**Windows Mobile Application Development**

**(Part 3)**

**Topic 1: Basic Windows Forms Application Development for Windows Mobile Devices**



[**http://msdn.microsoft.com/rampup**](http://msdn.microsoft.com/rampup)

**Introduction**

Developing applications for Windows Mobile Devices is similar to developing applications for the desktop, especially if you make use of either Visual Basic.NET or Visual C#. You can use the same development tools to develop your applications, although there are also differences between developing applications for Windows Mobile Devices and developing desktop applications. Devices have smaller screens, limited resources, are extremely portable and are usually powered by batteries.

This article provides information on how to develop Windows Forms based applications for Windows Mobile Devices. You will learn how to develop applications that can make use of device capabilities, that are battery friendly and that display correctly regardless of the display orientation mode of the device.

**Different Devices have Different Form Factors**

There are a large number of different Windows Mobile Devices on the market. These devices not only have different form factors, they can also run different versions of the Windows Mobile Platform. In this article we will concentrate on Windows Mobile 6 devices (both Standard and Professional).

Windows Mobile 6 Standard Devices

These devices were formally known as Smart Phones. All these devices have phone capabilities and they do not have touch screens. They are typically used in a one-hand operation. To develop applications for Windows Mobile 6 Standard Devices you need the Windows Mobile 6 Standard SDK with Visual Studio 2005 Professional or better.

Windows Mobile 6 Professional Devices

These devices were formally known as Pocket PC’s. These devices might have phone capabilities and they all have touch screens. They are typically used in a two-hand operation, in combination with a stylus or finger controlled. To develop applications for Windows Mobile 6 Professional Devices you need the Windows Mobile 6 Professional SDK in combination with Visual Studio 2005 Professional or better.

**Developer Experience**

There is not that much difference in developing managed applications using Visual Studio 2008 for the desktop and Windows Mobile Devices. The developer experience is similar. When you start Visual Studio 2008 you can decide what type of project you want to create. Once you have decided on the project type and target platform, Visual Studio will create an initial project for you and displays an empty form on which you can start developing your User Interface. The biggest difference when developing for Windows Mobile devices is the limited screen size and the variety of different screen sizes. Another difference becomes clear when you start dragging User Interface Controls from the Visual Studio 2008 Toolbox to drop them onto your form. You will see that you only have a limited number of User Interface Controls available compared to desktop developers. When you are developing a Windows Mobile 6 Standard application, the number of User Interface Controls that are available out of the box is even more limited then when you are developing an application for a Windows Mobile Professional device. Since Windows Mobile 6 Standard Devices do not have touch screens, some of the controls that are available for Windows Mobile Professional devices simply don’t make sense to use.

Creating a User Interface for your application simply means dragging controls from the Toolbox and dropping them anywhere on a Form or dropping them inside a container control that is already added to the form. You of course have the possibility to create multiple forms inside your application and you can create your own controls, either derived from existing controls or you can create controls yourself from scratch. Besides ‘simple’ User Interface controls a number of common dialogs are also available to give you a head start and to make your application look consistent with other applications that are running on a Windows Mobile device.

**Designing a simple Windows Mobile 6 Professional User Interface**

To start developing a new Device Application you first need to create a new project in Visual Studio 2008. The project should be a Smart Device project in either C# or Visual Basic.NET, targeting the Windows Mobile 6 Professional SDK. When you are creating a new Device Application, Visual Studio 2008 will show you an empty form with the correct dimensions for the target device, even including a device skin to match a real device as close as possible. Using the **Device Controls** toolbox that is displayed in figure 1, you can drag and drop controls to the application’s form.

