

Exchanging Data using Windows Mobile, Windows Communication Foundation, .NET Compact Framework and Exchange 2007

Mark Arteaga

OpenNETCF Consulting

June 2008

Contents

Introduction

Prerequisites

Why E-Mail?

Getting Things Done

Dispatch Application

Picture Sharing Application

Conclusion

Introduction

Several new features have been added to .NET Compact Framework 3.5 that you as a developer can take advantage of in your applications. An overview of what is new is available on [MSDN](#). One such feature is Windows Communication Foundation (WCF). For an introduction to Windows Communication Foundation, you should read [Chris Tacke's article](#). This article will focus on "Store and Forward messaging" using Exchange Server 2007.

There are two sample applications accompanying this article. The first sample is a typical Line of Business (LOB) application where a central dispatcher notifies field workers of a new customer request. The second sample is a peer-to-peer application where users can share photos they have taken with their Windows Mobile device. The techniques used in the second example could easily be used to share documents and other payloads in an enterprise scenario.

Prerequisites

- Visual Studio 2008
- Exchange 2007

For the device applications:

- .NET Compact Framework 3.5
- Windows Mobile 5.0 or later

For the desktop application:

- .NET Framework 3.5
- .NET Compact Framework 3.5 SDK

Why E-Mail?

You are probably wondering “*why use email to transport data? Why not use an ASP.NET Web service and have the device connect to that Web service?*” The simple answer is addressability. Mobile devices are just that – *mobile*. They cannot guarantee a constant connection unless you are within the walled garden of a corporate network, even then there may well be connectivity black spots. Also, many enterprise scenarios leverage public cellular networks, where not only is connectivity not guaranteed, but you face the bigger challenges of uncontrollable DHCP leases and carrier network address translation (NAT). In light of this, e-mail is a very compelling transport for a number of reasons. Most people already use email and many enterprises have already deployed Direct Push Email with Exchange so leveraging this mechanism for custom applications requires no changes to existing infrastructure. You can assume that application data will be delivered to the user’s device using Exchange Direct Push Email whether the user is behind the firewall or outside the firewall.

Getting Things Done

To use WCF with Exchange, all we do is add references to the assemblies listed below to our device and desktop projects:

- Device Project
 1. Microsoft.ServiceModel.Channels.Mail (.NET Compact Framework version)
 2. Microsoft.ServiceModel.Channels.Mail.WindowsMobile
 3. System.Runtime.Serialization
 4. System.ServiceModel
- Desktop Project
 1. Microsoft.ServiceModel.Channels.Mail (.NET Framework version)
 2. Microsoft.ServiceModel.Channels.Mail.ExchangeWebService
 3. System.Runtime.Serialization
 4. System.ServiceModel

One thing to note is that the Microsoft.ServiceModel.Channels.Mail.dll assemblies in the above lists are two separate assemblies - one for the device and one the desktop. We discuss this a bit more in the [Sharing Code Between Platforms](#) section below.

We have also created two helper classes to make it easier to use the mail channel to be able to send data between devices and desktop. These two classes are [WCFMessagingManager](#) and [XmlSerializerWrapper](#).

XmlSerializerWrapper

To serialize and de-serialize your data, WCF requires an object inherit from [XmlObjectSerializer](#). Desktop developers can use [DataContractSerializer](#) however this class is not available in the .NET Compact Framework. Instead, we will simply create an **XmlSerializerWrapper** class which inherits from XmlObjectSerializer and has an internal XmlSerialzier member variable:

```
public class XmlSerializerWrapper : XmlObjectSerializer
```

```

{
    XmlSerializer m_serializer;
    ....
}

```

In the constructor, we create a new XmlSerializer of the type we want to serialize as follows:

```

public XmlSerializerWrapper(Type type)
{
    m_serializer = new XmlSerializer(type);
}

```

We also override the WriteObject() and ReadObject() methods which WCF calls to serialize and de-serialize our object as follows:

```

public override void WriteObject(XmlDictionaryWriter writer, object graph)
{
    m_serializer.Serialize(writer, graph);
}

```

And to de-serialize as follows:

```

public override object ReadObject(XmlDictionaryReader reader)
{
    return m_serializer.Deserialize(reader);
}

```

If you want more information on this take a look at the following article [Using the XmlSerializer as an XmlObjectSerializer with WCF](#).

WCFMessagingManager

WCFMessagingManager is a class that makes it easier for developers to send and listen for messages using WCF. It derives from the Messaging class available in this [MSDN article](#), which outlines creating a mobile chat system using the Compact Framework and Windows Mobile.

The WCFMessagingManager class is a generic class that provides the flexibility to accept any type of object to send. With the Dispatch Application we use the DispatchMessage type and with the Picture Sharing application we use the PhotoData object. We will discuss these classes in more detail in the [Dispatch Application](#) and [Picture Sharing Application](#) sections.

