1 Getting Started

You have a Pocket PC device, and it's hooked up to your Windows-based computer. ActiveSync works, you've found your way around the interface of the device, and you'd like to put something on that screen that reflects your own ideas. It's time to fire up some development tools and put your own stamp on the machine.

Visual Studio .NET

For the first release of the .NET Compact Framework, you're stuck using Visual Studio .NET as your development environment. SharpDevelop, the Mono tools, or even command-line compilation might seem more appealing, but Microsoft hasn't provided any other choices for now. (They've talked about doing so in future releases.) Even buying Visual C#.NET or Visual Basic.NET won't help - these $99 packages don't include the .NET Compact Framework parts you need. You need Visual Studio .NET 2003 Professional Edition, a roughly $500 proposition if you have either some Microsoft or competing software from which to upgrade. Otherwise, it's more like $700.

It could be worse - they could have required the Enterprise Edition, which is $1500 and up.

Your likely first step in programming for your Pocket PC, should you choose to go the .NET Compact Framework route, is to spend as much as you paid for the Pocket PC on software that will let you write programs for it. Microsoft does offer a 60-day trial version of Visual Studio .NET, or you could be very brave and try the "Community Edition" betas of the next version of Visual Studio .NET. Unfortunately the free Express Editions also lack .NET Compact Framework support. At this point, it looks like Microsoft really wants you to pay them to develop software that will run on their devices.

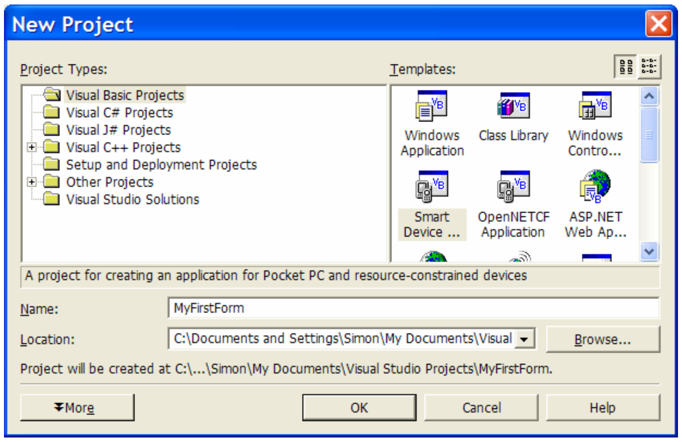
If you don't want to send your money to Redmond, there are a few other options. A version of Python that runs on Pocket PC is available from http://www.murkworks.com/Research/Python/PocketPCPython/.

PocketC, which is based on C but not ANSI-standard C, is at http://www.orbworks.com/wince/index.html, SuperWaba (which uses Java) at http://www.superwaba.com.br/, and even a version of Scheme at http://www.mazama.net/scheme/pscheme.htm. Undoubtedly there are more out there, and more will appear over time. If you felt especially strongly, you could run Linux on your Pocket PC, and get an even wider set of choices. All of these environments have pluses and minuses, one consistent minus being that they're not covered in this book. Is that a minus for them or for this book? You'll have to decide that for yourself.

Once you've purchased Visual Studio .NET, or received your 60-day trial, you'll get to spend a while installing it. You should budget a substantial amount of space on your hard drive, as well as a fair amount of time to watch the install program slowly crunch by. Finally, when you have Visual Studio .NET installed, you probably want to download one extra item: the Pocket PC 2003 SDK and emulator, which you can find at http://msdn.microsoft.com/mobility/downloads/sdks/default.aspx. Fortunately, Microsoft doesn't want to charge you for that piece. Now that you've done all that, it's time to create your first - and very simple - program for the Pocket PC.

Creating a program without writing code

You can't get very far in .NET Compact Framework development without writing some code, but you can create an initial demonstration that shows you how to create a project and deploy it to your Pocket PC. To get started, fire up Visual Studio and create a new project. You can go to File > New > Project..., click New Project on the Start Page, or press Ctrl-Shift-N. It doesn't matter. You'll see the dialog shown in Figure 1-1.



A project for creating an application for Pocket PC and resource-constrained devices

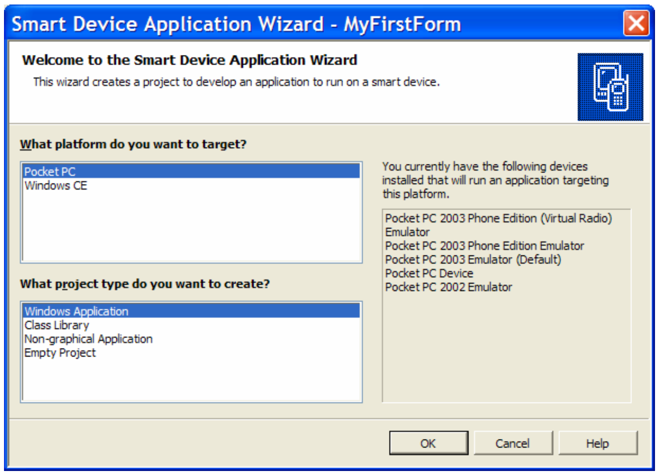
Name: MyFirstForm

Location: C:\Documents and Settings\Simon\My Documents\Visual

Project will be created at C:\...\Simon\My Documents\Visual Studio Projects\MyFirstForm.

Figure 1-1. The New Project dialog box.

For the project we'll be creating, you'll want Smart Device Application from the Visual Basic templates. Enter a name for your project, choose where you want it saved, and click OK. Next, Visual Studio will ask you more precisely which platform you're targeting, as shown in Figure 1-2.