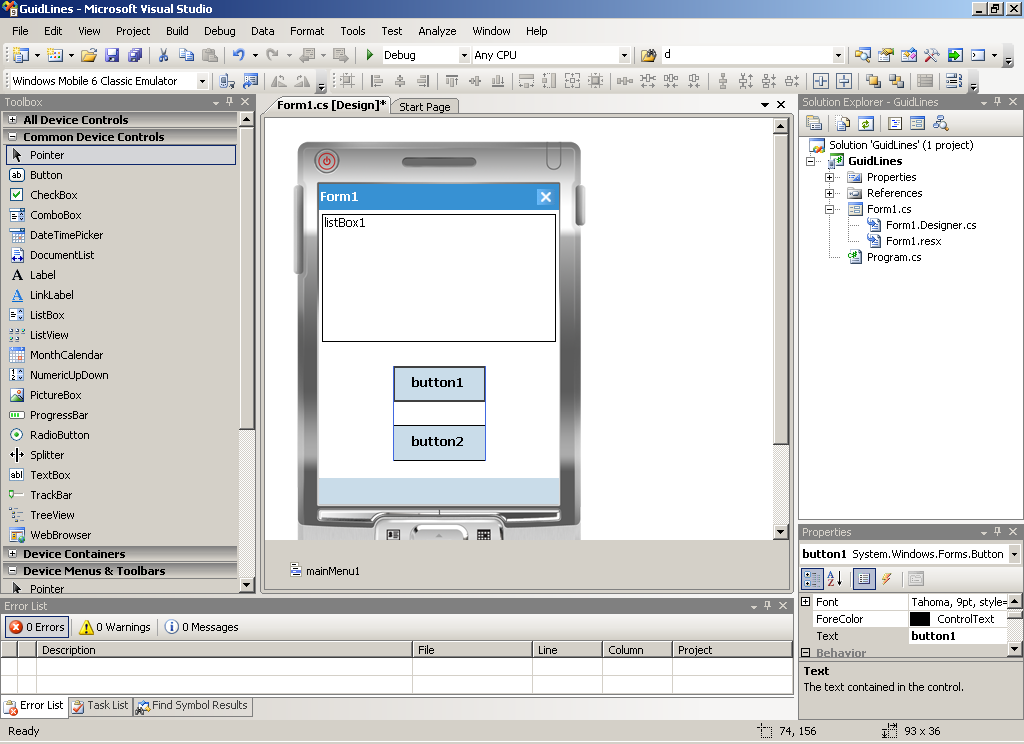


Figure - Visual Studio 2008 in Forms Design View

Take a look at the individual controls that you can use to develop your User Interface. If you are not familiar with the new controls, the best thing you can do is create a simple application, using the new controls, take a look at the properties and study their behavior. You should of course also take a look at the online help for the particular control. If you start designing your user interface, one of the things that immediately gets your attention is how the User Interface designer assists you in aligning controls and suggesting a minimum distance between controls. The automatic display of guide lines will help you tremendously in creating a good looking user interface in a short time.

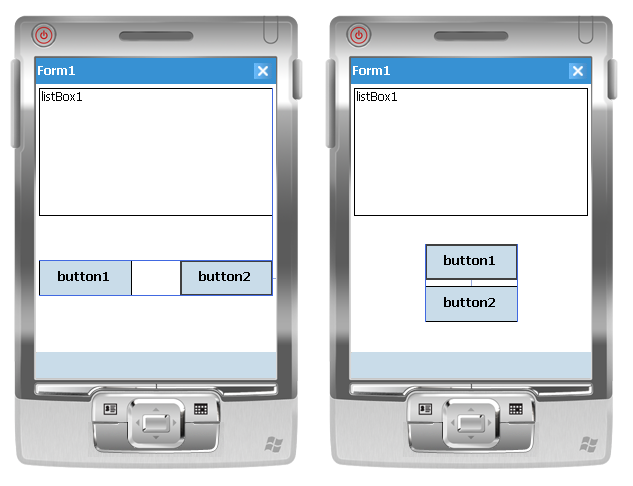


Figure - Designer Guide Lines will help you laying out your User Interface

**Creating one application that correctly deals with display orientation modes**

The large number of different devices with different capabilities and form factors make developing one single application that targets multiple devices a bit challenging. Visual Studio will support you by providing docking and anchoring functionality for User Interface controls. The Windows Mobile 6 SDKs will support you by providing information about different device capabilities and operating system version numbers. In this article we limit ourselves to creating Windows Forms based applications that has the following requirements:

* Be resource and battery friendly
* Targets Windows Mobile 6 Devices
* Displays correctly both in Portrait and Landscape Mode and on Devices with Square Screens

Even though Windows Mobile 6 Standard devices are not making use of different display orientation modes, it is thinkable that this will change in the future. For some operations, displaying information in landscape mode gives end users a better experience. Visual Studio 2008 will support you when designing Windows Forms based User Interfaces by allowing you to dock and anchor User Interface Controls.