Creating a WCFMessagingManager

To send our data objects we need to create our mail binding objects. On the desktop we create and ExchangeWebServiceMailBinding as follows:

```

ExchangeWebServiceMailBinding binding =
new ExchangeWebServiceMailBinding(new Uri(Properties.Settings.Default.ExchangeS
erver), new NetworkCredential(Email, Password));

```

And on the device we create a WindowsMobileMainBinding object as follows:

```
WindowsMobileMailBinding binding = newWindowsMobileMailBinding();
```

You will notice that on the desktop, you have to specify an Exchange server address and your network credentials for the email account. This data is used to poll the mail box for any messages sent with the appropriate channel names. On the Windows Mobile side you don't have to provide credentials since WindowsMobileMailBinding monitors the mail in the Outlook inbox on the local device.

To use the WCFMessagingManager class we create a new instance of the object and pass in a few parameters as follows:

```
WCFMessagingManager<PhotoData> m_messagingManager;  
m_messagingManager =  
newWCFMessagingManager<PhotoData>(binding, Settings.Default.IncomingChannel);
```

When we create the WCFMessagingManager, we explicitly set PhotoData as the type for the generic class and the for constructor parameters we provide the MailBindingBase (which both WindowsMobileMailBinding and ExchangeWebServiceMailBinding both inherit from) and the channel we would like to listen on. The channel must be unique to every user using the application, but you can share the same email address for sending messages.

In the constructor of WCFMessagingManager we first build an IChannelFactory using the mail bindings as follows:

```
//Create a param collection for both the input channel and output channel  
BindingParameterCollection param = newBindingParameterCollection();  
  
//Build the channel factory  
m_channelFactory = m_mailBinding.BuildChannelFactory<IOutputChannel>(param);  
m_channelFactory.Open();
```

We build the IChannelFactory using the MailBindingBase object passed in and open the IChannelFactory.

Once the IChannelFactory is created, we create a listener channel so we can listen to incoming messages.

```
//Build the channel listeners  
m_channelListener = m_mailBinding.BuildChannelListener<IInputChannel>(MailUriHelper.CreateUri(m_channelNameListen, ""), param);
```

The channel name used is the channelNameListen parameter passed in the constructor.

Sending Messages

WCFMessagingManager exposes a method to send data using e-mail as the transport, which is defined as follows:

```
////// Sends a message to the receiving end  
///</summary>  
///<param name="recipient"></param>  
///<param name="body"></param>
```

```
public void SendMessage(string recipient, string channel, T body)
{ ... }
```

The SendMessage method is fairly straightforward, requiring three parameters. The first one, recipient, is the destination email address that the message will be going to. The channel parameter is the channel the destination will be listening on. The body parameter is the object that will be sent to the destination. Notice that the body parameter is of a generic type, so in the case of Photo Sharing application, we will be sending a PhotoData object.

The implementation of SendMessage is also fairly straightforward:

```
//Use the channel factory to create an output channel
IOutputChannel outputChannel = m_channelFactory.CreateChannel(
newEndpointAddress(MailUriHelper.Create(channel, recipient)));

//Open the output channel
outputChannel.Open();

//Create a message to send via the channel
Message message = Message.CreateMessage(
MessageVersion.Default, UrnInternal,
body, m_xmlSerializerWrapper);

//Send the message
outputChannel.Send(message);

//Close the output channel
outputChannel.Close();
```

Using the IChannelFactory we created in the constructor, we create a new IOutputChannel specifying an EndpointAddress. The EndpointAddress requires an output channel (or the listening channel of the destination) and the destination email address of the recipient.

Once the IOutputChannel is created, we create a Message (defined in System.ServiceModel.Channels) using the static Message.CreateMessage method. Here we pass in a MessageVersion (which we set to default), a Urn, the T body, and the XmlSerializerWrapper which will handle serializing and deserializing our body.

When a message is sent, you will get a message in your mail box with a cryptic subject as follows:

```
SM:v=3.5;CN=treochannel;ID=cbb5d3c798c84af3b1daf7615b780fb3;SD=63347243604000
0000;
```

And the actual message will look something like this

```
<s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing" xmlns:s="http://www.
w3.org/2003/05/soap-envelope">
<s:Header>
<a:Action:mustUnderstand="1">urn:photoMessage</a:Action>
```

```

<a:Tos:mustUnderstand="1">net.mail://treochannel/#marteaga@opennetcf.com</a:To>
</s:Header>
<s:Body>
</s:Body>
</s:Envelope>

```

This XML is automatically created by the `System.ServiceModel.Channels.Message` class. In the case of a `PhotoData` object it will be serialized and added in between the body tags:

```

<PhotoData xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<Id>9873fd22-189f-4525-97a5-4572dbcd7ad0</Id>
<FileName>\My Documents\My Pictures\img002.jpg</FileName>
<Latitude>43.6919</Latitude>
<Longitude>-79.5782</Longitude>
<FileSize>6776</FileSize>
<Base64Data>...</Base64Data>
</PhotoData>

