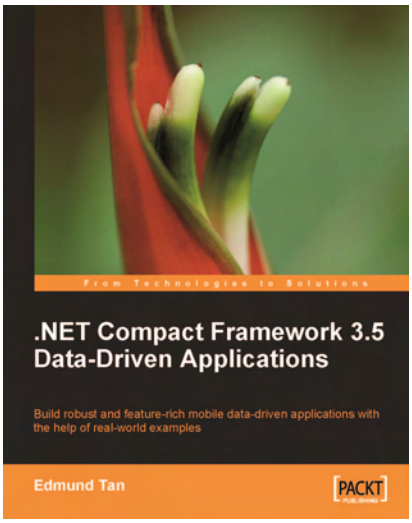
# .NET Compact Framework 3.5 Data-Driven Applications

**Edmund Tan**




## Chapter No.5

## "Building Integrated Services"

## In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.5 "Building Integrated Services"

A synopsis of the book's content

Information on where to buy this book

# About the Author

**Edmund Tan** is the CTO and co-founder of a leading e-forms and workflow solution vendor based in Singapore. He holds more than eight years of experience building performance-critical .NET e-forms and workflow solutions for smart devices on top of Oracle and Microsoft SQL Server databases for large companies and governmental institutions located in Singapore and Malaysia. Edmund is also a regular public speaker at various conferences held in Singapore and Malaysia on the topic of on-the-go BPM (Business Process Management) hosted on smart device technologies. During his free time, Edmund works on trying to create machines capable of emulating human thought.

My first and heartfelt word of thanks goes out to David Barnes without whom this book would not have been possible at all. I also thank James Lumsden for giving my e-mail a chance, Rakesh Shejwal and Greg, my long-time collaborator, for their always insightful edits, Srimoyee Ghoshal for keeping me on schedule, Lata Basantani, as well as everyone else involved in this book at Packt.

I extend my gratitude to my parents, Laumee and Obagi, for their undying love and support, my aunt Laumar for instilling the bookworm in me, my wife Shen for keeping the beverages fl owing, my kid Sophie simply for being the pride of my life, and, last but not least, my brother Edwin for his jokes during the really, really late hours of the night.

# .NET Compact Framework 3.5 Data-Driven Applications

As business systems become increasingly distributed, the mobile device becomes an increasingly important tool on the enterprise stage. The large amount of processing power available to mobile devices nowadays bring to it a whole new range of possibilities as a mobile extension to traditional server-based enterprise systems.

Harnessing this power is the .NET Compact Framework, which has seen tremendous improvements over the last few versions. The .NET Compact Framework provides a rich set of managed classes that does away with a big chunk of the menial labor required to perform common tasks, leaving the developer to focus on building business logic instead.

This book is not intended to be a complete reference tome of the .NET Compact Framework. There are numerous books and documentation online that serve this purpose. Rather, it will show you how to apply the .NET Compact Framework in interesting ways to solve real-world business problems. We will explore commonly encountered design decisions and technology comparisons along the way and ultimately build clean solutions that keep to best practices such as the three-tier design and the Model View Controller (MVC) model.

Using a sales force application as the central example and theme in this book, you will have a clear step-by-step guide on building one of the most popular types of business applications in the market today from ground up. Through these pages, you will learn how to create robust data-driven mobile applications that work seamlessly with other mobile devices and database servers. You will get to explore the little nuances of .NET Compact Framework programming, and how to get around them using its advanced features. You will also get a firsthand look at how you can use third-party libraries such as the open source Smart Device Framework to add a host of rich functionality to your applications. Towards the end of this book, you will have accumulated enough understanding of the capabilities and limitations of the .NET Compact Framework and its tools to confidently tackle an enterprise mobile application of any size or complexity.

I hope in the process of getting there you will have as much fun reading this book and trying out the samples as I had writing it.

## What This Book Covers

*Chapter 1, CRMLive.NET: An Overview,* provides a technical and scope overview of CRMLive.NET, a mobile customer relationship management suite comprising three individual applications (a mobile sales force, mobile dashboard, and mobile support case application). Chapter 1 also outlines the four different mobile client models and a comparison of their strengths and weaknesses.

*Chapter 2, Building the Data Tier,* shows how a plugin-based data tier based on both the Microsoft SQL Server Compact and Oracle Lite databases can be created using ADO.NET.

*Chapter 3, Building the Mobile Sales Force Module,* walks the reader through building the logic and presentation tiers of the mobile sales force application, illustrating various concepts along the way such as UI object reusability, validation, paging, record navigation, sorting, and grouping.

*Chapter 4, Building Search Functionality*, illustrates how full-text search and parameterized-search functionality can be added to the mobile sales force application.

*Chapter 5, Building Integrated Services,* illustrates how the sales force application can make use of the .NET Compact Framework and P/Invoke calls to access underlying Windows Mobile operating system and mobile device functionality such as the Bluetooth, Infrared, Calendar, and Telephony services.

*Chapter 6, Data Synchronization,* covers one of the most important topics in the book—the process of data synchronization between the mobile device and the remote database. In this chapter, we will look at how the sales force application can perform bidirectional synchronization using Microsoft SQL Server Compact's **SQL RDA** and Oracle Lite's **mSync** technologies.

*Chapter 7, Optimizing for Performance,* illustrates how the sales force application's performance can be measured and improved using various techniques such as data caching and data compression.

*Chapter 8, Securing the Application,* covers the various ways to secure locally stored data on the mobile device. It also covers the various authentication mechanisms available during data synchronization with the remote database.

*Chapter 9, Globalization,* illustrates how the reader can globalize the sales force application to intrinsically support double-byte (Unicode) languages.

*Chapter 10, Building the Dashboard,* walks the reader through the building of the second application in CRMLive.NET—the mobile dashboard. It will cover the use of stateless asynchronous web service calls to retrieve XML-based data from a remote server.

*Chapter 11, Building the Support Case System,* walks the reader through the building of the third application in CRMLive.NET—the mobile support case application. It will cover how a messaging backbone based on **Microsoft Messaging Queue (MSMQ)** technology can be built to support disconnected-state messaging between two remote applications.

*Chapter 12, Testing and Debugging,* looks at how the tools provided in the *PowerToys for .NET CF 3.5* suite can assist in the testing and debugging process of the CRMLive.NET application.

*Chapter 13, Packaging and Deployment,* walks through the packaging and deployment process of the CRMLive.NET application and how a network-aware, automated update service can be created to assist in the deployment of application upgrades.

# 5

# Building Integrated Services

I still remember those days when mobile phones were just mobile phones. Nowadays, they are loaded with tons of hardware features that extend their use as full-blown life organizers and enterprise tools. As mobile device manufacturers find ingenious ways to squeeze in ever more hardware functionality, the .NET Compact Framework and other third-party vendors have been busy trying to write the software to keep up. Through Windows Mobile 6, a whole range of functionality has been added to let you, the developer, tap into the various hardware features of the mobile device.