Docking and Anchoring Controls

To make sure that a form is displayed correctly in both portrait and landscape mode, it is possible to dock and anchor controls to a specified location inside a parent control like a Form or a Panel Control. There is a subtle, yet important difference between docking and anchoring. When you anchor a control to one or more edges of a container, you make sure that the relative position to the anchored edges is maintained when the display orientation of the device is changed. When you dock a control, you specify the edge of the container against which the control will be positioned. You can even specify that a control should be docked to all edges, meaning that the control will fill the entire real estate of the parent control.

To show the difference between anchoring and docking and to show the behavior of anchored and docked controls in different display orientations, you can create a simple Form that contains a number of Label controls—each with different background colors—and then you can simply look at the behavior of anchoring, docking, and orientation changes. The user interface design is not particularly meaningful, but it will help you understand the difference between docking and anchoring. It will also show the impact on changing from portrait mode to landscape mode. Suppose you have created a user interface that looks like Figure 1.

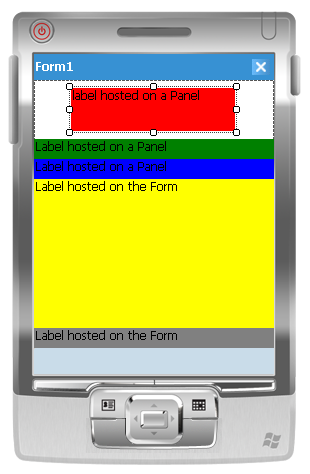


Figure - Windows Mobile 6 Professional in portrait mode, controls neither docked or anchored

The three upper labels are hosted on a panel; the two lower labels are hosted on the form itself. For all controls that you add on the form, make sure to explicitly remove anchoring. The default behavior of Visual Studio 2008 is to anchor all controls to the top and left edges or their hosting containers. The user interface looks fine in portrait mode, but when you change the orientation to landscape mode, you will run into problems, as shown in Figure 2.

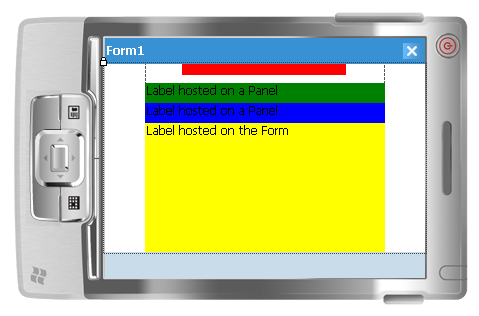


Figure - Windows Mobile 6 Professional in landscape mode, controls neither docked or anchored

You can immediately see that not all of the labels that showed up in portrait mode are visible in landscape mode. Part of the panel, including part of the top label, is off of the screen. Also the bottom label has disappeared from the screen, and the right and left sides of the display are not filled in landscape mode. It looks like landscape mode merely views part of the portrait mode's user interface.

To solve this problem, without having to write actual code, you can make use of docking and anchoring. You can simply anchor a control to one or more edges of the screen by setting the **Anchor** property. Anchoring a control means that the relative position to the anchored edge always remains the same—independent of display orientation. By using the **Dock** property, you can specify how a particular control aligns to an edge of its parent control or fills up an area of the screen. Docking multiple controls to the same edge of their parent control causes the controls to stack upon the other. In our sample, the upper three labels are all owned by a panel control. The topmost label is not docked, but it is anchored to the left, top, and right edges of the panel control, which in turn is docked to the top of the form. Independent of the display orientation, the distance between the left, top, and right edges and the label will be the same. The label will be resized when the user interface is displayed in landscape mode. The other two labels inside the panel control are displayed on top of each other; both are docked to the bottom of the panel control. The tab order determines the way in which these labels are displayed. If you now change the display's orientation to landscape mode, all of the controls on the form will remain visible, and the entire size of the form is used to fill up the controls, as shown in Figure 3.



Figure - Mobile 6 Professional in landscape mode with docked and anchored controls

The user interface shown in Figure 3 looks much better than the one that is shown in Figure 2 because the individual controls have resized to properly display in landscape. To really get a feeling about how docking and anchoring influence the position of controls in both portrait and landscape mode, you should spent some time experimenting with it.