Welcome to the Smart Device Application Wizard

This wizard creates a project to develop an application to run on a smart device.

What platform do you want to target?

- Pocket PC
- Windows CE

You currently have the following devices installed that will run an application targeting this platform.

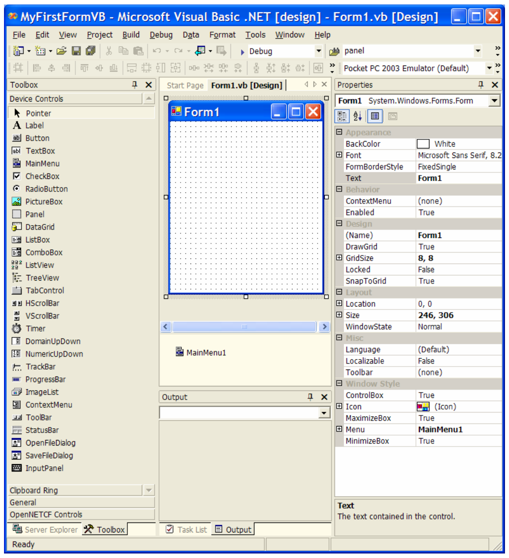
- Pocket PC 2003 Phone Edition (Virtual Radio) Emulator
- Pocket PC 2003 Phone Edition Emulator
- Pocket PC 2003 Emulator (Default)
- Pocket PC Device
- Pocket PC 2002 Emulator

What project type do you want to create?

- Windows Application
- Class Library
- Non-graphical Application
- Empty Project

Figure 1-2. The Smart Device Application Wizard.

Your list of installed devices may vary, but for now we want the default choices of "Pocket PC" and "Windows Application", so you can just click OK. Visual Studio will churn for a few moments, and then present the empty form and surrounding context shown in Figure 1-3. (If the Toolbox is missing, click the hammer and wrench icon at the left of the window and then click its pushpin icon to lock it in place.)



Form1 System.Windows.Forms.Form

Appearance	
BackColor	White
Font	Microsoft Sans Serif, 8.2
FormBorderStyle	FixedSingle
Text	Form1
Behavior	
ContextMenu	(none)
Enabled	True
Design	
(Name)	Form1
DrawGrid	True
GridSize	8, 8
Locked	False
SnapToGrid	True
Layout	
Location	0, 0
Size	246, 306
WindowState	Normal
Misc	
Language	(Default)
Localizable	False
Toolbar	(none)
Window Style	
ControlBox	True
Icon	(Icon)
MaximizeBox	True
Menu	MainMenu1
MinimizeBox	True

Text
The text contained in the control.

Figure 1-3. Visual Studio at the beginning of a Pocket PC project.

The heart of the application, the Form that we're going to create and display, is at the center of the screen. Controls are on the left. You can drag-and-drop controls from the left onto the form, or you can click on a control name and then draw it on the form. Below the form itself is a menu class.

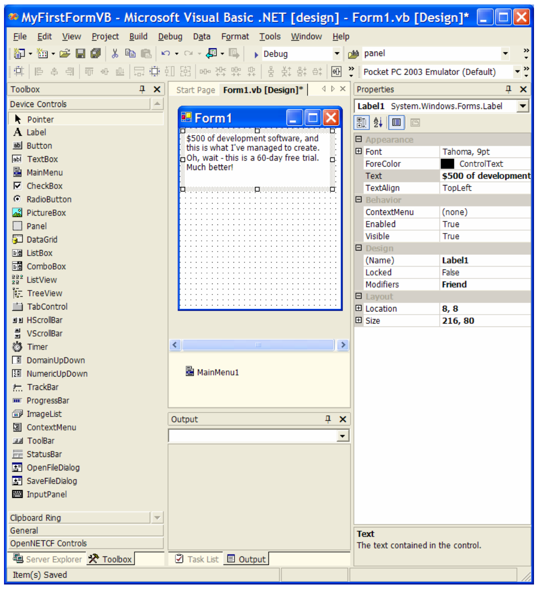
If a Pocket PC application doesn't have a menu class, users can't get to the Software Input Panel (SIP), otherwise known as the on-screen keyboard/block recognizer/letter recognizer. For some applications, where you know the Pocket PC has a keyboard (like my iPaq 4350), or where users only provide input through buttons, this is probably fine, but most of the time you'll want at least the blank menu to give users a bit more freedom in choosing their devices and entering data.

Properties are listed on the right-hand side of the screen. You can select a property and edit its value, and the property list changes depending on which item or items are selected on the form itself. (If the Properties Window or any of the other windows are missing, visit the View menu to bring them back.) For this particular application, all you need to do is put a label on the form and set its text to whatever you like.

That's a little trickier than it probably seems. Unlike sane applications, say Microsoft's own Access, Visual Studio demands that you use the properties to change the content of the label. Don't double-click on it, or you'll find yourself in a code window that lets you adjust the code behind the label rather than the text of the label. So put the label on the form, and then look for the Text property in the Properties Window at the right of the screen. Enter whatever value you want in there.

If you double-click on the label, you'll open the code window for the label. If you hit Alt-F-C, thinking that will close the window, you'll be disappointed - Visual Studio will close the project instead! Be cautious when editing forms. Eventually you'll train yourself to Visual Studio's expectations and maybe things will make more sense.

We'll do one more thing for this application, to let it explicitly quit. Microsoft expects Pocket PC applications to be instantly available to the user once they've started running, so quitting isn't normal behavior for a Pocket PC program. As this application doesn't really have much to offer, it's probably smarter (and easier for debugging) to have it offer a proper quit option rather than hanging around. To do that, click on the form outside of the label, and look for the MinimizeBox property. Set it to false, and then save the project. Your screen should look something like Figure 1-4.