```

Listening for Incoming Messages

`WCFMessagingManager` exposes a `BeginListening` method which starts a separate thread that listens for incoming messages in the background. The core implementation of the listening thread is as follows:

```

//Open the listener channel
m_channelListener.Open();

//Accept and open the inputChannel
m_inputChannel = m_channelListener.AcceptChannel();
m_inputChannel.Open();

Message message;
//Calling IInputChannel.Receive will block until a message is received or
until the inputChannel is closed
while (true)
{
    message = m_inputChannel.Receive();
    if (message == null)
        break;
    try
    {
        T body = message.GetBody<T>(m_xmlSerializerWrapper);
        OnIncomingMessage(body);
    }
    catch (Exception e)
    {
        //Raise the exception handler
        OnListenException(e);
    }
}

```

The first thing we do is open the `IChannelListener` we created in the `WCFMessagingManager` constructor. Using the `IChannelListener` we create an `IInputChannel` using the `AcceptChannel()` method and then `Open()` the `IInputChannel`. Once all this is setup, we step into the `while()` loop and call

Receive() on the IInputChannel. IInputChannel.Receive is a blocking call and will not return until an incoming message is received or the IInputChannel.Close() method is called.

When a message is received, we get the Body of the message using the GetBody method (passing in our XmlSerializerWrapper) and then raise the IncomingMessage event. If there is an error receiving the message, the ListenException event is raised to notify the user that something has gone wrong.

To stop listening for incoming messages, WCFMessagingManager exposes a StopListening method which will shut down all channels and threads.

WCFMessagingManager Events

WCFMessagingManager also comes with two events so a user can be notified when a new message is received on the device or desktop application. These events are:

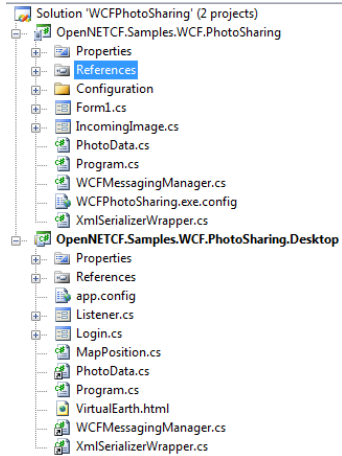
```
////// Occurs when an incoming message is received  
///</summary>  
publiceventIncomingMessageEventHandler IncomingMessage;  
  
////// Occurs when an error occurs in the listen thread  
///</summary>  
publiceventExceptionHandler ListenException;
```

These events are primarily called from the listening thread discussed above. These events are self explanatory and we will use them in the sample applications discussed below.

Sharing Code Between Platforms

A common way to share code between Windows Mobile and Desktop applications is to build an assembly targeting the .NET Compact Framework. Since .NET Compact Framework assemblies are re-targetable, they can be used in full .NET Framework applications without being recompiled. Unfortunately this is not the case with WCF because the .NET Framework implementation of Microsoft.ServiceModel.Channels.Mail.dll is vastly different to its .NET Compact Framework counterpart.

Luckily, the main source code files can be shared between the desktop and device projects by adding the files as 'Links'. The following image shows the files shared for the Photo Sharing application:



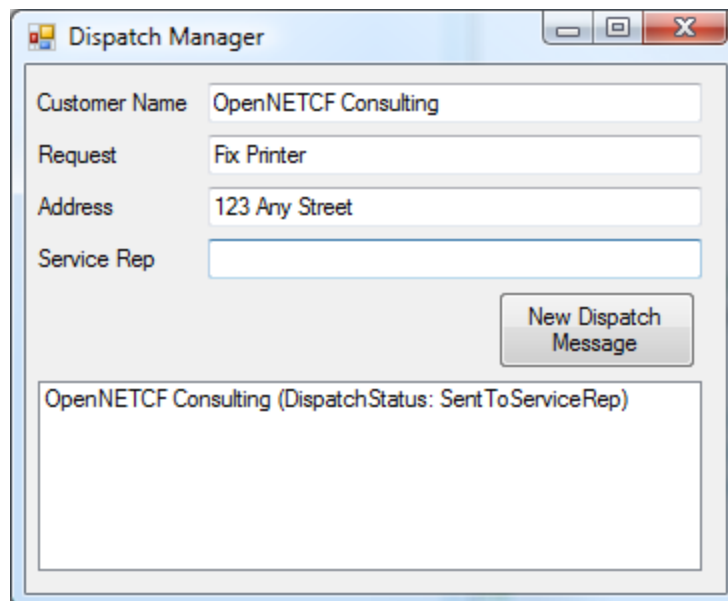
Dispatch Application

Scenario

The Dispatch Application is used where a central dispatcher sends out new requests to field service workers. The solution consists of a desktop application sending messages to and receiving messages from a Windows Mobile application using Exchange Server 2007.