Third-party libraries like the Smart Device Framework have also been active in this area, releasing classes and controls that make .NET Compact Framework development easier than ever. In this chapter, you will learn how to add the following functionalities to the sales force application:

- Placing phone calls and sending SMS messages/e-mails to a lead directly
- Detecting and logging all incoming and outgoing communication (SMS messages and phone calls)
- Using the Bluetooth and Infrared capabilities of your mobile device to transfer a lead account between devices
- Capturing a customer's signature

## Sending SMS and e-mail from your application

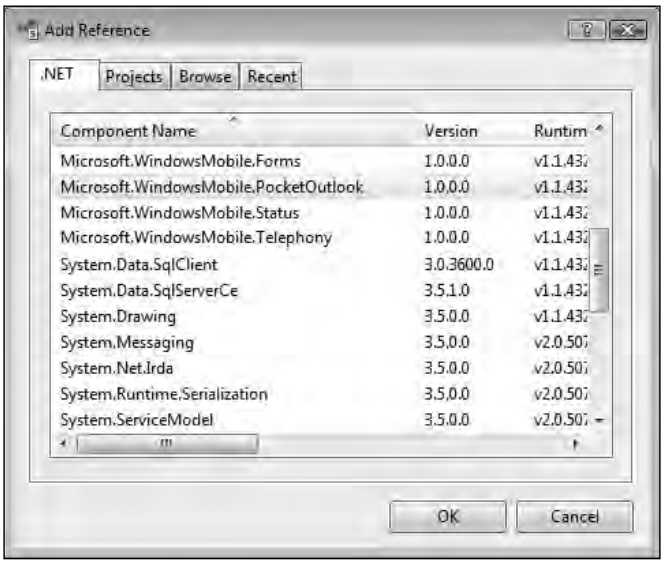
As you are probably aware, Windows Mobile uses the integrated Outlook Mobile software to manage your tasks, appointments, e-mails, SMS, and MMS messages. Microsoft provides the `Microsoft.WindowsMobile.PocketOutlook` namespace, which allows you to utilize Outlook Mobile to programmatically send an SMS or e-mail.

The **POOM** (**Pocket Outlook Object Model**) allows you to automate the Windows Mobile messaging application's UI. This means that you can use POOM to send SMS and e-mail messages, create tasks, iterate through your appointments, and so on.

There are two ways to include SMS- and e-mail-sending capability in your application. One way is to build your own UI to let the user key in his or her message and use POOM to programmatically send the message. Alternatively, you could also delegate this functionality to the default Windows Mobile Compose window (and, to make it more convenient for the end user, prefill some of the fields in this window with the minimum data required—such as the intended recipient of the message). Let's take a look at both approaches in the following sections:

# Sending SMS and e-mail directly through code

Sending an SMS through POOM is straightforward. Before you can use POOM, however, you must first import a reference to the `Microsoft.WindowsMobile.PocketOutlook` library.




All the code that you need to send an SMS is shown as follows:

```
using Microsoft.WindowsMobile.PocketOutlook;

public void SendSMS(string phoneNumber, string message)
{
```

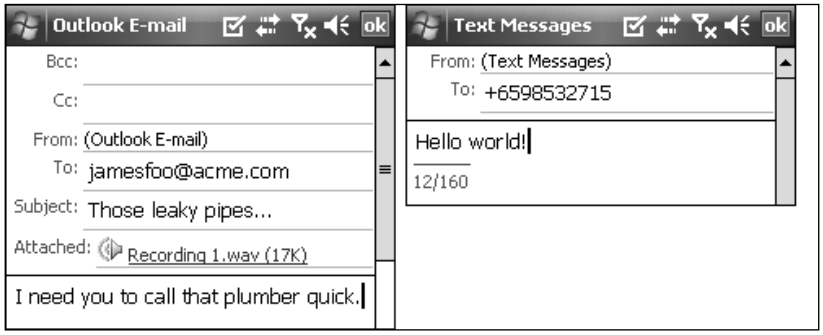
```
    SmsMessage _msg;
    _msg = new SmsMessage(phoneNumber, message);
    _msg.Send();
}

public static void main()
{
    SendSMS("+6598532715","Hello world!");
}
```

You can send an e-mail through POOM using the `EmailMessage` class. The code to do this is similarly straightforward.

```
public void SendEmail()
{
    EmailMessage _em = new EmailMessage();

    //Define recipients
    _em.To.Add(New Recipient("John Mayer",
       "johnmayer@acme.com"));
    _em.To.Add(New Recipient("Ed Stables",
       "edstables@acme.com"));

    //Define CC or BCC list, if any
    _em.CC.Add(New Recipient("Greg Yap", "gregyap@acme.com"));

    //Define attachments - you can attach multiple files
      _em.Attachments.Add(New
       Attachment("C:\documents\contract.docx"));
    _em.Attachments.Add(New
       Attachment("C:\documents\contract2.docx"));

    //Define other e-mail attributes
    _em.Importance = Importance.High;
    _em.Sensitivity = Sensitivity.Confidential;

    //Define the main e-mail subject and body
    _em.Subject = "The contract documents you asked for";
    _em.BodyText = "Hey guys, as requested...";

    //Send out the e-mail using an e-mail account on Mobile Outlook
    _em.Send("AcmeSMTPAccount");
}
```

# Delegating to the default Windows Mobile Compose UI

Outlook Mobile by default provides Compose windows that let you compose an SMS or e-mail message in Windows Mobile (accessible under the **Messaging** menu). These windows can be seen as follows:




Your application could use POOM to launch these windows (prefilled with some minimal data). For example, if you wanted to provide functionality to send an SMS using a phone number available in your application, you could launch the Compose Text Message window with the **To** phone number field filled in and the content of the SMS defaulted to some value. Let's see how this could be done in code:

```
SMSMessage _msg = new SMSMessage();
_msg.To.Add(New Recipient("Ed", "+6598532715"));
_msg.Body = "This message was sent from CRMLive!";
MessagingApplication.DisplayComposeForm(_msg);
```

You could do the same thing with e-mails by creating an `EmailMessage` object and feeding it to the same `MessagingApplication.DisplayComposeForm()` function.

# Intercepting incoming SMS

In *Chapter 3, Building the Mobile Sales Force Module,* you've created a **History** tab in the Account Details window that displays logs of all communication with a lead, opportunity, or customer. These logs include incoming phone calls and SMS messages. In this section, you will write the code that *generates* these logs.

To perform this, you would need to be able to detect SMS messages as they come in. Your application would then need to do the following:

1. Look at the origin (phone number) of these messages
2. Locate the lead account with the matching phone number in the sales force application
3. The log entry is then created under this account (if it is found)

# Intercepting an SMS message

Intercepting an SMS message requires a bit more work compared to sending one. The `Microsoft.WindowsMobile.PocketOutlook.MessageInterception` namespace provided by Microsoft allows you to define a call back function that is automatically called when a new SMS message arrives.

Let's take a look at how this is done in code. The first thing to do is to import the namespace we need. This is shown as follows:

```
using Microsoft.WindowsMobile.PocketOutlook.MessageInterception;
```

Next, you need to create a `MessageInterceptor` object. You can specify one of these two values for the `InterceptionAction` argument in the constructor of this object:

- **InterceptionAction.Notify**

  This allows your application to 'peek' at the incoming SMS message without deleting it—this message is still available to other applications.
- **InterceptionAction.NotifyAndDelete**

  If you choose this option, the SMS message will not be available to other applications after your application has received it.

The second argument defines whether to use the form thread to process the events. If your application does not use any form (for example, in the case of a console application), set this value to `false`.

```
MessageInterceptor _SmsInterceptor;
_SmsInterceptor = new MessageInterceptor(InterceptionAction.Notify,
True);

//You must then add an event handler for the MessageReceived() event
//in the interceptor object.
_SmsInterceptor.MessageReceived+= SmsReceivedEventHandler;
```

Through the event arguments of the event handler, you can retrieve an `SmsMessage` object, which you can then use to retrieve the originating phone number (or SMS content).

```
public void SmsReceivedEventHandler(ByVal sender As Object,
   ByVal e As MessageInterceptorEventArgs)
{
   SmsMessage _msg = (SmsMessage) e.Message;
   MessageBox.Show("Received SMS from the following number: "
      + _msg.From.Address.ToString(),"");
}
```

The `MessageInterceptor` object seen previously is typically created when your application starts, and you will only receive SMS receipt notifications when your application remains open. The moment you close your application, the object goes out of scope and you stop receiving SMS receipt notifications as well.

If you are planning to process each and every single SMS message that comes into your phone over extended periods of time, it would make more sense to create a window-less application that runs constantly in the background, listening to the events raised by the `MessageInterceptor` class. You will see how you can do exactly this in the next few sections.

Unfortunately, the `MessageInterceptor` object cannot be used to intercept an e-mail message. To intercept an e-mail, you would need to invoke **MAPI** (**Messaging API**) functionality on the device, which is out of the scope of this book.