$500 of development software, and this is what I've managed to create. Oh, wait - this is a 60-day free trial. Much better!

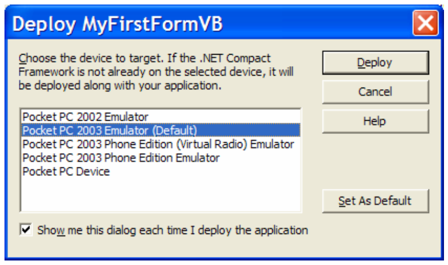
Label1 System.Windows.Forms.Label

Appearance	
Font	Tahoma, 9pt
ForeColor	ControlText
Text	$500 of development
TextAlign	TopLeft
Behavior	
ContextMenu	(none)
Enabled	True
Visible	True
Design	
(Name)	Label1
Locked	False
Modifiers	Friend
Layout	
Location	8, 8
Size	216, 80

Text
The text contained in the control.

Figure 1-4. Visual Studio project after a bit of fiddling.

It isn't glorious, but it's a base to work from. Now it's time to make this work. Press the F5 button. The window at the bottom center will turn to an Output window, showing you the results of your code (well, code Visual Studio created for you) compiling and building. Then you'll get the question shown in Figure 1-5: on what device do you want to see this?



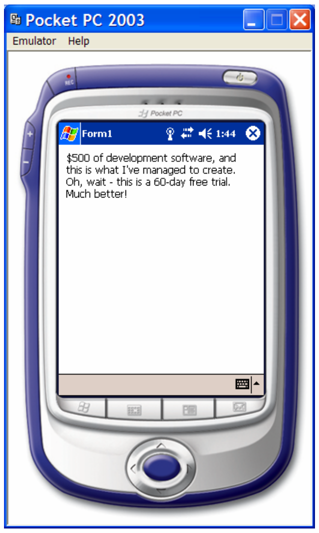
Choose the device to target. If the .NET Compact Framework is not already on the selected device, it will be deployed along with your application.

Pocket PC 2002 Emulator
Pocket PC 2003 Emulator (Default)
Pocket PC 2003 Phone Edition (Virtual Radio) Emulator
Pocket PC 2003 Phone Edition Emulator
Pocket PC Device

☑ Show me this dialog each time I deploy the application

Figure 1-5. Choosing a deployment environment.

If you choose Pocket PC 2003 Emulator and click the Deploy button, you'll see more activity in the Output window, and then you'll see Figure 1-6, or something similar.



$500 of development software, and this is what I've managed to create. Oh, wait - this is a 60-day free trial. Much better!

Figure 1-6. Our super-simple application deployed on the Pocket PC emulator.

Even in this really simple example, there are a few key things to note here. Along the top of the window, we see "Form1", the name of the form. You'll probably want to change this label to something more exciting. The OK button in the top right isn't the usual X, because we set MinimizeBox to false. The X would mean minimize, whereas OK actually dismisses the form. Dismissing the only form of a one-form application conveniently exits the application. The text we put in our label is there, so there's at least some contribution on our part. At the bottom, you can see the little keyboard icon that means that the SIP is available to users, if not particularly necessary for this application.

When you get tired of admiring this simple creation, click the OK button with your mouse. (In general, the mouse pointer behaves in the emulator as your stylus.) The emulator will remain up, showing the Today page, but Visual Studio knows that the fun is over, as shown in this Output Window listing:

```
'MyFirstFormVB.exe': Loaded 'C:\Documents and Settings\Simon\My
Documents\Visual Studio Projects\MyFirstFormVB\bin\Debug\mscorlib.dll',
No symbols loaded.
'MyFirstFormVB.exe': Loaded 'C:\Documents and Settings\Simon\My
Documents\Visual Studio Projects\MyFirstFormVB\bin\Debug\MyFirstFormVB.exe',
Symbols loaded.
'MyFirstFormVB.exe': Loaded 'C:\Documents and Settings\Simon\My
Documents\Visual Studio Projects\MyFirstFormVB\bin\Debug\System.Drawing.dll',
No symbols loaded.
'MyFirstFormVB.exe': Loaded 'C:\Documents and Settings\Simon\My
Documents\Visual Studio Projects\MyFirstFormVB\bin\Debug\System.dll',
No symbols loaded.
'MyFirstFormVB.exe': Loaded 'C:\Documents and Settings\Simon\My
Documents\Visual Studio Projects\MyFirstFormVB\bin\Debug\System.Windows.Forms.dll',
No symbols loaded.
The program '[3668] rundll32.exe: MyFirstFormVB.exe' has exited with code 0 (0x0).
```

For a more exciting time, you can run the same application on your own Pocket PC. Put it in the docking cradle, let ActiveSync do it's thing, and then hit F5 again. Instead of choosing the emulator from the list shown in Figure 1-5, choose Pocket PC Device. The results should be very similar, except that they happen on the device itself. The output window should look the same as it did.

Emulators and Devices

Now that we've used an emulator and a device (assuming you have a device), it's probably worth pausing to look at what exactly is going on here and how reliable you can expect the emulator to be. I'm can't claim to be an old hand at Windows CE development, but my understanding is that the old emulators weren't much fun, and connected calls to regular Windows APIs, so things might work in an emulator but not on a device or vice-versa. Things are better now, but it's still worth considering some of the issues that arise because of developing with emulators and because of variations among devices.