Desktop Implementation

The desktop application is the central dispatch controller and sends new requests to the devices in the field. The user interface is a very simple implementation:



To send a message to and from the device we define a class that will represent the data we are sending back and forth. We define the business object using the DispatchMessage class as follows:

```

///<summary>
/// Defines a dispatch message sent to field service workers
///</summary>
publicclassDispatchMessage
{
///<summary>
/// A unique identifier
///</summary>
publicGuid Id { get; set; }

///<summary>
/// The current Status of the message
///</summary>
publicDispatchStatus DispatchStatus { get; set; }

///<summary>
/// The customer name for the request
///</summary>
publicstring CustomerName { get; set; }

///<summary>
/// A description of the request
///</summary>
publicstring Request { get; set; }

///<summary>
/// The destination address of the request
///</summary>
publicstring Address { get; set; }
}

```

The DispatchMessage object will be added in between the **Body** tags of the WCF message when sent and the XmlSerializerWrapper will handle serializing and de-serializing the object.

When the application loads, it creates a new WCFMessagingManager and starts listening for messages. When a new message is required, the user fills in the appropriate fields and sends it off to the destination by clicking on the “New Dispatch Message” button. The code then dispatches the message using the following code:

```

//Create the dispatch message
DispatchMessage dm = newDispatchMessage();
dm.Address = txtAddress.Text;
dm.CustomerName = txtCustomerName.Text;
dm.DispatchStatus = DispatchStatus.SentToServiceRep;
dm.Id = Guid.NewGuid();
dm.Request = txtRequest.Text;

//Add the dispatch message to the list
m_messagesSent.Add(dm);
UpdateListbox();

//Send the message
m_messagingManager.SendMessage(txtServiceRep.Text,
ChannelNames.ServerChannelName, dm);

```

When the IncomingMessage event is raised, we update the listbox on the main form with an update status from the device using the following:

```
void m_messagingManager_IncomingMessage(DispatchMessage body)
{
    this.Invoke(newEventHandler(delegate(object sender, EventArgs ea)
    {
        //find the message in the internal list
        var message = from tmsg in m_messagesSent
        where tmsg.Id.Equals(body.Id)
        select tmsg;

        if (message.Count() == 1)
        {
            //Update the dispatch status
            message.First().DispatchStatus = body.DispatchStatus;
        }
        else
        {
            //add the item to the list
            m_messagesSent.Add(body);
        }

        UpdateListbox();
    }));
}
```

Windows Mobile Implementation

The Windows Mobile application listens for new customer dispatch requests while out in the field. The UI implementation again is very simple and allows for the DispatchMessage object details to be displayed.

Dispatch Client

Customer Name
OpenNETCF Consulting

Request
Fix Printer

Address
123 Any Street

Id
75c73195-d548-45bb-afcd-a4f6f203d280

Status
SentToServiceRep

Update Status

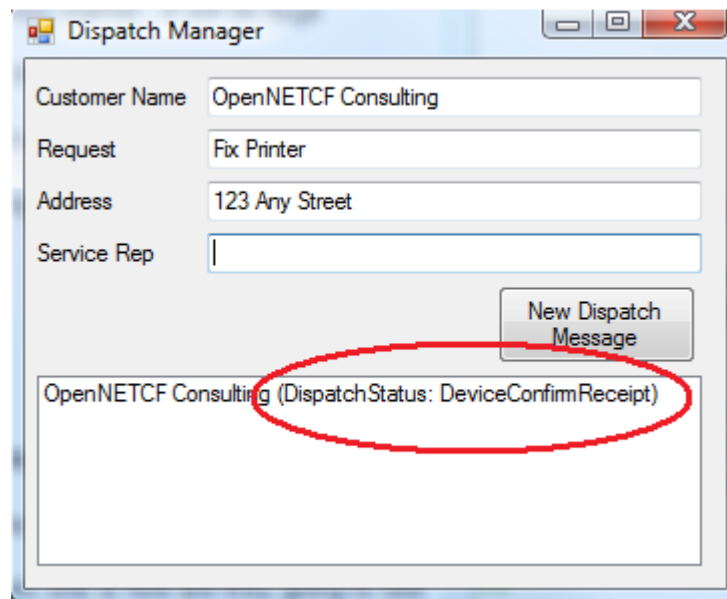
Update Message sent for OpenNETCF Cons

When a message first arrives on the device, it will automatically respond with a 'DeviceConfirmReceipt' status:

```
void m_messagingManager_IncomingMessage(DispatchMessage body)
{
    this.Invoke(newEventHandler(delegate(object sender, EventArgs ea)
    {
        this.statusBar1.Text = string.Format("New Message Received for
        {0}",body.CustomerName);
        Application.DoEvents();
        this.txtAddress.Text = body.Address;
        this.txtCustomerName.Text = body.CustomerName;
        this.txtRequest.Text = body.Request;
        this.cmbStatus.Text = body.DispatchStatus.ToString();
        this.txtId.Text = body.Id.ToString();

        if (body.DispatchStatus == DispatchStatus.SentToServiceRep)
        {
            //Respond with a confirm receipt
            CreateAndSendMessage(txtAddress.Text, txtCustomerName.Text,
            DispatchStatus.DeviceConfirmReceipt, txtRequest.Text, txtId.Text);
        }
    }));
}
```

This will be displayed on the desktop application:



The field service rep will also have the option to change the status to the following:



This way the central location will always know what the current status of the job is.