# Placing phone calls from your application

In most mobile development projects, convenience is a central theme. User friendliness is measured by how quick a user can get to the information that he or she needs, or how quickly he or she can execute a desired action.

With this in mind, let's consider the following example. John might have created an application that manages a list of customers and their phone numbers. When viewing the details of a particular customer, the end user might wish to make a phone call to the customer. It wouldn't make much sense to have the user key in the phone number of the customer again as it is already onscreen. We could make it more convenient for the end user by placing a button called *Make a phone call* that automatically dials the onscreen number.

The `Telephony` library provided in the `Microsoft.WindowsMobile` namespace allows you to easily place a phone call directly from your application. Let's see how this can be done:

```
Using Microsoft.WindowsMobile.Telephony;

public object MakePhoneCall(string phoneNumber)
{
    Phone _phone = new Phone();
    _phone.Talk(phoneNumber);
}
```

# Detecting incoming phone calls

You saw earlier how you could detect incoming SMS messages using the `MessageInterception` class. How about incoming phone calls? You can do the same thing for phone calls but you'll need to use a different class—the `SystemState` class.

The `SystemState` class is a useful class that allows you to query almost every type of state information on the mobile device. It allows you to query information such as battery life, missed phone calls, ActiveSync status, Bluetooth status, calendar events, appointment events, notification events, camera status, and phone signal strength, just to name a few. Not only can you query these statuses but you can also receive events in your code when these values change.

Let's take a look at the code you can use to do this. The method used to detect phone calls is roughly the same as the one used to detect incoming SMS messages. You need to set it to listen in the background for a specific status field and to register a call back function that can be automatically invoked when the status value changes. In the following code, we set it to listen to a status change in the incoming phone number:

```
using Microsoft.WindowsMobile.Status;

public void StartDetector()
{
   _phoneCaller = new
      SystemState(SystemProperty.PhoneIncomingCallerNumber,
      true);
   _phoneCaller.Changed += phoneCaller_Changed;
}
```

In the event handler, simply print the incoming phone number.

```
private void phoneCaller_Changed(object sender,
   ChangeEventArgs args)
{
   _IncomingphoneNumber =
      Strings.Trim(SystemState.PhoneIncomingCallerNumber);
   MessageBox.Show("The incoming call is from this number:
      " + _IncomingphoneNumber,"");
}
```

The format of the phone number retrieved depends on the numbers you have stored in your Windows Mobile Contacts area. For instance, if you have a Windows Mobile Contact named `My wife` with the phone number `+60163176148` and you receive a call from this number, the function preceding would yield:

```
My Wife <+60163176148>
```

If the number does not exist in your Windows Mobile Contacts area, it would simply return the number (without any brackets):

```
+60163176148
```

You can easily use regular expressions to parse and strip away all the unnecessary text if you want to extract only the phone number from this string.

# Populating the History tab in the sales force application

Now that you know how to use the various mobile phone and OS features, you will need to build it into your sales force application. To jog your memory a little, you've created the `HistoryList` control in the `AccountViewer` form to view a list of all incoming and outgoing communication (SMS messages/phone calls) for each account. This screen is shown as follows:

To populate this list, you will need to be able to detect SMS messages and phone calls as and when they are sent or received, and then generate the corresponding historical records in the database. You will need to do the following tasks:

- Create the relevant functions in the data tier to insert new historical records
- Encapsulate SMS and phone functionality in a class each
- Create the background application to intercept incoming SMS messages and phone calls—this will generate the corresponding historical records
- Handle outgoing SMS messages and phone calls—this will also generate the corresponding historical records

# Creating the data tier functions to insert historical records

The first thing you need to do is to create the Oracle Lite and SQL Server CE functions to insert historical records into the `AccountHistories` table. You'll need to define the following functions in the `IDataLibPlugin` interface:

```
bool InsertHistoricalRecord(Guid AccountGUID, int
   OriginatingSource, string Subject, string Description);

bool InsertHistoricalRecordByPhone(string phoneNumber, int
   OriginatingSource, string Subject, string Description);
```

The following list describes what each function can do:

- `InsertHistoricalRecord()`: This function generates a history record for a specified account.
- `InsertHistoricalRecordByPhone()`: This function allows you to pass in the incoming phone number. It locates an account with the matching phone number and then calls the `InsertHistoricalRecord()` function to generate the history record under that account.

You will also notice in the preceding functions that you need to insert an `OriginatingSource` value into the `AccountHistories` table. This is an integer value that takes on the value of `0` (incoming) or `1` (outgoing). It would be more intuitive to use an enumerated type to represent this in your code. Add the following enumerated type to the `GlobalVariables` class in the `CRMLiveFramework` project.

```
public enum Communications
{
   Incoming=0,
   Outgoing=1
}
```

Now let's take a look at the SQL Server CE implementation for these two functions:

```
public bool InsertHistoricalRecord(System.Guid AccountGUID,
   int OriginatingSource, string Subject, string Description)
{
   SqlCeCommand _command;
   bool _result;
   _command = _globalConnection.CreateCommand();
   _command.CommandText = "INSERT INTO
      AccountHistories(AccountGUID,OriginatingSource,Subject,
      Description,Timestamp) VALUES (@AccountGUID,
      @OriginatingSource, @Subject, @Description, GETDATE())";

   _command.Parameters.Add("@AccountGUID", AccountGUID);
   _command.Parameters.Add("@OriginatingSource",
      OriginatingSource);
   _command.Parameters.Add("@Subject", Subject);
   _command.Parameters.Add("@Description", Description);
   try
   {
      if (_command.ExecuteNonQuery() == 1)
      {
         _result = true;
      }
      else
      {
         _result = false;
      }
   }
   catch (Exception ex)
   {
      throw (ex);
      _result = false;
   }
   _command.Dispose();
   _command = null;
   return _result;
}

public bool InsertHistoricalRecordByPhone(string phoneNumber,
   int OriginatingSource, string Subject, string Description)
{
   SqlCeCommand _command;
   SqlCeDataReader _datareader;
```

```
    Guid _accountGUID;
    bool _result;

    //First we attempt to retrieve an account that matches the
    //incoming phone number passed in
    _command = _globalConnection.CreateCommand();
    _command.CommandText = "SELECT AccountGUID FROM Accounts
       WHERE MobPhoneNo LIKE @PhoneNo OR ResPhoneNo LIKE
       @PhoneNo";
    _command.Parameters.Add("PhoneNo", phoneNumber);
    try
    {
       _datareader = _command.ExecuteReader();

       //If an account is found, we create the historical record for it
       if (_datareader.Read() == true)
       {
          _accountGUID = _datareader.GetGuid
             (_datareader.GetOrdinal("AccountGUID"));
          InsertHistoricalRecord(_accountGUID,
             OriginatingSource, Subject, Description);
       }
       _result = true;
    }
    catch (Exception ex)
    {
       _result = false;
       throw (ex);
    }
    _command.Dispose();
    _command = null;
    return _result;
}
```