Emulator Issues

Whether you're working with an emulator or with a device, Visual Studio .NET communicates with the program you're running for debugging purposes. From the programmer's perspective, they produce the same results in the output window. There are some differences to keep in mind, however:

- I/O devices will likely be very different. My iPaq 4350 has Bluetooth built into it, while the Pocket PC 2003 emulator doesn't.
- Network connectivity may be different or variable. A Pocket PC using wireless that loses reception as the user walks around will present a very different network experience than you get in an emulator on a PC using reliable Ethernet for its connection.
- Most users won't have keyboards. A text-entry application may feel fine to you in the emulator, but may not be as appreciated by users on Pocket PCs in the field working on tiny keyboards or through the SIP.
- Patterns of use are different. The click-and-hold that the Pocket PC uses instead of a right-click may produce some different behavior as shaky users wobble their stylus

more on a selection than a programmer testing an app moves their mouse. Pocket PCs in the field are likely to get more intermittent use than emulators on a desktop being tested.

In general, the emulator will work well, but you'll want to test your applications on devices in real-life situations in addition to testing them on the emulator.

You can run the emulator without running Visual Studio, if you want to show your friends how snappy the interface is, or programs you've built for the Pocket PC in previous sessions. In a typical installation, the emulator is at `C:\Program Files\Microsoft Visual Studio .NET 2003\CompactFrameworkSDK\ConnectionManager\Bin\emulator.exe`. You can run it with the `/help` option to learn more about the options it supports. You'll need to specify the `/CEImage` option to make it run, and the image files representing different versions of Windows CE are all in the `Images` directory.

Device Variations

While most people think "Pocket PC" these days when Windows CE (sometimes expanded to "Compact Edition," though not by Microsoft, and sometimes disparagingly referred to as "wince") comes up, there are a lot of variations in the Windows CE family. They aren't all Pocket PCs, and there's even some variation among Pocket PCs.

I'm still sad about my IBM WorkPad z50, a tiny ThinkPad-like computer that ran Windows CE 2.11. It weighed 2.6 pounds, had a VGA screen, 48MB of RAM, and could take a 32MB Compact Flash card. With PocketWord, it seemed like the perfect lightweight answer to my note-taking needs, and it even had a VGA output, so I could use it for presentations. I bought it used for $100, and unfortunately it proved to be for sale for a reason: it reset itself periodically, erasing everything on it at inconvenient times. Even when it didn't reset itself, it frequently slowed down to accept about one keystroke per second. Its form factor, the traditional laptop clamshell, hasn't been popular among Windows CE vendors, and sadly IBM hasn't tried again with anything similar.

Today, most Windows CE devices that aren't Pocket PCs are custom devices. Windows CE gives vendors a lot of flexibility to install or not install different parts of the operating system, so it can lurk underneath buttons and an LCD display as well as underneath a VGA display with a touchscreen. A lot of barcode scanners use Windows CE, as do many electronic readout tools, and you can certainly build Windows CE devices into things like cars.

Pocket PC is a specific profile of Windows CE. Vendors have to provide a core set of functionality, as well as a basic set of buttons and a touchscreen. Programs written explicitly for Pocket PC should be able to run on a variety of Pocket PCs, whether they came from Dell, HP, Samsung, or someone else. (Not all Pocket PCs are slim enough to fit in an ordinary pocket. As I was writing this, a Schwann's truck came and delivered some frozen pizza. Their driver uses a big Pocket PC device and a Bluetooth printer to handle inventory, credit cards, and receipts.)

There have been a number of steps in Pocket PC evolution. Only the more recent steps support the .NET Compact Framework:

Handheld PC
: These had bigger screens and often keyboards (like my z50), but the .NET Compact Framework doesn't support these.

Pocket PC 2000
: The original Pocket PC operating system shipped on a number of different microprocessors. (If you wonder why Visual Studio produces so many different output files, this and continuing support for Windows CE more broadly are why.) The .NET Compact Framework does support these devices, though you'll need to install the framework in addition to your application.

Pocket PC 2002
: This version only runs on ARM processors, and is supported by the .NET Compact Framework.

Pocket PC 2002 Phone edition
: The .NET Compact Framework runs on these devices, but without any built-in support for their telephone capabilities.

Pocket PC 2003
: This version, the most common as I'm writing, includes the .NET Compact Framework in ROM, so you don't need to install that to the device. The Second Edition adds VGA-resolution screens and horizontal operation, though I don't think the .NET Compact Framework supports that directly.

Windows Mobile Smartphone 2003
: Cell phones have even fewer interface options than Pocket PCs, as they generally lack touchscreens. To support these devices, Microsoft developed a version of Windows CE that works with their limitations, and the 2003 version of that supports a subset of .NET Compact Framework functionality.

I've seen rumors of more versions that will appear in the near future, but hopefully these devices will continue to run the .NET Compact Framework in some available version for a long time to come. You can also run .NET Compact Framework applications on devices running Windows CE 4.1 or later.

> You should be aware that most vendors don't provide operating system upgrades for Pocket PCs. These devices seem to be regarded as disposable, and somehow the notion of consumers shelling out additional money on a regular basis appeals to the industry. On the other hand, you definitely should visit your vendor's website to see if there are any updates to the Flash ROM of your device. Updating the Flash ROM of my iPaq didn't solve an annoying wireless issue for me, but it did make it a lot more stable.

So what is the .NET Compact Framework?

We've now created an application that uses it, and looked over a list of devices that support it, but what is this thing?

The .NET Compact Framework is closely related to Microsoft's .NET Framework, which includes a number of components:

VB.NET, C# and some other languages

.NET is designed to support a number of languages which produce *managed code*, and to connect them to code that isn't managed. As part of this process, Microsoft developed both some new languages and extensions for older languages. In the .NET Compact Framework, you'll only use VB.NET or C# in your code (and you can mix them if you like), though you can connect to DLLs written in other languages.