Enhancements

The sample dispatch application could be enhanced to better fit into an enterprise scenario. For example, the addition of SQL Server to store the dispatch requests, a list of field service reps currently in the field and integration with GPS data. You can also use SQL Replication to actually store the data and have it synchronize the data, and WCF can be used notify the application to initiate a sync.

Picture Sharing Application

Scenario

The Picture Sharing application allows users to share recently taken photos with their friends by leveraging WCF and Exchange to send the photos. The sample consists of a Windows Mobile application running on two separate devices and a desktop application.

In this sample we are assuming that the application will be used between friends that want to share recently taken pictures, but the same concept can be used in a LOB type application. For example, a field worker wanting to get some advice on a component while out in the field could use an application of similar architecture. The worker can snap a picture, send it back to the central dispatch location and the central dispatch location will automatically call back to provide assistance.

Windows Mobile Implementation

In the previous example we had the desktop application initiating the messages to send to the devices. In this scenario, the devices will be initiating and will be sending the messages to a desktop application as well as a secondary device.

The Windows Mobile application consists of two forms and uses the `WCFMessagingManager` and `XmlSerializerWrapper` classes.

The use of `WCFMessagingManager` and `XmlSerializerWrapper` is similar to the `DispatchSample` except in this sample we are sending a `PhotoData` object instead of `DispatchMessage` object.

We create our `WCFMessagingManager` as follows:

```
WindowsMobileMailBinding binding = newWindowsMobileMailBinding();  
m_messagingManager = newWCFMessagingManager<PhotoData>(binding,  
m_config.IncomingChannel);
```

The `PhotoData` class is defined as follows:

```

publicclassPhotoData
{
    privateBitmap m_bitmap = null;

    public PhotoData(string filename)
    {
        //Hardcoded to the toronto Congress Center
        this.Latitude = 43.6919;
        this.Longitude = -79.5782;

        //Set the file name
        FileName = filename;
        SetProperties();

        //Set the id
        Id = Guid.NewGuid();
    }

    public PhotoData()
    {
    }

    publicstring Comment { get; set; }

    publicGuid Id { get; set; }

    publicstring FileName { get; set; }

    publicdouble Latitude { get; set; }

    publicdouble Longitude { get; set; }

    publicint FileSize { get; set; }

    publicstring Base64Data { get; set; }

    ///<returns></returns>
    publicBitmap GetBitmapImage()
    { ... }

    privatevoid SetProperties()
    { ... }
}

```

Here you can see that we have various properties such as Comment, FileName, Latitude/Longitude of where the picture was taken, FileSize and Base64Data.

When sending the message over the wire using WCF, we want to make sure to convert the actual image to Base64 string. We accomplish this with the following helper method within PhotoData class:

```

privatevoid SetProperties()
{
    if (File.Exists(FileName))

```

```

    {
//Grab the byte data from the file
FileStream fs = File.Open(FileName, FileMode.Open);
byte[] data = newbyte[fs.Length];
    fs.Read(data, 0, data.Length);
    fs.Close();

//Set the file size
    FileSize = data.Length;

//convert the byte[] to base64
    Base64Data = Convert.ToBase64String(data);
    }
}

```

This method is called from the PhotoData(string) constructor and the constructor automatically sets the appropriate properties within the class.

You may have noticed that PhotoData has two constructors, the first taking a string parameter and the second taking no parameters. The reason for this is that XmlSerializer (used in XmlSerializerWrapper) requires a parameter-less constructor to de-serialize the XML data. So when an incoming PhotoData object is received, XmlSerializer will automatically de-serialize the data.

There is also a public method in the class which returns a Bitmap object to display on the destination.

```

publicBitmap GetBitmapImage()
{
if (m_bitmap == null)
    {
if (Base64Data != null)
    {
byte[] imageData = Convert.FromBase64String(Base64Data);
        m_bitmap = newBitmap(newMemoryStream(imageData));
    }
}
return m_bitmap;
}

```

So on an incoming PhotoData object, displaying the image is as simple as calling GetBitmapImage.

The UI of the Windows Mobile application is straight forward. Main form, is a very simple form using a PictureBox and two menu items to take a new picture and to send the pictures.



To take the pictures we use the Microsoft.WindowsMobile.Forms.CameraCaptureDialog which is available with Windows Mobile 5 and later devices as follows:

```
using (CameraCaptureDialog ccd = new CameraCaptureDialog())
{
    if (ccd.ShowDialog() == DialogResult.OK)
    {
        CurrentPhotoData = new PhotoData(ccd.FileName);
        CurrentPhotoData.FileName = ccd.FileName;
        pictureBox1.Image = new Bitmap(CurrentPhotoData.FileName);
    }
}
```

When the picture is taken, we set the PictureBox to display the currently displayed image.