# Encapsulating SMS functionality

You could of course access the POOM classes directly from your application, but it is a good idea to encapsulate its functionality in a custom class of your own. The `MessagingService` class handles both outgoing and incoming SMS messages. Add this class to the `CRMLiveFramework` project. Let's take a look at the following class:

```
using System;
using System.Data;
using Microsoft.WindowsMobile.PocketOutlook;
using Microsoft.WindowsMobile.PocketOutlook.
        MessageInterception;
using System.Text.RegularExpressions;

namespace CRMLive
{
   public class MessagingService
   {
      private SmsMessage _msg;
      private MessageInterceptor _SMSInterceptor;
      private string _IncomingSource;
      public delegate void
         IncomingSMSReceivedEventHandler(string
         IncomingPhoneNumber);
      private IncomingSMSReceivedEventHandler
         IncomingSMSReceivedEvent;

      public event IncomingSMSReceivedEventHandler
         IncomingSMSReceived
      {
         add
         {
            IncomingSMSReceivedEvent =
               (IncomingSMSReceivedEventHandler)
               System.Delegate.Combine
               (IncomingSMSReceivedEvent, value);
         }
         remove
         {
            IncomingSMSReceivedEvent =
               (IncomingSMSReceivedEventHandler)
               System.Delegate.Remove
               (IncomingSMSReceivedEvent, value);
         }
      }
```

```
public string IncomingSource
{
   get
   {
      return _IncomingSource;
   }
}

public void CreateSMS(string phoneNumber, string
   message)
{
   _msg = new SmsMessage(phoneNumber, message);
}

public void StartDetector()
{
   _SMSInterceptor = new
      MessageInterceptor(InterceptionAction.Notify,
      true);
   _SMSInterceptor.MessageReceived += new
      Microsoft.WindowsMobile.PocketOutlook.
      MessageInterception.MessageInterceptorEventHandler
     (SMSReceivedEventHandler);
}
```

You've seen earlier that the incoming phone number retrieved might look something like this:

```
Ed <+6598532715>
```

You can use a simple Regular Expression to extract just the phone number (`+6598532715`) from between the < > brackets. Let's take a look at this function in detail:

```
private void SMSReceivedEventHandler(object sender,
   MessageInterceptorEventArgs e)
{
   _msg = (SmsMessage)e.Message;
   _IncomingSource =
      _msg.From.Address.ToString().Trim();
   if (_IncomingSource.Length > 0)
   {
      if (Regex.IsMatch(_IncomingSource, "\\<(.*)\\>")
         == true)
      {
```

```
                _IncomingSource = Regex.Match(_IncomingSource,
                "\\<(.*)\\>").Groups[1].Value;
            }
            if (IncomingSMSReceivedEvent != null)
               IncomingSMSReceivedEvent(_IncomingSource);
         }
      }

      public void SendSMS()
      {
         _msg.Send();
      }

      public void LaunchSMSComposeWindow()
      {
         MessagingApplication.DisplayComposeForm(_msg);
      }
   }
}
```

# Encapsulating phone functionality

The `Telephony` class is the equivalent class for phone functionality in the sales force application. Add the following class to the `CRMLiveFramework` project:

```
using System;
using System.Data;
using Microsoft.WindowsMobile.Telephony;
using Microsoft.WindowsMobile.Status;
using System.Text.RegularExpressions;

namespace CRMLive
{
   public class Telephony
   {
      private SystemState _IncomingCallState;
      private SystemState _phoneCaller;
      private string _IncomingphoneNumber;
      public delegate void IncomingCallReceivedEventHandler
         (string incomingPhoneNumber);
      private IncomingCallReceivedEventHandler
         IncomingCallReceivedEvent;
      public event IncomingCallReceivedEventHandler
         IncomingCallReceived
```

```
{
   add
   {
      IncomingCallReceivedEvent =
         (IncomingCallReceivedEventHandler)
         System.Delegate.Combine
         (IncomingCallReceivedEvent, value);
   }
   remove
   {
      IncomingCallReceivedEvent =
         (IncomingCallReceivedEventHandler)
         System.Delegate.Remove
         (IncomingCallReceivedEvent, value);
   }
}

public object IncomingPhoneNumber
{
   get
   {
      return _IncomingphoneNumber;
   }
}

public object MakePhoneCall(string phoneNumber)
{
   Phone _phone = new Phone();
   _phone.Talk(phoneNumber);
}

public void StartDetector()
{
   _phoneCaller = new SystemState
      (SystemProperty.PhoneIncomingCallerNumber, true);
   _phoneCaller.Changed += new
      Microsoft.WindowsMobile.Status.ChangeEventHandler
      (phoneCaller_Changed);
}

private void phoneCaller_Changed(object sender,
   ChangeEventArgs args)
{
```

```
            _IncomingphoneNumber =
               SystemState.PhoneIncomingCallerNumber.Trim();
            if (_IncomingphoneNumber.Length > 0)
            {
               if (Regex.IsMatch(_IncomingphoneNumber,
                  "\\<(.*)\\>") == true)
               {
                  _IncomingphoneNumber =
                     Regex.Match(_IncomingphoneNumber,
                     "\\<(.*)\\>").Groups[1].Value;
               }
               if (IncomingCallReceivedEvent != null)
                  IncomingCallReceivedEvent
                  (_IncomingphoneNumber);
            }
         }
      }
   }
```

# Intercepting incoming SMS messages and phone calls in the background

As hinted earlier, your sales force application would likely need to detect incoming SMS messages and phone calls even if it is not running. The best way to achieve this is to write a *separate* program that constantly runs as a background service in memory. You should also have this program automatically run when the mobile device starts up.

If you are wondering why you need to write a separate program, that's because your sales force application is a heavy application—if you take a look at the file size of the generated `SalesForce.exe` file, you can see it's somewhere around 150Kb by now. It wouldn't be a good idea to keep this whole application constantly running in the background. A separate program on the other hand, only has a 10Kb foot print.

Let's take a look at how you can create such a program. Create a new Windows Forms project (named `CRMLiveInterceptor`) and add a form called `MainForm` to your project. You will need to add a reference to the `CRMLiveFramework` project because this application will make use of the database plugins.

```
using System;
using System.Windows.Forms;
using System.Reflection;
using System.IO;
using CRMLive;

public class MainForm : Form
{
```

You will of course need to create a few objects that you will be using in this application:

```
   private PluginManager _PluginManager = new
      PluginManager();
   private MessagingService _Interceptor = new
      MessagingService();
   private Telephony _Telephony = new Telephony();

   //The constructor for the class. In this constructor we
   //setup some of the event handlers
   public MainForm()
   {
      this.Activated +=new EventHandler(Form_Activated);
      _Interceptor.IncomingSMSReceived += new
         MessagingService.IncomingSMSReceivedEventHandler
         (Interceptor_IncomingSMSReceived);
      _Telephony.IncomingCallReceived += new
      _Telephony.IncomingCallReceivedEventHandler
         (Telephony_IncomingCallReceived);
      //Here we create a shortcut to this application in the
      //Windows\Startup folder if it does not exist yet
      try
      {
         CreateStartupShortCut();
         _Interceptor.StartDetector();
         _Telephony.StartDetector();
      }
      catch (Exception ex)
      {
         MessageBox.Show(ex.Message, "Initializing app");
```

```
      }
   }

   //Because we want this form to run constantly in the
   //background, you will need to hide it when it is activated
   public void Form_Activated(object sender, EventArgs e)
   {
      this.Hide();
   }
```

To get your application to automatically run when the Windows Mobile OS starts up, there are generally two methods that most developers consider:

- **Placing the path to your application in the registry in the HKLM\Init key:**

  This method works well for native applications. For .NET CF applications, it may or may not execute successfully. This is because there is a possibility your application may run before the .NET CF classes are loaded. For .NET CF applications, this method is not desirable.