Common Language Specification and Common Type System

These aspects of .NET make it possible for different languages to work with the same objects and with each other.

Common Intermediate Language (IL)

Instead of being compiled to machine code, .NET programs are compiled to bytecodes. This sounds like a nuisance, but it has portability and security advantages.

Common Language Runtime (CLR)

The CLR is the environment which runs the programs, compiling the bytecode to machine code, and managing objects and memory allocation and deallocation. If you're a Java programmer, you can think of this as similar to the Java Virtual Machine (JVM).

Common Language Infrastructure (CLI)

This is a set of libraries which Microsoft included in its ECMA standardization process, and which can be used to create a .NET-compatible environment. It does not, however, include all of the .NET Framework or the .NET Compact Framework.

> Because Microsoft chose to go through a standards body (http://www.ecma-international.org/) in creating .NET, other people can create .NET-compatible environments. Microsoft created Rotor (http://msdn.microsoft.com/net/sscli/), an implementation for FreeBSD and Mac OS X. Outside of Microsoft, Ximian, now part of Novell, created Mono (http://www.mono-project.com/), which runs on Linux and Windows.

The .NET Compact Framework builds on these foundations, but its focus on smaller devices meant a fair amount of rewriting from the ground up, especially in the supporting libraries. Microsoft chopped out a huge amount of supporting code, trimming 25MB of the regular framework into 2MB in the compact version. Along the way, many properties and methods disappeared, though there are still some annoying echoes left in the API, properties and calls which still exist but don't actually do anything on a number of objects. You'll also find you have to manage a lot of interface issues yourself rather than expecting them to be taken care of for you, and that a lot of interface components do less than you'd expect of their desktop counterparts.

Microsoft also rewrote a lot of functionality so that it makes more sense in an environment driven by a touchscreen. Using a blank menu to expose the SIP, as we did in the example above, is one example of this, as is the use of the MinimizeButton property. To better integrate Pocket PCs with their SQL Server systems, they also wrote a local database system that can work while the Pocket PC is disconnected from a network and then resychronize when it comes back to the network.

If you're feeling brave, or just prefer C and C++, you can write applications for the Pocket PC using those languages and the Win32 API. You'll be in direct contact with the pulsing heart of the machine, and won't have to hope that Microsoft wrote .NET-specific support for what you want to accomplish. Even .NET Compact Framework developers may occasionally have to come in contact with the Win32 API, though in general you'll create (or find, or buy) wrappers to let you stay in .NET as much as possible.

Looking at the code

But there wasn't any code in the example, right? Well, not quite. Visual Studio .NET wrote a fair amount of code to make that work. Some was code that arrived by default to put that form up on the screen, and some was connected to the label on the form. The Designer part of Visual Studio lets you write code - simple code - by dragging and dropping pieces from the Toolbox and setting their properties. To see this code, right-click on the form, and select View Code. All of the 'real' code is hidden, but if you click the plus sign on the left, you'll see code that looks like Example 1-1.

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Friend WithEvents Label1 As System.Windows.Forms.Label
    Friend WithEvents MainMenu1 As System.Windows.Forms.MainMenu

#Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

    End Sub

    'Form overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        MyBase.Dispose(disposing)
    End Sub

    'NOTE: The following procedure is required by the Windows Form Designer
    'It can be modified using the Windows Form Designer.
    'Do not modify it using the code editor.
    Private Sub InitializeComponent()
        Me.MainMenu1 = New System.Windows.Forms.MainMenu
        Me.Label1 = New System.Windows.Forms.Label
        '
        'Label1
        '
        Me.Label1.Location = New System.Drawing.Point(8, 8)
        Me.Label1.Size = New System.Drawing.Size(216, 80)
        Me.Label1.Text = "$500 of development software, and this " & _
        "is what I've managed to create.  Oh, wait " & _
        "- this is a 60-day free trial.  Much better!"
        '
        'Form1
        '
        Me.Controls.Add(Me.Label1)
        Me.Menu = Me.MainMenu1
        Me.Text = "Form1"
    End Sub

#End Region

End Class
```

Example 1-1. The code Visual Studio created to display a form with a label.

This code creates three objects of three different classes. The first, `Form1`, inherits from the `Form` class, fully identified as a `System.Windows.Forms.Form` class. Inheritance means that the `Form1` class has all of the functionality of `System.Windows.Forms.Form`, but also has some functionality of its own. In this case, that functionality adds a `Label` and a `MainMenu` to the `Form` when the `Form` is created.

The `Label` and the `MainMenu` are declared near the top of the code for the `Form1` class:

```
    Friend WithEvents Label1 As System.Windows.Forms.Label
    Friend WithEvents MainMenu1 As System.Windows.Forms.MainMenu
```

The `Label1` object will be of the `System.Windows.Forms.Label` class, and the `MainMenu1` object will be of the `System.Windows.Forms.MainMenu` class. We're not doing anything complicated enough with these objects for it to be worth creating a new class that inherits from those objects - their default behaviors are fine. Our program just needs to get those objects and set some properties on them. The `Friend` part means that these objects are available to any other program in the same assembly (something we'll get to later), and `WithEvents` means that these objects can raise events. We'll be able to have users interact with them and process those interactions from the form.

The next line is a warning that code placed here is subject to change by the Windows Form Designer, the interface we used to lay out the form's controls:

```
#Region " Windows Form Designer generated code "
```

Everything from this line to:

```
#End Region
```

may get erased if you make changes in the Designer. There's one spot in the next subroutine where you can safely make changes, however.