Sending a PhotoData object is also straight forward and we simply need to call WCFMessagingManager.SendMessage as follows:

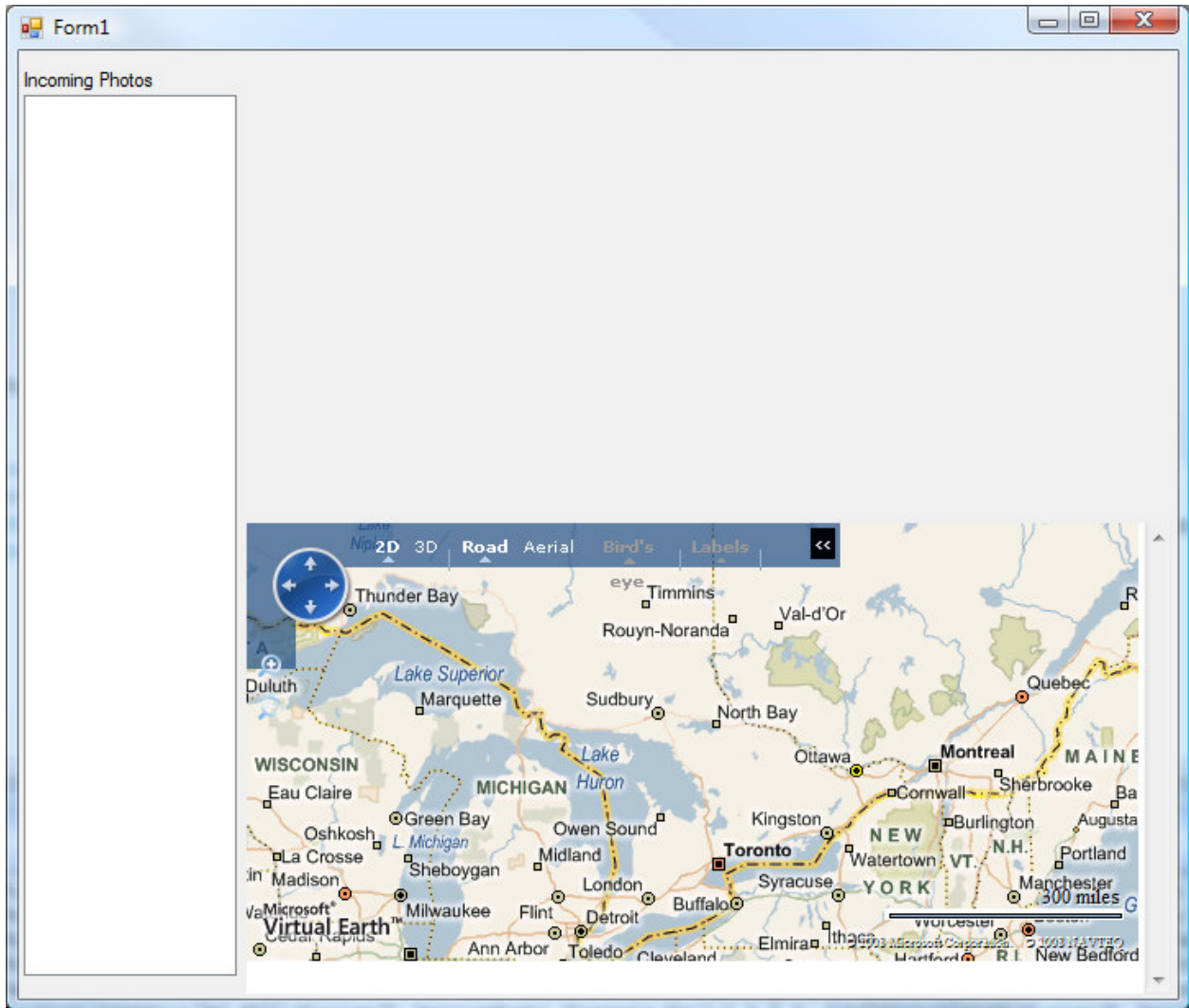
```
//Send the message to the device
m_messagingManager.SendMessage(m_config.SendToEmail,
m_config.OutgoingChannel, CurrentPhotoData);
//Send the message to the desktop
m_messagingManager.SendMessage(m_config.SendToEmail,
m_config.OutgoingChannelDesktop, CurrentPhotoData);
```

WCFMessagingManager will handle sending the PhotoData objects to the appropriate email address and the corresponding channels those emails should be listening to.

Desktop Implementation

The desktop implementation primarily just listens for incoming PhotoData objects and displays them. The user interface consists of a ListBox to display the current PhotoData objects in memory, a PictureBox to display the image and a WebBrowser control to display a Virtual Earth map and a push pin of where the picture was taken.