In Visual Studio 2008, it is very easy to view your user interface in both orientation modes—even in Designer view. To change the orientation of the device in Designer view, simply right-clicking on the device skin. In the popup menu select either **Rotate Left** or **Rotate Right** to change the screen display orientation, as shown in Figure 4. Alternatively, after selecting the entire form, you can use toolbar buttons to change the orientation.

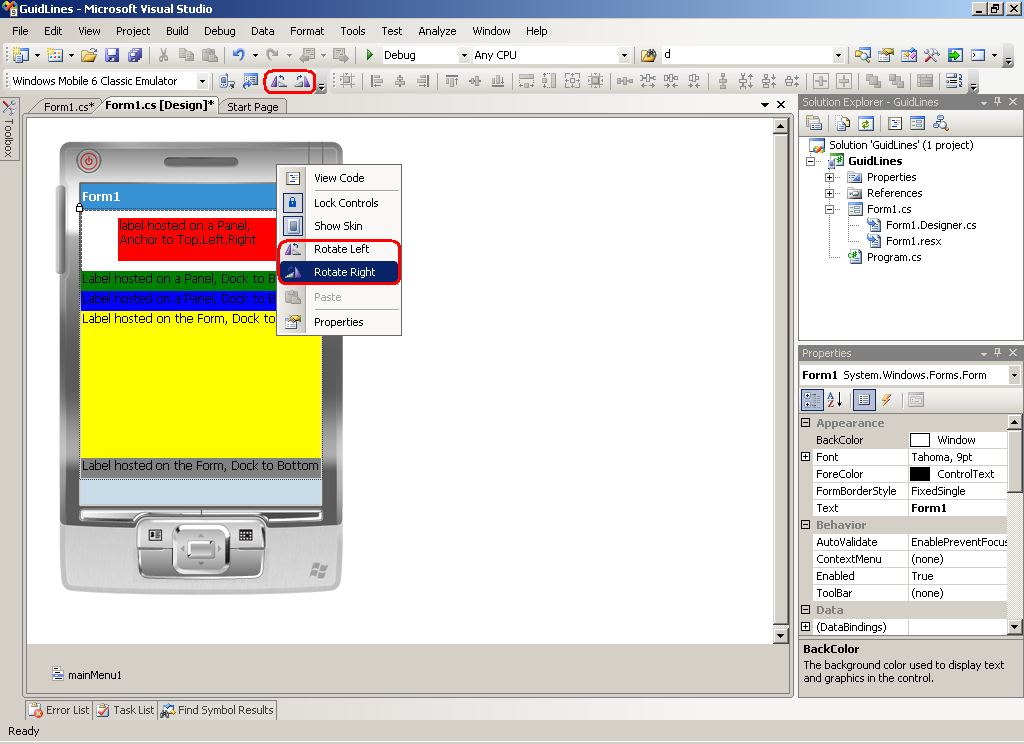


Figure - Changing display orientation in design mode

**Creating one application that runs on multiple devices**

If you are creating an application that targets a one of the Windows Mobile 6 SDK’s (either Professional or Standard), you can change the Target Platform in your solution to target the other SDK. However, in this case your application will be converted from one target to the other, meaning your application will be modified by Visual Studio 2008. If you take this approach to convert an application that originally targeted the Windows Mobile 6 Professional SDK you might run into some challenges. Since Windows Mobile 6 Standard Devices support less User Interface Controls then Windows Mobile 6 Professional Devices, unsupported controls must be converted to alternative controls in order to build an run the converted application on the new targeted platform. Visual Studio 2008 will help you converting a project from one targeted platform to another. Of course, from a software development point of view, it is a good idea to separate business logic from user interface logic to make target platform conversions as simple as possible. Not supported User Interface Controls on the new target platform will be marked by the Visual Studio 2008 Forms Designer, allowing you to convert these controls to the closest matching alternatives that are supported on the new target platform. If you don’t want to convert an existing project to target another platform you can take advantage of Visual Form Inheritance to maintain one single code base and still have the possibility to run your application on multiple platforms.