```
    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

    End Sub
```

The first line here, `MyBase.New()`, calls the `New()` method of the base class for the form, which is `System.Windows.Forms.Form` in this case. The `New()` method will set up a blank form that this class can use as its foundation. (The `MyBase` keyword always refers to the class from which this class inherits.) The next call is to `InitializeComponent()`, which we'll see in a minute. Don't make any changes here until you get past the next comment. (Lines starting with a single quote in VB.NET are comments, and don't get compiled.)

```
        'Add any initialization after the InitializeComponent() call
```

If your form needs initialization separate from the laying out of controls that the Designer will do for you automatically, you can add your code after this comment and the Designer won't obliterate it. We'll do this later.

In addition to the initialization code, there is also some code for cleaning up at the end of the form's lifetime:

```
    'Form overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        MyBase.Dispose(disposing)
    End Sub
```

This method is Protected, so that only other code which inherits from this class has access to it. The Overloads keyword marks it as providing a variant using different argument types than another function with the same name. It's not clear to me why that is necessary here, as it calls the same function of the parent class with the same type of argument. The Overrides keyword marks it as replacing code in its parent class, though since all this method does is call the same method for its parent class, it's not clear why this keyword needs to be here at all. Deleting it, however, makes Visual Studio complain during the build:

```
sub 'Dispose' cannot be declared 'Overrides' because it does not
override a sub in a base class.
sub 'Dispose' shadows an overloadable member declared in the base
class 'Control'.  If you want to overload the base method, this
method must be declared 'Overloads'.
```

You can insert code above the call to `MyBase.Dispose (disposing)` without the Designer replacing it, though in my tests (with `MsgBox`, admittedly) it didn't seem to do anything when I ran it.

The next method contains lots of simple but interesting code, but it's code you should read and never change. This code is completely generated by the Designer, and reflects controls that have been placed on the form and their properties. The warning at the beginning is real; you should treat this as read-only code.

```
    'NOTE: The following procedure is required by the Windows Form Designer
    'It can be modified using the Windows Form Designer.
    'Do not modify it using the code editor.
    Private Sub InitializeComponent()
        Me.MainMenu1 = New System.Windows.Forms.MainMenu
        Me.Label1 = New System.Windows.Forms.Label
        '
        'Label1
        '
        Me.Label1.Location = New System.Drawing.Point(8, 8)
        Me.Label1.Size = New System.Drawing.Size(216, 80)
        Me.Label1.Text = "$500 of development software, and this " & _
        "is what I've managed to create.  Oh, wait " & _
        "- this is a 60-day free trial.  Much better!"
        '
        'Form1
        '
        Me.Controls.Add(Me.Label1)
        Me.Menu = Me.MainMenu1
        Me.Text = "Form1"

    End Sub
```

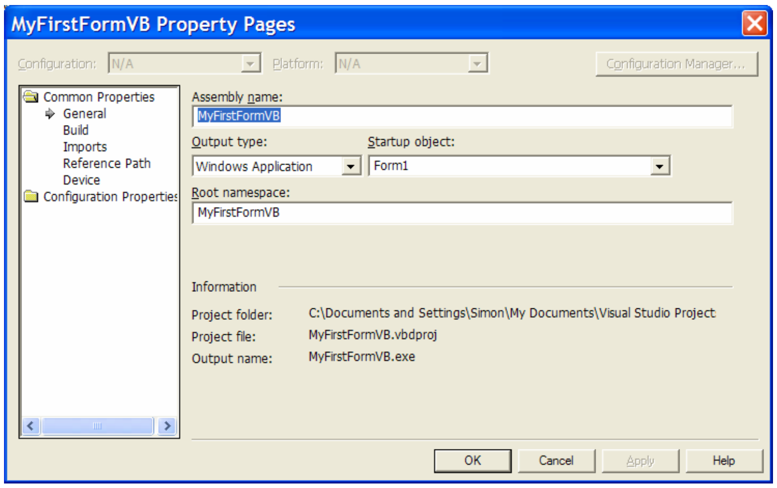
It creates a menu control, `MainMenu1`, which it leaves empty. It creates a label control, `Label1`, assigns it a location and a size, and then provides text. Once it's configured, this routine adds the label control to its collection of controls. Then it adds the menu control, which even though it's empty will result in the SIP being available for input. Finally, it sets the name of the form, which will appear in the top left of the screen, to "Form1".

You should refrain from changing any of the code here, as it will just get overwritten, but there's still plenty to learn here. If you want to create forms by hand, these are the parts you'll need, and you can also create things like panels which contain controls using the same code - but not necessarily the Designer.

How This Application Runs

All of that code is interesting, but there's no actual entry point. How does the Pocket PC or emulator know where to begin in the code?

In C#, programs include an explicit Main() method that fires up the program. In VB.NET, the starting form for the program is set in the properties for the project, which you can find in the Project menu at its very bottom. The Properties window, shown in Figure 1-7, will appear.



MyFirstFormVB Property Pages

Configuration: N/A
Platform: N/A

Assembly name: MyFirstFormVB
Output type: Windows Application
Startup object: Form1
Root namespace: MyFirstFormVB

Information
Project folder: C:\Documents and Settings\Simon\My Documents\Visual Studio Project
Project file: MyFirstFormVB.vbdproj
Output name: MyFirstFormVB.exe

Figure 1-7. The Properties window for a VB.NET Compact Framework Project.

The startup object, to the right side of the middle row of form fields, is where your program will start. If you want to use a different form to start, or even if you just change the name of your form, you'll need to change the Startup object here. If you don't, you'll get build errors like:

```
'Sub Main' was not found in 'MyFirstFormVB.Form1'.
```

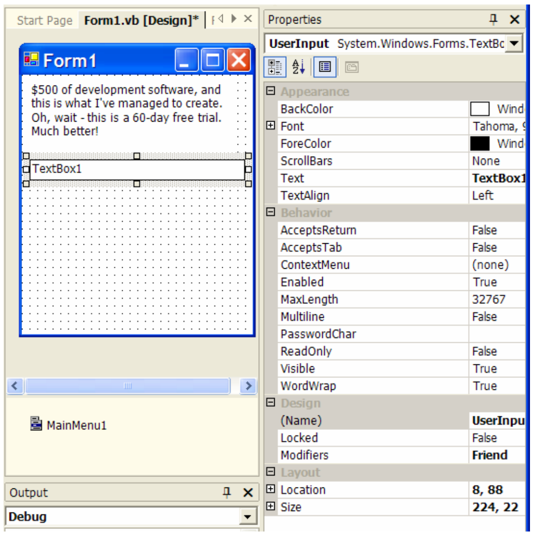
When the program ran above, the compiled code included a reference to Form1. The .NET environment created a new instance of Form1, which displayed the form on the screen. Because there isn't much you can do with this app, it waited until the user clicked on the X in the top right corner and then closed the form. Closing the base form for the program will exit the program completely.

.NET Compact Frameworks must have a base form and other forms. It's a change from ordinary Windows (or other GUI) applications, where you might open the program, create a new window for a different document, and close the original window while still working in the new window. This requirement (and the .NET Compact Framework's limited built-in form management capabilities) will require you to structure and manage your applications carefully.

Adding Some Action

We've gone a long way in doing very little, but it's time to take the next big step and give this application some interactive capabilities beyond presenting a form and waiting for the user to close the form. Our first step will still be pretty small, but it will give you a basic idea of how to process events and respond to them in the .NET Compact Framework environment. We'll add a `TextBox` for user input and two buttons. One button will display the user input in a message box, and the other will change the content of the label to match what the user put into the `TextBox`. (It's not very exciting, I know, but starting small has its own virtues.)

Drag a `TextBox` from the Toolbox and put it on the form so it looks roughly like Figure 1-8. Change its name property from `TextBox1` to `UserInput`, and make it stretch across the form. You can change other properties if you want to make it look more interesting, but the name is the important part for the purposes of the code. The `TextBox` will give users a place to enter information, but the actual processing of that information will be handled by code in the buttons.



$500 of development software, and this is what I've managed to create. Oh, wait - this is a 60-day free trial. Much better!

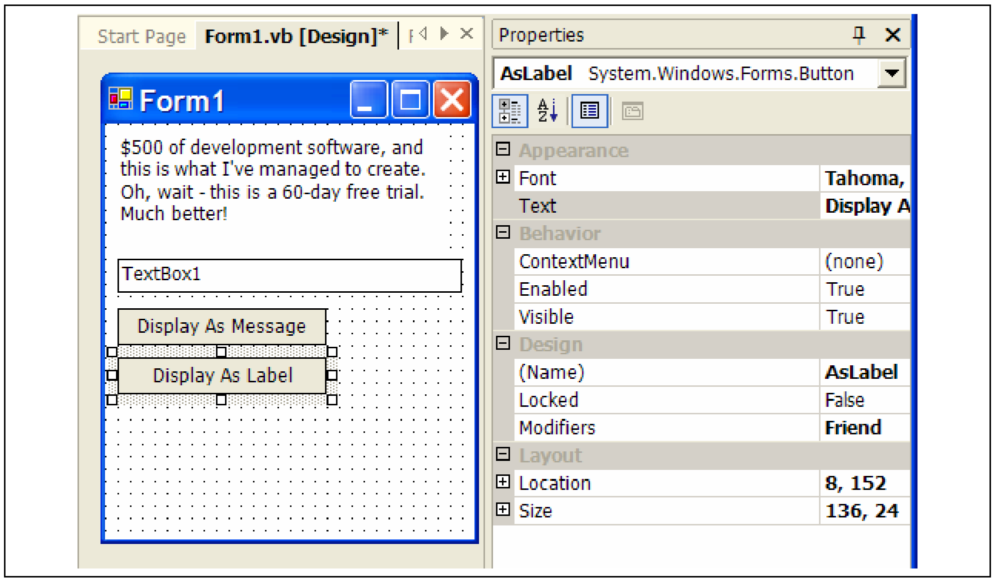
TextBox1

UserInput System.Windows.Forms.TextBc

Appearance	
BackColor	⬜ Wind
Font	Tahoma, 9
ForeColor	⬛ Wind
ScrollBars	None
Text	**TextBox1**
TextAlign	Left
Behavior	
AcceptsReturn	False
AcceptsTab	False
ContextMenu	(none)
Enabled	True
MaxLength	32767
Multiline	False
PasswordChar	
ReadOnly	False
Visible	True
WordWrap	True
Design	
(Name)	**UserInpu**
Locked	False
Modifiers	**Friend**
Layout	
Location	**8, 88**
Size	**224, 22**

Figure 1-8. Form with TextBox added.

To add the first button, drag a button from the toolbar to the form and change its Label property to "Display as Message" (you'll probably want to widen the button) and its Name property to AsMessage. Then drag a second button to the form and change its Label property to "Display as Label" and its Name property to AsLabel. You should end up with something that looks like Figure 1-9.


Start Page
Form1.vb [Design]*
Form1
$500 of development software, and this is what I've managed to create. Oh, wait - this is a 60-day free trial. Much better!
TextBox1
Display As Message
Display As Label
Properties
AsLabel System.Windows.Forms.Button

Appearance	
Font	Tahoma,
Text	Display A
Behavior	
ContextMenu	(none)
Enabled	True
Visible	True
Design	
(Name)	AsLabel
Locked	False
Modifiers	Friend
Layout	
Location	8, 152
Size	136, 24

Figure 1-9. Form with buttons.

Next we'll add some logic to those buttons so that they actually do something. Double click on the "Display As Message" button to bring up its code window. You'll find yourself in the code for the Form which we saw above, with the cursor in the middle of this new subroutine at the end:

```
Private Sub AsMessage_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles AsMessage.Click

End Sub
```

Visual Studio .NET has written the framework your event handler is required to have. For processing a click you're not going to worry about the sender (where the event came from) or the e value with its additional arguments about the event, but you'll need those in later more complex event processing. It also notes that this bit of code Handles AsMessage.Click, which tells the .NET Compact Framework that this code should get called whenever a user clicks on the button named AsMessage.Click.

If you look through the rest of the code in the Form1 class, you'll also find declarations and initializations for the components you just added. The only thing you need to change to make the button do work, however, is in the events.

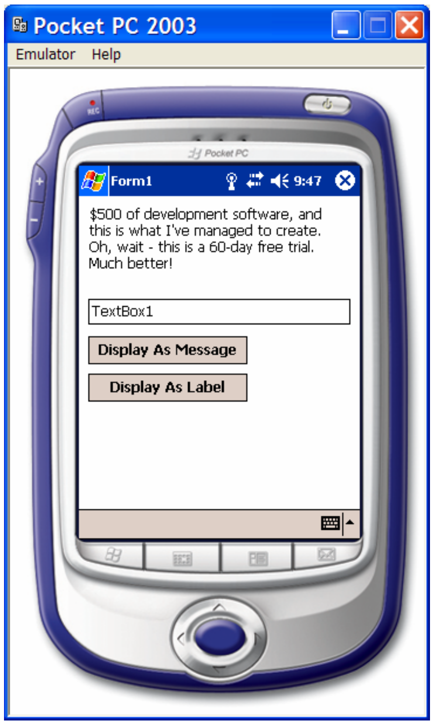
This button should display a message box containing the contents of the UserInput text field when the user clicks on it. Inserting this code into that AsMessage_Click subroutine will do just that:

```
MsgBox("You typed: " + UserInput.Text)
```

For the AsLabel button, we'll do exactly the same thing except that instead of using a message box we'll change the Text property of Label1:

```
Label1.Text = UserInput.Text
```

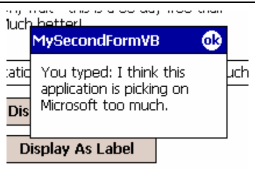
Now we have an application which will actually let the user do something, though that something is still pretty meaningless. Fire up the application and debugging by pressing F5 and selecting the Pocket PC 2003 emulator. You'll see something like Figure 1-10.



$500 of development software, and this is what I've managed to create. Oh, wait - this is a 60-day free trial. Much better!

Figure 1-10. The Pocket PC application's initial state.

It's not precisely beautiful, but we'll get there. For now, what matters is that you can click on the text box under the label and type whatever you like into it. When you're content with what you've typed, you can click one of the buttons and have your typing reflected in either a message box or in a change to the text of the label. (It's a bit unusual to have user input change the interface itself, but the technique may be useful in other contexts as well. For example, if you type "I think this application is picking on Microsoft too much." and click on the Display As Message button, you'll see a message box like that in Figure 1-11.

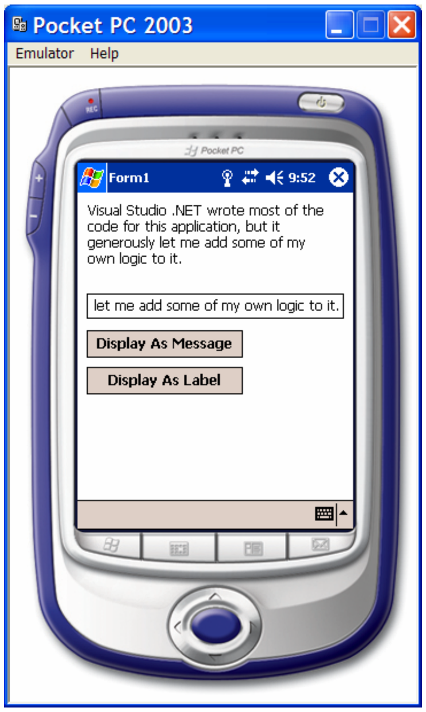


You typed: I think this application is picking on Microsoft too much.

Figure 1-11. A message box displayed by the application.

To dismiss the message box, click on the ok button in its top right corner. For our next demonstration, we'll change the label to something more Microsoft-friendly. Type

"Visual Studio .NET wrote most of the code for this application, but it generously let me add some of my own logic to it." and then click on the Display As Label button. The application will then convey a very different attitude, as shown in Figure 1-12.



Visual Studio .NET wrote most of the code for this application, but it generously let me add some of my own logic to it.

let me add some of my own logic to it.

Figure 1-12. A changed label with a changed attitude.

Undoubtedly you'll want to do a lot more going forward, but this is a start. We've built a simple interface, wired up a few events, and created an application which lets the user do something, if not very much.

If you ever screw up the windows in Visual Studio .NET and don't think you'll ever get it back to sane, go to the Options... item on the Tools menu and click "Reset Window Layout".