The following is a screen shot of the user interface:



When the `WCFMessagingManager.IncomingMessage` event is handled by the main UI we want to update our UI appropriately as follows:

```
this.BeginInvoke(new EventHandler(delegate(object sender, EventArgs ea)
{
//find the message in the internal list
var message = from tmsg in m_messagesReceived
where tmsg.Id.Equals(body.Id)
select tmsg;

if (message.Count() == 0)
{
//add the item to the list
m_messagesReceived.Add(body);
}

UpdateListbox();
}));
```

The `m_messagesReceived` variable is of type `List<string>` and is used as the `DataSource` for the `ListBox` as follows:

```
listBox1.DataSource = m_messagesReceived;
```

On the `ListBox.SelectedIndexChanged` event, we display the picture and a push pin within the Virtual Earth map to give a visual location of where the image was taken. We accomplish this using the following:

```
if (listBox1.SelectedIndex >= 0 && listBox1.SelectedIndex <
m_messagesReceived.Count)
{
PhotoData pd = m_messagesReceived[listBox1.SelectedIndex];
//Grab the photo data object and display the image
this.pictureBox1.Image = pd.GetBitmapImage();

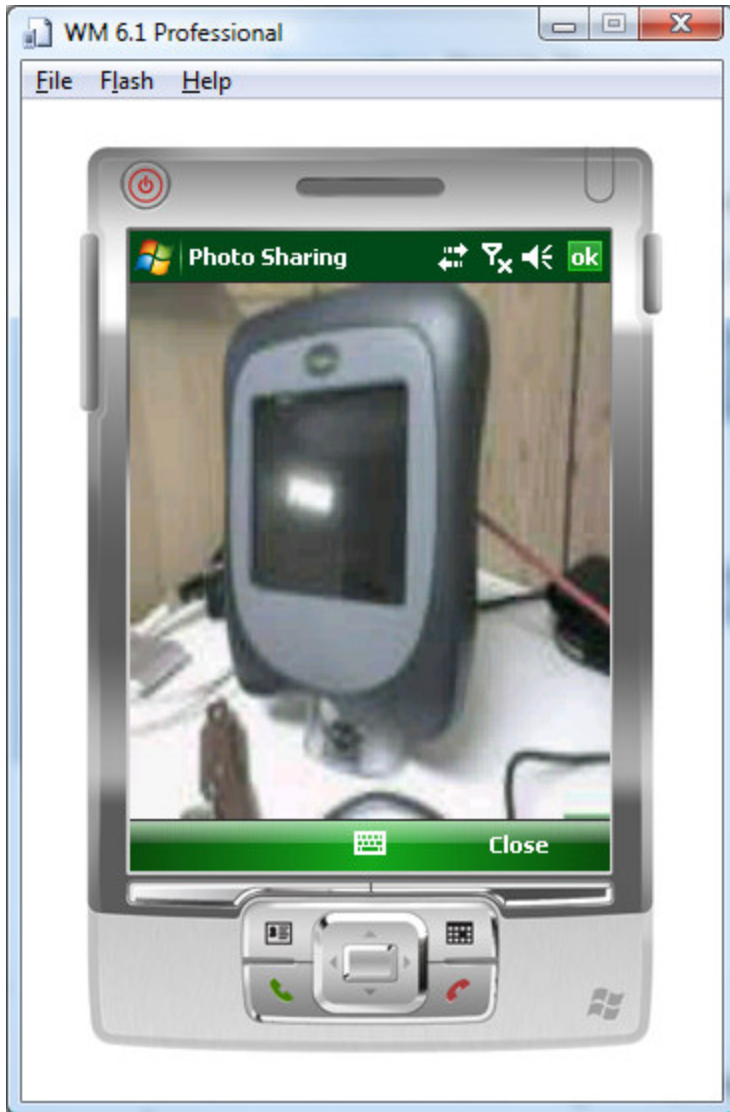
//Show a push pin on the map of where the image was taken
executeScript(ClearPushpinsScript);

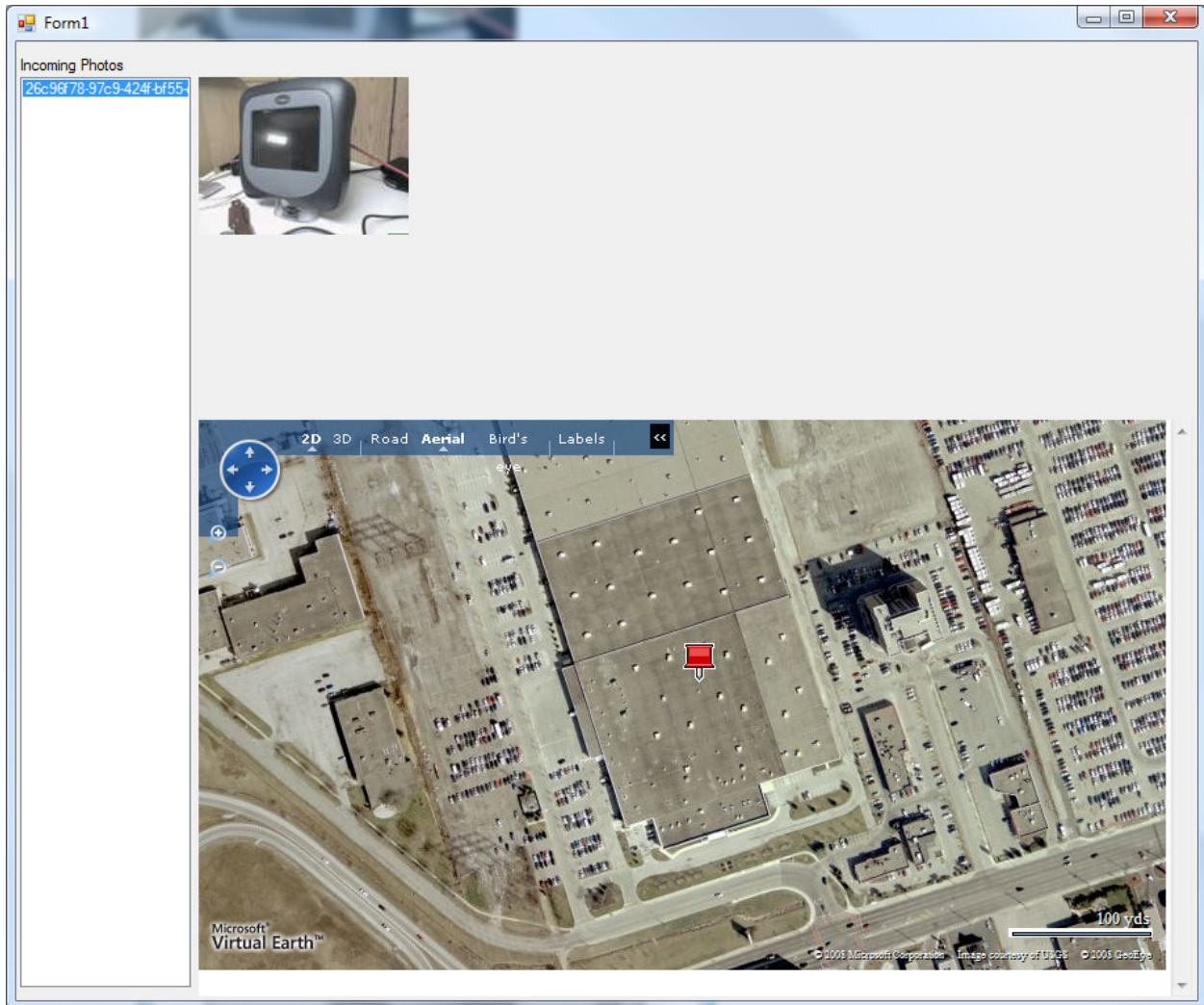
executeScript(AddPushpinScript, pd.Id.ToString(),
(pd.Comment == null ? "No Comments" : pd.Comment.ToString()),
pd.Latitude, pd.Longitude);
//center on the push pin
executeScript(SetCenter, pd.Latitude, pd.Longitude);
//zoom in on the push pin
executeScript(Zoom, 18);
}
```