- **Placing a link to your application in the \Windows\Startup folder:**

  The better way to get your application to automatically start up is by placing a link to it in the `\Windows\Startup` folder. You can create your own `.LNK` (link) file quite easily. The format of the `.LNK` file follows: `<Length of the application file path>#<full application file path>`

Let's take a look at the function to create this `.LNK` file:

```
private void CreateStartupShortCut()
{
   string _startupPath =
      Environment.GetFolderPath
      (Environment.SpecialFolder.Startup);
   string _data = "";
   string _path = "";

   _startupPath = _startupPath.TrimEnd('\\') +
      "\\CRMLiveInterceptor.lnk";
   if (File.Exists(_startupPath) == false)
   {
      _path = "\"" + Assembly.GetExecutingAssembly()
         .GetName().CodeBase + "\"";
      _data = Convert.ToString (_path.Length) + "#" +
         _path;
      SaveTextToFile(_data, _startupPath);
   }
```

```
}

public bool SaveTextToFile(string strData, string FullPath)
{
   bool _result = false;
   StreamWriter objReader = default(StreamWriter);
   try
   {
      objReader = new StreamWriter(FullPath, false);
      objReader.Write(strData);
      objReader.Close();
      _result = true;
   }
   catch (Exception ex)
   {
      MessageBox.Show(ex.Message, "Creating lnk file");
   }
   return _result;
}
```

The final two functions in this form are event handlers that connect to the database and write the corresponding historical record to the database using the functions you've created earlier. This is done for every incoming SMS and phone call.

```
private void Interceptor_IncomingSMSReceived(string
   IncomingPhoneNumber)
{
   try
   {
      _PluginManager.GetActivePlugin.ConnectDatabase();
      _PluginManager.GetActivePlugin.
         InsertHistoricalRecordByPhone
         (IncomingPhoneNumber,
         (int)GlobalVariables.Communications.Incoming,
         "SMS received", "");
      _PluginManager.GetActivePlugin. DisconnectDatabase();
   }
   catch (Exception ex)
   {
      MessageBox.Show(ex.ToString(), "Incoming SMS
         received");
   }
}
```

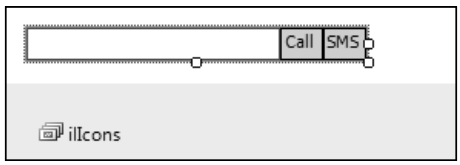
```
      private void Telephony_IncomingCallReceived(string
         incomingPhoneNumber)
      {
         try
         {
            _PluginManager.GetActivePlugin.ConnectDatabase();
            _PluginManager.GetActivePlugin.
               InsertHistoricalRecordByPhone(incomingPhoneNumber,
               (int) GlobalVariables.Communications.Incoming,
               "Phone call received", "");
            _PluginManager.GetActivePlugin.DisconnectDatabase();
         }
         catch (Exception ex) {
            MessageBox.Show(ex.ToString(), "Incoming call
               received");
         }
      }
   }
```

# Handling outgoing SMS messages and phone calls

When the user sends an SMS message or places a phone call through the sales force application, you would also need to generate a corresponding historical record.

There is no equivalent way to 'detect' an outgoing SMS message or phone call using POOM or the `SystemState` class without resorting to the MAPI functions, so we will look at an easier approach.

As you only need to log all outgoing phone calls and SMS messages sent through your application, you can create a general user control to do this (as shown in the following screenshot). This user control will be used in place of the normal text box in your application whenever you display phone numbers.



The **Call** button allows the application to dial the number contained in the adjacent text box, while the **SMS** button launches a custom SMS Compose screen. Create the SMS Compose form as shown in the following screenshot and name it `SendSMS`.


Whenever the user sends an SMS message through this window, you will generate the corresponding history record.

Let's take a look at the following code for this user control (called `FlexiControl`). The highlighted code generates the historical records. Take note that this user control needs a valid `AccountGUID`. You will need to pass in this value before using the user control.

```
using System;
using System.Windows.Forms;
using System.Drawing;

namespace CRMLive
{
   public partial class FlexiControl
   {
      private Guid _AccountGUID;
      public Guid AccountGUID
      {
         get
         {
            return _AccountGUID;
         }
         set
         {
            _AccountGUID = value;
         }
      }
```

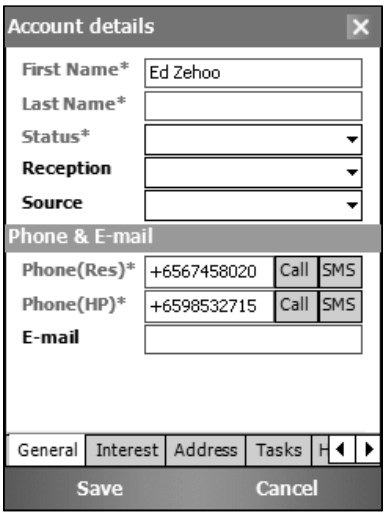
```
public override string Text
{
   get
   {
      return txtData.Text;
   }
   set
   {
      txtData.Text = value;
   }
}

public FlexiControl()
{
   // This call is required by the Windows Form
   //Designer.
   InitializeComponent();
}

public void FlexiControl_Resize(object sender,
   System.EventArgs e)
{
   this.Height = txtData.Height;
}

public void btnMakeCall_Click(System.Object sender,
   System.EventArgs e)
{
   Telephony _call = new Telephony();
   GlobalArea.PluginManager.GetActivePlugin.
   InsertHistoricalRecord(_AccountGUID,
   System.Convert.ToInt32
   (GlobalVariables.Communications.Outgoing),
   "Outgoing phone call", "");
   _call.MakePhoneCall(txtData.Text);
}

public void btnSendSMS_Click(System.Object sender,
   System.EventArgs e)
{
   //Here we launch the SendSMS Custom Compose window
   SendSMS _sendSMSForm = new SendSMS();
   if (_sendSMSForm.ShowDialog() == DialogResult.OK)
   {
      GlobalArea.PluginManager.GetActivePlugin.
      InsertHistoricalRecord(_AccountGUID,
      System.Convert.ToInt32
```

```
                (GlobalVariables.Communications.Outgoing),
                "Outgoing SMS", "");
            MessagingService _sms = new MessagingService();
            _sms.CreateSMS(txtData.Text, sendSMSForm.Message);
            _sms.SendSMS();
        }
    }
  }
}
```

Now that you have done this, you can replace all phone number text boxes in the `AccountViewer` form with this new user control. As this control still exposes the `Text` property, data binding will work as usual. The only extra step you need to do is to initialize this control with the current `AccountGUID` in the form load event of the `AccountViewer` form (as shown in the following code snippet):

```
fcLandPhone.AccountGUID = _account.AccountGUID
fcMobilePhone.AccountGUID = _account.AccountGUID
```

The following screenshot shows what the new `AccountViewer` screen will look like after your changes:




# Testing your code

You can try what you've built so far! Now, obviously you can only test your application with a real device. You will not be able to send or receive an SMS/phone call via the mobile device emulator.

After compiling the relevant projects, run the `CRMLiveInterceptor` project. You won't be able to see much (as the form is hidden). You will know that the application has started after the Windows Mobile **Busy** icon goes away.

Create a new lead account and set the **Phone (HP)** field to your friend's number. (Take care to specify the + and country code before the number and also not to type in any whitespaces in between numbers).

You can create more robust code by handling whitespaces, country and area codes, number separators (such as brackets), and so on.

Now, get your friend to call your phone (you will need to pick up the call) or send your phone an SMS message. Again, you won't see much happen on your phone. If you go back into the lead account and click on the **History** tab, you will be able to see that the incoming SMS or phone call has been captured:


Now try using the `AccountViewer` form to place a phone call directly from a lead account. After the phone call has been successfully made, you can check the **History** tab again. You will notice a new entry—denoting that the outgoing call has been logged. You can also repeat your experiment with SMS messages.

# Synchronizing with Windows Mobile Contacts

Windows Mobile Contacts is a pretty useful tool provided with Windows Mobile that allows you to store a list of all your contacts. These contacts can also be readily synced with Microsoft Outlook (via ActiveSync) on the desktop to provide you with an always updated list of contacts.

If your users use the sales force application to create a lead, opportunity, or customer, why should it be any different? These are also your contacts and should, therefore, make their way into the Windows Mobile Contacts area. Using the `OutlookSession` class in the `Microsoft.WindowsMobile.PocketOutlook` namespace, you can access all the Contacts stored on your mobile device.