Visual Form Inheritance

Visual Studio 2008 gives you full designer support when deriving one Form from another. To show Visual Form Inheritance in action, let’s start with a simple Base Form that can be used as a starting point for an application that will target multiple Windows Mobile Platforms. Even though not absolutely necessary it makes a lot of sense to create a separate class library that will hold the Base Form. It is important to think ahead about the Base Form; you should know which controls can be modified in inherited forms and which controls not. To be able to modify controls or the behavior of controls, you must make sure to set the access modifier property of the particular control to something other than Private. In figure 7 you see a simple Base Form that you can use as starting point for your application.

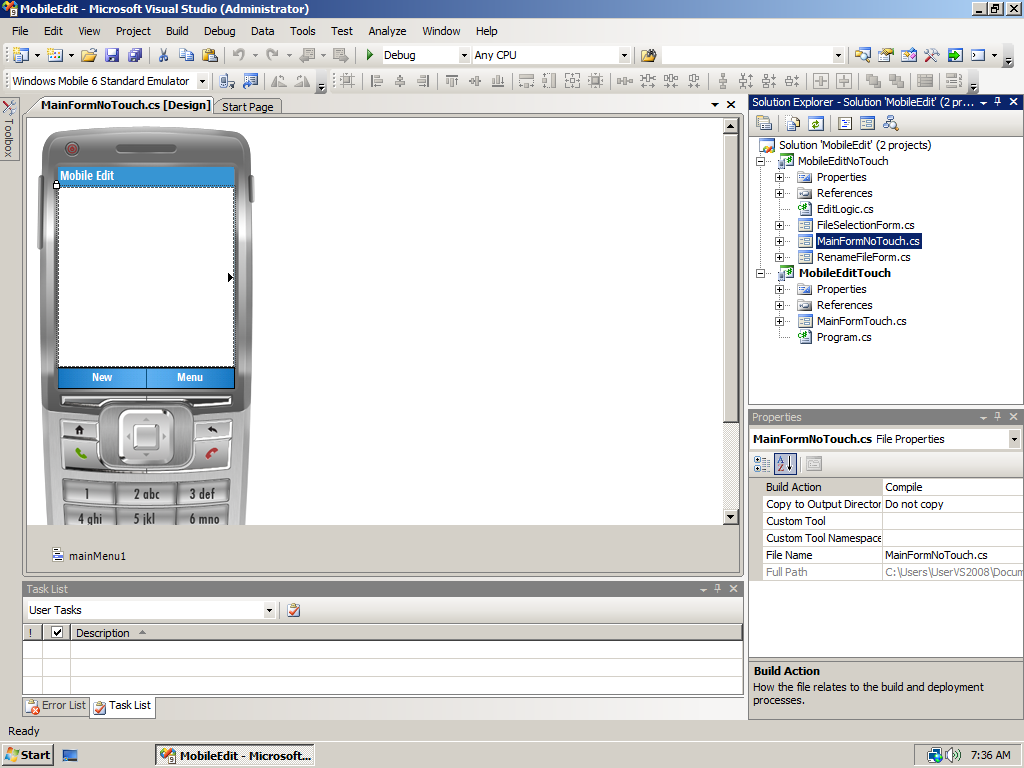


Figure - Base Form targeting a Windows Mobile Standard Device

The Base Form contains an Edit Control that is docked to fill up the screen and a menu. The application is a simple text editor. It can create new text files, modify existing text files and allows text files to be renamed and deleted. Even though the files created by this application are regular text files, they have a unique extension to easily filter them for file manipulation purposes. The Base Form is created for a Windows Mobile Standard Device. The reason is that Windows Mobile Standard Devices have fewer User Interface Controls available then Windows Mobile Professional Devices. In order to manipulate files, two additional forms are available to display a list of existing files and to rename existing files. Since Windows Mobile Standard Devices do not have touch screens, all actions on files are performed through menu items. In this article you will not get a description of the entire source code for the text editor. Focus will be on creating one single application to target multiple devices. However, the Mobile Edit sample application is available for download.

In figure 8 you will see the Main Form for a Windows Mobile Professional Device. Even though it looks different, it is derived from the Main Form that you already saw in figure 7, with the addition of one User Interface Control that is available for Windows Mobile Professional only, a Document List. Even though the Main Form contains an edit control, in the Forms Designer it is not visible, because the Document List is displayed on top of it. Inside the application the Document List will be hidden programmatically when it is not needed. Controls that are part of the base form are inherited on the derived form. You can see that they are inherited, because they show an arrow symbol in their upper left corner as is visible for the Menu Control.