Final Results

The following are screen shots of a test run. Here I took a picture using a Palm Treo 700wx running Windows Mobile 5.0, sent it to the desktop application as well as a Windows Mobile 6.1 Professional Emulator.







Virtual Earth Integration on the Desktop

Integration with Virtual Earth came up as the PhotoData object contained Latitude and Longitude values. To integrate Virtual Earth with the WinForms application, [WinForms Earth v2](#) from Via Virtual Earth was leveraged. A few updates had to be done to get things to work as the original code was extremely out date.

To get a Virtual Earth map to load within a WinForms application and to communicate with the Virtual Earth map through code, there are a few things that need to be done.

1. An local HTMLpage is required that will load up the Virtual Earth Map. This HTMLfile will also contain JavaScript functions that will be called from the .NET application
2. `ComVisible(true)` Attribute must be added to your form. This can also be added to a class that will handle the communication but between the .NET code and the HTMLdocument but for simplicity we added it to the Form.

3. Interaction from .NET to the HTMLdocument JavaScript code is done by calling `WebBrowser.Document.InvokeScript` method which accepts the javascript method name and the parameters for the method
4. Interaction from HTMLdocument to .NET is done by using the JavaScript function `window.external.X` where X represents your method name in your .NET Code. This method must be public. See the original source for WinForms Earth V2 as this has the sample there.

Enhancements

Some enhancements that can be done to the application would be integrating GPS functionality with the Windows Mobile application (see the [GPS Sample](#) application available with Windows Mobile SDK). On the desktop side, you can have the PhotoData objects stored in a SQLServer Compact Database and also integrate any incoming data with Flickr or Facebook Photos.

Conclusion

E-mail has been around for a long time, and e-mail was one of the first applications implemented on mobile devices. With Windows Mobile, now that we have direct push email and .NET Compact Framework 3.5 we can leverage email and the direct push functionality in our Windows Mobile applications. As developers it allows us to focus on our application and not worry about how we are going to get data from point A to point B,C,E etc. If using Exchange is not possible with your scenario, you can always extend the WCF functionality with something like XMPP, but that's an entire article on its own!