Let's see how we can write the following function code to sync the latest information in the sales force application to the Windows Mobile Contacts area:

```
using Microsoft.WindowsMobile.PocketOutlook;

public void SyncToWindowsMobileContacts(string
   mobilePhoneNumber, string firstName, string lastName,
   string address, string emailAddress)
{
   OutlookSession _outlookapp = new OutlookSession();
   PropertyDescriptor _contact;
   int _contactIndex;
   Contact _contactItem;
```

You can search through your Windows Mobile Contacts using the `Find()` function as follows. This function is flexible enough to allow you to search by any property in the `PocketOutlook.Contact` class.

```
   _contact = TypeDescriptor.GetProperties(typeof(Contact))
                        ["MobileTelephoneNumber"];
   _contactIndex = _outlookapp.Contacts.Items.Find(_contact,
                        mobilePhoneNumber);
```

If a result returned from the search is `-1`, this means that a contact matching the mobile phone number passed in cannot be found, you should proceed to create the contact in this case. If the result was a value other than `-1`, you can retrieve the contact and update its details with the latest data from the sales force application.

```
   if (_contactIndex == - 1)
   {
      _contactItem = new Contact();
      _contactItem.FirstName = firstName;
      _contactItem.LastName = lastName;
      _contactItem.MobileTelephoneNumber = mobilePhoneNumber;
      _contactItem.AccountName = firstName + " " + lastName +
         "@CRMLive";
      _contactItem.Email1Address = emailAddress;
      _outlookapp.Contacts.Items.Add(_contactItem);
   }
   else
```

```
      {
         _contactItem =
            _outlookapp.Contacts.Items[_contactIndex];
         _contactItem.FirstName = firstName;
         _contactItem.LastName = lastName;
         _contactItem.Update();
      }
   }
```

Now that you have created this function, a suitable place to call it is when the user clicks on the **Save** button of the `AccountViewer` form. This is because you need to update the Windows Mobile Contacts area with the latest first name and last name of the lead account. Let's take a look at the `btnSave_Click()` function in the `AccountViewer` form. The changes in this function are highlighted as follows:

```
   private void btnSave_Click(System.Object sender,
      System.EventArgs e)
   {
      AccountBindingSource.EndEdit();
      if (Account.Validate == false)
      {
         Interaction.MsgBox(Account.GetValidationResult(),
            MsgBoxStyle.Exclamation, "Save error");
      }
      else
      {
         Telephony _telephony = new Telephony();
         _telephony.SyncToWindowsMobileContacts
         (txtMobPhone.Text, txtFirstName.Text,
         txtLastName.Text, txtStreet.Text, txtEmail.Text);
         this.DialogResult = Windows.Forms.DialogResult.OK;
         this.Close();
      }
   }
```

You can test this functionality by running the application and subsequently creating or updating an existing lead, opportunity, or customer. You will notice that the account's information is automatically propagated to the Windows Mobile Contacts area (as shown in the next screenshot).


# Synchronizing with Windows Mobile Tasks

The Windows Mobile Tasks area (accessible through **Programs** | **Tasks**) allows you to keep track of your personal tasks. You can also synchronize the sales force tasks with the tasks in this area.

Each Windows Mobile Task item contains a unique `ItemId` property value. By retrieving these values and saving them together in the `AccountTasks` table, you can associate each Windows Mobile Task item with a task in your sales force application.

Let's take a look now at how you can create the code to access Windows Mobile tasks. Through the same `OutlookSession` class, you can find a particular Windows Mobile Task items using its `ItemId` property.

```
using Microsoft.WindowsMobile.PocketOutlook;

public string SyncToWindowsMobileTasks(string taskSubject,
   string taskDescription, DateTime taskDueDate, string
   taskID)
{
   OutlookSession _outlookapp = new OutlookSession();
   PropertyDescriptor _task;
   int _taskIndex;
   Task _taskItem;

   _task =
      TypeDescriptor.GetProperties(typeof(Task))["ItemId"];
   _taskIndex = _outlookapp.Tasks.Items.Find(_task, taskID);
```

If a task with the matching item ID cannot be found, you will need to create a new task. However, if a task with a matching ID is found, you must then update the subject, body, and due date of that task item with the latest data from your sales force application.

```
        if (_taskIndex == - 1)
        {
            _taskItem = new Task();
            _taskItem.Body = taskDescription;
            _taskItem.Subject = taskSubject;
            _taskItem.DueDate = taskDueDate;
            _taskItem.ReminderDialog = true;
            _taskItem.ReminderRepeat = true;
            _taskItem.ReminderSound = true;
            _taskItem.ReminderVibrate = true;
            _outlookapp.Tasks.Items.Add(_taskItem);
        }
        else
        {
            _taskItem = _outlookapp.Tasks.Items[_taskIndex];
            _taskItem.Body = taskDescription;
            _taskItem.Subject = taskSubject;
            _taskItem.DueDate = taskDueDate;
            _taskItem.Update();
        }
        return _taskItem.ItemId.ToString();
    }
```

A good place to call this function would be in the `btnSave_Click()` function in your `AccountViewer` form. You need to iterate through all the task items and run the `SyncToWindowsMobileTasks()` function on each task. This will be left to you as an exercise.

# Sharing an account between two devices

In any sales force application, it is a common requirement to share leads, opportunities, and customers with other mobile devices. Without any form of data-sharing capability, the data is locked in your device, and the only way to get it to another device is the unattractive option of rekeying in the data piece by piece.

There are a few ways to send raw data from one device to another. Infrared (IrDA) and Bluetooth are two examples. Before we go into the details of each implementation, let's first decide on a transmission format for your data.

As you know, an account (together with its file attachments, historical records, and tasks) can be conveniently represented using a single dataset object. The easiest way to transmit this account to another device would be to serialize this dataset into a byte array for easy transmission and then to deserialize it back into a dataset at the targeted device.

Let's take a look at the code that you can use to do this:

```
public static byte[] SerializeDataset(DataSet Data)
{
   StringWriter strWriter = new StringWriter();
   string _result;
   ASCIIEncoding _encoding = new ASCIIEncoding();
   Data.WriteXml(strWriter);
   _result = strWriter.GetStringBuilder().ToString();
   return _encoding.GetBytes(_result);
}

public static DataSet DeserializeDataset(byte[] Data)
{
   ASCIIEncoding _encoding = new ASCIIEncoding();
   string _stringData;
   _stringData = _encoding.GetString(Data, 0, Data.Length);
   StringReader _reader = new StringReader(_stringData);
   DataSet _ds = new DataSet();
   _ds.ReadXml(_reader);
   return _ds;
}
```

Take note that this Accounts dataset contains only the file attachment metadata and not the actual file attachments themselves. You will need to write additional code to transmit these files one by one to the targeted device as well via Infrared or Bluetooth.

Now that you have decided on a suitable format for data transmission, let's see how you can transmit these byte arrays across Infrared and Bluetooth channels.

# Sharing an account between two devices using Infrared (IrDA)

Although new devices in the market nowadays don't really come with Infrared capability anymore, it is still a quick and cheap form of data transmission for older devices. Infrared requires line of sight during sending and receiving but, unlike Bluetooth, does not require the time-consuming process of device pairing.