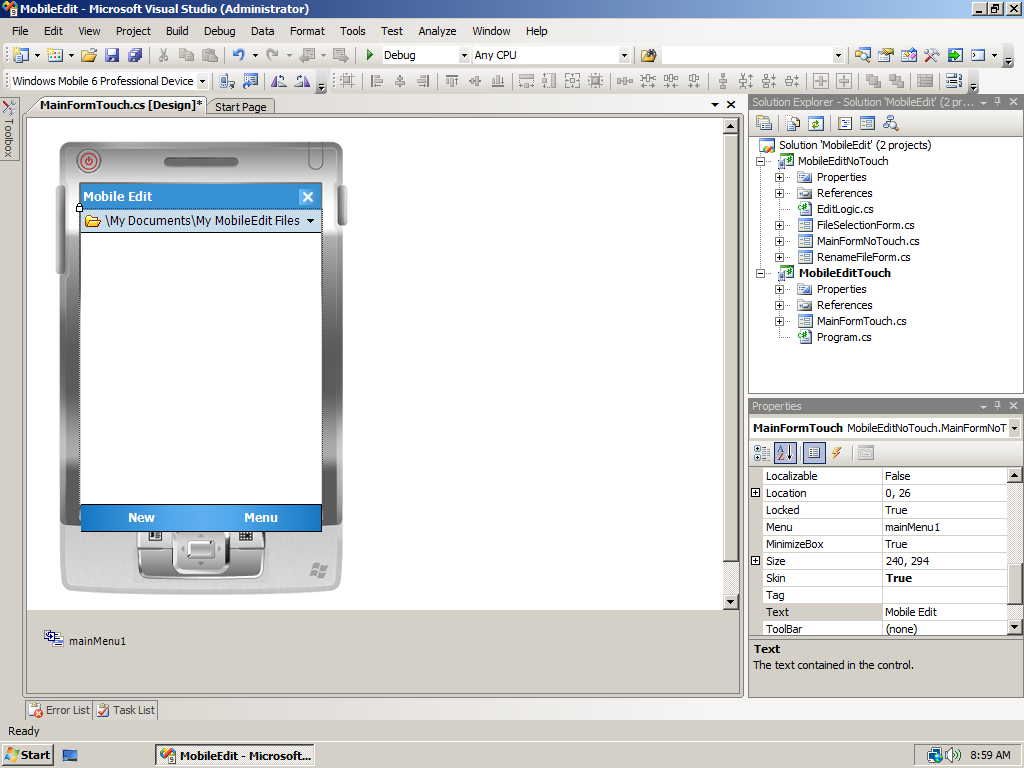


Figure - Derived Base Form for a Windows Mobile Professional Device

Since Windows Mobile Professional Devices support additional controls, the application itself can be simpler and can utilize touch screen functionality. Making use of a Document List to select and manipulate files means no additional forms are needed. Since the goal is to create one single binary that can target both Windows Mobile Standard and Windows Mobile Professional devices, during application startup we will determine what the target device is and display the appropriate form. The .NET Compact Framework 3.5 makes it easy to determine the device type, as you can see in the following code snippet that is part of Program.cs:

static void Main()

{

if (SystemSettings.Platform == WinCEPlatform.PocketPC)

{

Application.Run(new MainFormTouch());

}

else if (SystemSettings.Platform == WinCEPlatform.Smartphone)

{

Application.Run(new MainFormNoTouch());

}

}

Depending on the device type we either use MainFormNoTouch or MainFormTouch as our application’s Main Form. The true power of Form Inheritance is the fact that we are re-using functionality on the Main Form for Windows Mobile Professional devices. For instance, if the user clicks the New menu entry, regardless of the device they are executing the application on, the event handler for the New menu entry that is defined for the MainFormNoTouch screen will execute:

protected virtual void menuNew\_Click(object sender, EventArgs e)

{

autoSave = false;

if (tbWorkArea.Text.Length != 0)

