packets in the channel. If the channel is free the message is sent immediately (unlike some protocols which decide to transmit or not randomly). If the channel is active, then the protocol will wait for a random amount of time between 0 and 500 microseconds and try again. The maximum rate at which MAC protocol can send messages is 20 packets a second. This is because the protocol uses an internal scheduler which runs roughly once every 50 ms. If your application needs to send packets at rates more than 20 packets/sec, you can use the radio layer directly to send and receive messages.

5. Application 2: Using the LCD in an Application

An LCD is provided on the eMote .NOW. The LCD consumes little power relative to an LED and may be used in similar applications such as 'hello' or 'blink'. In this section, we list API that can be used to develop your application involving the LCD component. These are given in Table 8 and Table 9. All are in the namespace "`Samraksh.SPOT.Hardware.EmoteDotNow`".

```
namespace Samraksh.SPOT.Hardware.EmoteDotNow
```

Table 8 enum LCD

```
public enum LCD
    {
        CHAR_NULL,
        CHAR_A,
        CHAR_B,
        CHAR_C,
        CHAR_D,
        CHAR_E,
        CHAR_F,
        CHAR_G,
        CHAR_H,
        CHAR_I,
        CHAR_J,
        CHAR_K,
        CHAR_L,
        CHAR_M,
        CHAR_N,
        CHAR_O,
        CHAR_P,
        CHAR_Q,
        CHAR_R,
        CHAR_S,
        CHAR_T,
        CHAR_U,
        CHAR_V,
        CHAR_W,
        CHAR_X,
        CHAR_Y,
        CHAR_Z,
        CHAR_a,
        CHAR_b,
        CHAR_c,
        CHAR_d,
        CHAR_e,
        CHAR_f,
        CHAR_g,
        CHAR_h,
        CHAR_i,
        CHAR_j,
        CHAR_k,
        CHAR_l,
        CHAR_m,
        CHAR_n,
        CHAR_o,
        CHAR_p,
        CHAR_q,
        CHAR_r,
        CHAR_s,
        CHAR_t,
        CHAR_u,
        CHAR_v,
        CHAR_w,
        CHAR_x,
        CHAR_y,
        CHAR_z,
        CHAR_0,
        CHAR_1,
        CHAR_2,
        CHAR_3,
        CHAR_4,
        CHAR_5,
        CHAR_6,
        CHAR_7,
        CHAR_8,
        CHAR_9
    };
```

Table 9 enum Blink and API for the EmoteLCD

```
enum Blink {
 FAST,
 MEDIUM,
 OFF,
 SLOW
}
```

API for the EmoteLCD

```
    public class EmoteLCD
    {
        public EmoteLCD()
        {
        }

    public bool Clear() ;

    public bool Initialize();

    public bool Uninitialize()   //to allow display to be turned off

    public bool Blink(Samraksh.SPOT.Hardware.EmoteDotNow.Blink blinkType)

    public bool SetDP(bool dp1, bool dp2, bool dp3, bool dp4)

    public bool Write(Samraksh.SPOT.Hardware.EmoteDotNow.LCD data4,
Samraksh.SPOT.Hardware.EmoteDotNow.LCD data3,
Samraksh.SPOT.Hardware.EmoteDotNow.LCD data2,
Samraksh.SPOT.Hardware.EmoteDotNow.LCD data1)

    public bool WriteN(int column, Samraksh.SPOT.Hardware.EmoteDotNow.LCD
data)

    public bool WriteRawBytes( int b1, int b2, int b3, int b4)
  }
```