Infrared functionality is based on a client-server model and is provided through the `IrDAListener` and `IrDAClient` classes. You can place both the client side and server side code in the same class. Let's take a look at the following code for this class:

```
using System;
using System.Data;
using System.Net;
using System.Net.Sockets;
using System.IO;
using System.Threading;

namespace CRMLive
{
   public class InfraredService
   {
      private IrDAListener _listener;
      private string _serviceName;
      private int _maxRetryCount;
      private IrDAClient _client = null;
      private Thread _serverThread;
      private byte[] _receivedData;
      public delegate void DataReceiveEndedEventHandler();
      private DataReceiveEndedEventHandler
         DataReceiveEndedEvent;

      public event DataReceiveEndedEventHandler
         DataReceiveEnded
      {
         add
         {
            DataReceiveEndedEvent =
               (DataReceiveEndedEventHandler)
               System.Delegate.Combine(DataReceiveEndedEvent,
               value);
         }
         remove
```

```
        {
            DataReceiveEndedEvent =
                (DataReceiveEndedEventHandler)
                System.Delegate.Remove(DataReceiveEndedEvent,
                value);
        }
    }
```

Because the server listens for Infrared connections in a blocking call, it's a good idea to run it in a thread. The following code shows how this is done:

```
public bool StartServer()
{
    _serverThread = new Thread(new
        System.Threading.ThreadStart(ReceiveFile));
    _serverThread.Start();
}

public bool StopServer()
{
    try
    {
        if (_serverThread != null)
        {
            _serverThread.Abort();
            _serverThread = null;
        }
    }
    catch (Exception ex)
    {
        throw(ex);
    }
}
```

This is the function that sets up the Infrared device to listen for connections. Once data is received via this connection, it is written into a byte array.

```
private void ReceiveFile()
{
    int _bytesRead = 0;
    IrDAListener listener = new
        IrDAListener(_serviceName);
    IrDAClient client = null;
    System.IO.Stream _stream = null;
    byte[] _byteArray = new byte[1048576];
```

```
            string _str = string.Empty;

            _receivedData = null;
            try
            {
               listener.Start();
               client = listener.AcceptIrDAClient();
               _stream = client.GetStream();
               int counter = 0;
               do
               {
                  _bytesRead = _stream.Read(_byteArray, counter,
                  8192);
                  counter += _bytesRead;
               } while (!(_bytesRead == 0));
            }
            catch (Exception ex)
            {
               throw(ex);
            }
            if (_stream != null)
            {
                _stream.Close();
            }
            if (client != null)
            {
                client.Close();
            }
            listener.Stop();
            if (DataReceiveEndedEvent != null)
               DataReceiveEndedEvent();
            _receivedData = _byteArray;
         }

         public bool Initialize(string ServiceName)
         {
            _serviceName = ServiceName;
         }

         public byte[] GetReceivedData()
         {
            return _receivedData;
         }
```

Now let's take a look at the client-side code. You can connect to a listening IrDA server by creating a new IrDA connection with the same service name as the server.

```
public bool StartClient()
{
   int _retryCount = 0;
   do
   {
      try
      {
      _client = new IrDAClient(_serviceName);
      }
      catch (Exception ex)
      {
         if (_retryCount >= _maxRetryCount)
         {
            throw (ex);
         }
      }
      _retryCount++;
   } while (_client == null && _retryCount <
      _maxRetryCount);

   if (_client == null)
   {
      return false;
   }
   else
   {
      return true;
   }
}
```

You can serialize a dataset using the functions you've created earlier into a byte array, which can then be transmitted through this IrDA channel using a stream object.

```
public bool SendData(byte[] byteArray)
{
   System.IO.Stream _stream = null;
   try
   {
      _stream = _client.GetStream();
      _stream.Write(_byteArray, 0, byteArray.Length);
   }
```

```
            catch (Exception ex)
            {
               throw(ex);
            }
            if (_stream != null)
            {
               _stream.Close();
            }
         }

         public int MaxRetryCounts
         {
            get
            {
               return _maxRetryCount;
            }
            set
            {
               _maxRetryCount = value;
            }
         }

         public bool StopClient()
         {
            try
            {
               if (_client != null)
               {
                  _client.Close();
               }
            }
            catch (Exception ex)
            {
               throw(ex);
            }
         }
      }
   }
```

To use this class, you can call the following code on the client side to send the dataset across:

```
InfraredService client = new InfraredService();
client.Initialize("MYSERVICE");
client.StartClient();
client.SendData(Generic.SerializeDataset(myDataset));
```

At the server side, you can use the following code to set up the Infrared connection. Once the dataset is received, the `DataReceiveEnded` event is raised. You can then obtain the received byte array using the `GetReceivedData()` function.

```
InfraredService server = new InfraredService();
server.Initialize("MYSERVICE");
server.StartServer();
```

# Sharing an account between two devices using Bluetooth

Bluetooth technology has gained widespread support ever since its inception and remains the best way to transfer large amounts of data wirelessly to another device. Bluetooth devices can exchange information anywhere from 10 meters to 100 meters apart.

Unlike Infrared, there is no built-in support for Bluetooth transmission in the .NET Compact Framework. Bluetooth, however, supports the serial port profile, which allows the data to be exchanged over serial communications. Fortunately, the latest version of the .NET Compact Framework provides the `SerialPort` class in the `System.IO.Ports` namespace, which allows you to easily do this. Let's take a look at what is needed in the server-side code for Bluetooth communications.

```
using System.Diagnostics;
using System;
using System.Data;
using System.IO;
using System.IO.Ports;

namespace CRMLive
{
   public class BluetoothService
   {
      public delegate void DataReceiveEndedEventHandler();
      private byte[] _ReceivedData;
      private SerialPort _port;
      private DataReceiveEndedEventHandler
         DataReceiveEndedEvent;

      public event DataReceiveEndedEventHandler
         DataReceiveEnded
      {
         add
         {
```

```
            DataReceiveEndedEvent =
               (DataReceiveEndedEventHandler)
               System.Delegate.Combine(DataReceiveEndedEvent,
               value);
         }
         remove
         {
            DataReceiveEndedEvent =
            (DataReceiveEndedEventHandler)
            System.Delegate.Remove(DataReceiveEndedEvent,
            value);
         }
      }
```

At the server side, we only need to create a new `SerialPort` object and assign an event handler for the `DataReceived` event. Whenever data is received through the port, your event handler is called.

```
public void StartServer()
{
   _port = new SerialPort();
   _port.DataReceived += new
      SerialDataReceivedEventHandler(DataReceived);
}

private void DataReceived(object sender,
   SerialDataReceivedEventArgs e)
{
   int _counter;
   int _bytesRead;
   byte[] Buffer = new byte[1048577];
   _bytesRead = 0;
   try
   {
      _counter = 0;
      do
      {
         _bytesRead = _port.Read(Buffer, _counter,
            8192);
         _counter += _bytesRead;
      } while (!(_bytesRead == 0));
   }
   catch (Exception ex)
   {
      throw (ex);
```

```
            }
            _ReceivedData = Buffer;
            if (DataReceiveEndedEvent != null)
               DataReceiveEndedEvent();
         }

         public byte[] GetReceivedData()
         {
            return _ReceivedData;
         }
```

On the client side, you can just as easily send data to the server. You must first create a `SerialPort` object, set up its connection parameters, and then use the `Write()` function to write binary data to the port.

```
         public bool SendData(byte[] Data)
         {
            try
            {
               _port = new SerialPort();
               _port.PortName = "COM1";
               _port.BaudRate = 9600;
               _port.Parity = Parity.None;
               _port.DataBits = 8;
               _port.StopBits = StopBits.One;
               _port.Open();
               _port.Write(Data, 0, Data.Length);
               _port.Close();
            }
            catch (Exception ex)
            {
                   throw (ex);
            }
         }
      }
   }
```

To use this class, you could call the following code on the client side:

```
BluetoothService client = new BluetoothService();
client.SendData(Generic.SerializeDataset(myDataset));
```

As for the server side, you can use the following code. Once the dataset is received, the `DataReceiveEnded` event is raised. You can then obtain the received byte array using the `GetReceivedData()` function.

```
BluetoothService server = new BluetoothService();
server.StartServer();
```

**Sending files across devices**

As the preceding code allows you to transfer binary data across devices, you can reuse the same code to send any type of object (including files) with a little extra work.

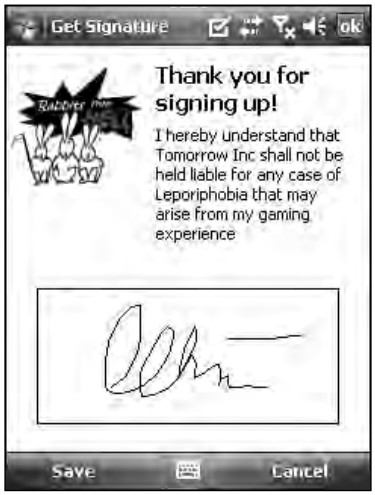
# Capturing handwritten input using the Smart Device Framework

The Smart Device Framework consists of a set of controls and classes that exposes a broad range of mobile device functionality to the .NET programmer. One of these controls is the Signature control, which allows the end user to perform freehand drawing (using the mobile device stylus). Let's take a look at how you can use this control in your sales force application to capture the customer's signature.