{

if (MessageBox.Show("Save contents before creating a new file",

"Save current contents",

MessageBoxButtons.YesNo,

MessageBoxIcon.Question,

MessageBoxDefaultButton.Button1) == DialogResult.Yes)

{

EditLogic.WriteFile(fileName, tbWorkArea.Text);

}

tbWorkArea.Text = string.Empty;

}

autoSave = true;

fileName = EditLogic.FileName;

}

This code is responsible for asking the user to save any file they were working on before opening a new file. As you can see, the functionality of the editor itself is available through another class called EditLogic to separate the User Interface completely from the application’s logic. If the user is running on a Windows Mobile Professional Device, some additional functionality is needed to disable the Document List control. You can achieve that by overriding the menuNew\_Click event handler as you can see in the next code snippet:

protected override void menuNew\_Click(object sender, EventArgs e)

{

base.menuNew\_Click(sender, e);

documentList.Visible = false;

MinimizeBox = false;

}

When the user clicks the New menu entry on a Windows Mobile Professional Device, first the code from the base form is executed, after which two additional statements are executed that are exclusively available for Windows Mobile Professional Devices. These statements are responsible for hiding the Document List, thus making the derived edit control visible and to display an OK button on the Form’s title bar respectively.

To override the menuNew\_Click event handler, you need to add it manually to the MainFormTouch.cs source file. If you would simply double click the New menu entry in the Windows Forms designer, the designer would create a new event handler for you, called menuNew\_Click\_1. This event handler would be called after the original menuNew\_Click event handler in the base form has executed.

Resource Friendly Development

Each time the application’s MainForm (for both types of devices) becomes invisible, an event is generated. Similarly, each time the application’s MainForm becomes visible another event is generated. These events can be used to free up resources and for instance to disable timers when the application is moved to the background. In that way, the application will free up memory when it does not need it and will also make sure to use as little processing power as possible. If the Mobile Editor goes to the background it will temporarily save the contents of an edit session to a file and restore the contents in the text box when the application goes to the foreground again. The following code will execute when the application goes to the background to free up the contents of the text box **tbWorkArea**:

protected virtual void MainFormNoTouch\_Deactivate(object sender, EventArgs e)

{

if (fileName != string.Empty && tbWorkArea.Text.Length != 0)

{

EditLogic.WriteFile(fileName, tbWorkArea.Text);

tbWorkArea.Text = string.Empty;

autoSave = true;

}

}

The following code will execute when the application comes back to the foreground to restore the contents of the text box **tbWorkArea**:

protected virtual void MainFormNoTouch\_Activated(object sender, EventArgs e)

{

if (autoSave)

{

tbWorkArea.Text = EditLogic.ReadFile(fileName);

}

}

Creating an application that properly runs on both Windows Mobile Standard and Windows Mobile Professional is possible, but you have to think about the application’s architecture. The story becomes even more complex when you are also taking different hardware capabilities of different devices into consideration. This is beyond the scope of this whitepaper.



Figure - Mobile Edit in Action on various devices

You now should have a good idea how to use form inheritance in combination with the Windows Forms designer. Of course you can host any control on the base form, as long as you want to reuse it in derived forms. Just make sure to change the **Modifiers** property of the control to **protected** or **public** if you want to modify the behavior of the particular control in a derived form. With the **Modifiers** property set to anything but **private**, you can even hide the control, dock it or resize it on a derived form.

# **Additional resources and references**

* [Window Mobile Developer Center](http://msdn.microsoft.com/en-us/windowsmobile/default.aspx)
* [Device Emulator Reference](http://msdn.microsoft.com/en-us/library/aa188178.aspx)
* [Create and Run a simple Windows Application in Managed Code](http://msdn.microsoft.com/en-us/library/bb158524.aspx)
* [Using Visual Studio to Design User Interfaces and Data for Device Applications](http://msdn.microsoft.com/en-us/library/aa454904.aspx)
* [How do I build a .NET Compact Framework 3.5 Application for Windows Mobile 6 Devices?](http://msdn.microsoft.com/en-us/windowsmobile/bb975138.aspx)
* [How do I programmatically determine the Windows Mobile Device Type using .NET Compact Framework 3.5?](http://msdn.microsoft.com/en-us/windowsmobile/cc197100.aspx)