When the end user saves the signature, it will be saved as a PNG image file, and will eventually need to be committed to the `AccountFiles` table. As you already have a facility to load and save file attachments in your `AccountViewer` form, let's take advantage of this existing functionality.

You will need to make some changes so that when the user clicks on **New** in the **Files** tab, the user is prompted to choose if he or she wants to upload a signature or a normal file. If the end user chooses **No**, it will launch the default `FileDetailViewer` form that you've built. If the user chooses **Yes**, it will launch the new signature-capturing form.

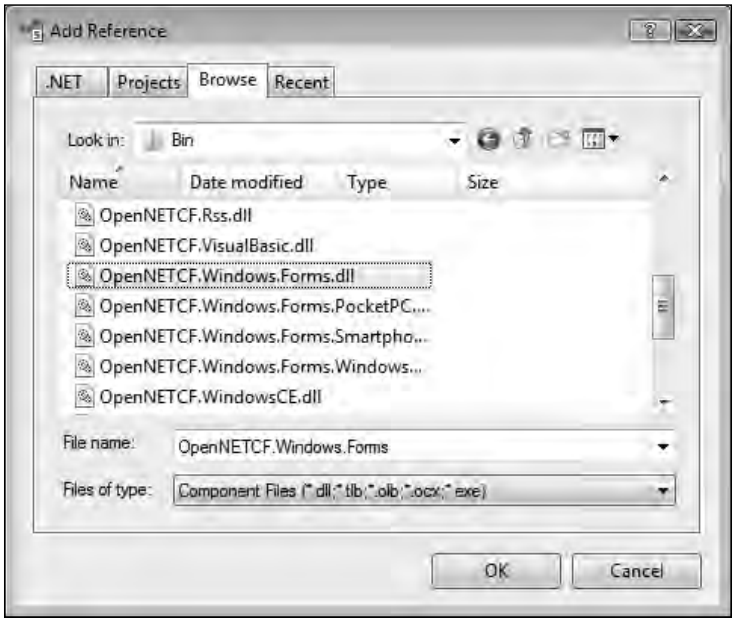
The signature-capturing form will display the Smart Device Framework's Signature control in the bottom half of the screen, with some text and images at the top. When the user clicks on **Save**, the signature is saved into a PNG image file. It will output a `File` object just like any other ordinary file attachment and can, therefore, be easily integrated into your sales force application.




Now that you have an idea how this works, let's start building the functionality. The first thing you need to do is to install the Smart Device Framework. You can download the Smart Device Framework from the following URL by clicking the **Download the Community Edition (free)** link:

```
http://opennetcf.com/CompactFramework/Products/SmartDeviceFramework/
tabid/65/Default.aspx
```

After you've installed the Smart Device Framework, add a reference to the `OpenNETCF.Windows.Forms.dll` library. This is typically located in the installation folder of the Smart Device Framework.




Once you have done this, you should now create the signature-capturing form. Name this form `GetSignature`.

**Visual designing support for the Smart Device Framework**

If you are using the free version of the Smart Device Framework, one of its limitations is that you will not be able to visually drag and drop the control onto the form as you normally would. You will have to create and add the control to the form using code.

Let's take a look at the code for this form:

```
using System;
using System.Windows.Forms;
using System.Drawing.Imaging;
using System.IO;
using System.Reflection;
using OpenNETCF.Windows.Forms;

namespace CRMLive
```

```
{
   public partial class GetSignature
   {
      private File _File;
      private FileManager _FileManager;
      private Signature _signature = new Signature();

      //Constructor for the form. Take note that we will pass
      //in a File object to this form
      public GetSignature(File FileObject)
      {
         InitializeComponent();

         _File = FileObject;
         _FileManager = new FileManager();
      }
```

The following code in the form load event generates the control and places it at the desired coordinates:

```
      public void GetSignature_Load(object sender,
         System.EventArgs e)
      {
         _signature.Name = "Signature1";
         _signature.Location = new System.Drawing.Point
            (20, 160);
         _signature.Size = new System.Drawing.Size(200, 90);
         this.Controls.Add(_signature);
      }
```

When the user clicks the **Save** button, you will first need to save the contents of the Signature control to a PNG image file. After that, you will use the `FileManager` class you've created earlier in *Chapter 2* to upload the file and return a relative file path. You will need to then fill in the `File` object passed in to this form with the details of the signature file.

```
      public void mnuSave_Click(System.Object sender,
         System.EventArgs e)
      {
         string _filePath;
         FileInfo _fileinfo;
         string _tempPath;
         _tempPath =
            System.IO.Path.GetTempPath().TrimEnd('\\');
```

```
            _filePath = _tempPath + "\\" +
               Guid.NewGuid().ToString() + ".png";
            _signature.ToBitmap().Save(_filePath,
               ImageFormat.Png);
            _fileinfo = new System.IO.FileInfo(_filePath);
            _File.Attachment = _FileManager.Store(_filePath);
            _File.AttachmentName = "Signature";
            _File.AttachmentSize = (int) _fileinfo.Length;
            this.DialogResult =
               System.Windows.Forms.DialogResult.OK;
            this.Close();
         }

         public void mnuCancel_Click(System.Object sender,
            System.EventArgs e)
         {
            this.DialogResult =
               System.Windows.Forms.DialogResult.Cancel;
            this.Close();
         }
      }
   }
```

Now, let's take a look at the changes required in the `AccountViewer` form. Look at the `btnNewFile_Click` function in the `AccountViewer` form. The changes required for this function are highlighted in the following code snippet:

```
public void btnNewFile_Click(System.Object sender,
   System.EventArgs e)
{
   File _File;
   DialogResult _dialogResult;
   _dialogResult=MessageBox.Show("Choose Yes to upload a
   signature, choose No to upload a normal file", "Add a
   file", MessageBoxButtons.YesNoCancel,
   MessageBoxIcon.Question,
   MessageBoxDefaultButton.Button1);
   switch (_dialogResult)
   {
      case DialogResult.Yes:
      _File = _account.NewFile();
      GetSignature _Signature = new GetSignature(_File);
      if (_Signature.ShowDialog() ==
      System.Windows.Forms.DialogResult.OK)
         {
            _account.Files.AddFile(_File);
            dgFiles.DataSource = null;
```

```
            dgFiles.DataSource = _account.Files;
          }
          _Signature.Close();
          _Signature.Dispose();
          _Signature = null;
          break;
          case DialogResult.No:
          _File = _account.NewFile()
          FileDetailViewer _EditView = new
          FileDetailViewer(_File);
          if (_EditView.ShowDialog() == DialogResult.OK)
            {
              _account.Files.AddFile(_File);
              dgFiles.DataSource = null;
              dgFiles.DataSource = _account.Files;
            }
          _EditView.Close();
          _EditView.Dispose();
          _EditView = null;
          break;
  }
  }
```

After you have done this, try creating a new file in your `AccountViewer` form, and choose **Yes** to capture your signature. You will find that it is saved to the database like any other file attachment.

# Summary

In this chapter, you've taken a look at various forms of integration with Windows Mobile and the mobile device. You have learned how to do the following:

- Detect incoming SMS messages and phone calls
- Send outgoing SMS and e-mail messages
- Place phone calls
- Synchronize with Windows Mobile Contacts and Tasks
- Transfer binary data using Infrared
- Transfer binary data using Bluetooth
- Capture and store signatures using the Smart Device Framework's Signature control

In the next chapter, you'll take a look at the all-important task of synchronizing data between your mobile device and a remote repository.

## Where to buy this book

You can buy .NET Compact Framework 3.5 Data-Driven Applications from the Packt Publishing website: `https://www.packtpub.com/data-driven-applications-with-.net-compact-framework-3-5/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.