SYBEX Tutorial

Mastering™ Visual Basic® .NET

Evangelos Petroutsos

Tutorial 1: Writing Multithreaded Applications with VB.NET

ISBN: 0-7821-2877-7

Sybex Inc.
1151 Marina Village Parkway
Alameda, CA 94501
U.S.A.
Phone: 510-523-8233
www.sybex.com

Tutorial 1

Writing Multithreaded Applications with VB.NET

YOU MAY HAVE NOTICED that one of the most advertised and talked about new features of Visual Basic is the ability to write multithreaded applications. An application with multiple threads is more responsive than a nonthreaded application, but it's also harder to debug. This tutorial is an overview of the techniques for creating a multithreaded application with Visual Basic .NET, and it includes some simple and practical examples that you can reuse in your own projects.

So, what's a thread and when should you run multiple threads in parallel? A thread is one of those things that is easy to understand with an example, yet difficult to explain. Each application runs on its own memory, has its own set of local variables, and is totally independent of any other application that may be running on the same computer at the same time. You can run two applications that are actually two instances of the same application, because each application is running in its own thread. For example, you can start Visual Studio twice and work on two different projects; you can even run both projects at the same time. You can't, however, open the same project in both instances of Visual Studio—in general, you can't open the same document twice. A thread, therefore, is a series of statements that are executed serially. When a statement is executed, it means that all previous statements have already been executed and the intermediate results they produced are available.

A multithreaded application tells the CPU to execute a subroutine (a series of statements) in a separate thread. The main application invokes a new thread and continues with the following statements, while the subroutine that runs in another thread is executing in the background. When the background thread is done, it notifies the main application (or the thread that initiated it) that it's done and the results are available.

So here's the first complication in writing multithreaded applications. You can't assume that the results of a procedure running on a separate thread are available when you need them. When a single-threaded application calls a procedure that takes a while to execute, the visible interface of the application simply freezes for a few (or not so few) seconds. The application will continue only when the procedure returns and the results are available. If you allow some operations to be executed on a separate thread, you must make sure that this thread has completed its calculations before you attempt to access the results.

Each application running on the PC starts its own thread, and some operations can even be carried out in parallel. For example, you can read a file while saving another. Most often, you would write code that saves the data to a file, then load another file, and finally continue with other statements. There's nothing wrong with this approach. Moving a few megabytes from memory to disk (or in the opposite direction) takes a few seconds, and you can usually afford to wait for the application to complete each task.

Say, for example, that the files are extremely large and that you must process them as you save them. Also, you want to make the application more responsive. To do this, you can use a separate thread for each operation. Each thread will perform its task, but both appear to be running in parallel. Not only that, but the application can also react to events as usual. It may not instantly process the click of a button, but at least it won't freeze. This approach is represented by the following pseudocode:

```
' statements
SaveFile(…)
OpenFile(…)
' more statements
```

The SaveFile() statement starts a thread that saves a file, and the OpenFile() statement starts another thread that opens a file. These are subroutine names and you will see shortly how to create and start a thread. Visual Basic starts a new thread and continues with the following statement. The main application's code (the main thread, as it's called) continues its execution. The other two threads spawned from within the main thread continue to execute as well. The operating allocates time slices to each thread and you have some control over the process. You're not in control of when the actual switching takes place, neither can you specify when a thread will give up the CPU to let another thread execute. You can, however, determine the priority at which every thread will execute.

If you want to use a separate thread for a task, then you must first create two threads—one for saving the data and another one for loading the data from a file. The threads are objects that contain the same statements as the two subroutines SaveFile() and OpenFile(). Assuming you've implemented the SaveFileThread and OpenFileThread objects, you can start two threads by calling their Start method:

```
' statements
SaveFileThread.Start
OpenFileThread.Start
' more statements
```

The first statement will start the execution of the code that saves the file, but it won't wait for the execution to complete. It will also start the execution of the code that loads the data from a file and then proceed with the following statements. The two threads will complete their tasks and then they'll "die."

You can also raise events from within the two threads to notify the application about unusual conditions or about the successful termination of their execution. They run in parallel and don't freeze your application. Here's a very simple example. You may be familiar with the MsgBox() function, which displays a message box on the screen. When the message box appears, the application freezes. It doesn't respond to any events, and you can't switch to the open form. You're essentially trapped in the message box, and you can't do anything unless you click one of the buttons on the message box to get it out of the way.

TIP *For best results with image processing, avoid trying to run another menu command while one is still processing.*

Trivial Multithreading

Enter the world of multithreaded applications. If you call the MsgBox() function as a separate thread, it will execute in its own thread. It won't lock your application, and you'll still be able to interact with the application. The message box will remain open, but it won't interfere with you, at least not in such an intrusive manner. You can even open another instance of the same message box! Potentially, the most bizarre behavior is that you could close the application but the message box remains visible. Actually, there's no simple way to kill its thread (you can kill most threads, but this one is an exception, as you will see shortly).

Of course, what good is a message box that doesn't grab your attention? The idea is to force users to notice the message (perhaps even read it) and then click the appropriate button. We'll come back to this example, but first let's have an overview of the mechanics of creating and starting a thread.

Creating a Thread

A thread is a task, similar to a subroutine. More specifically, it's a subroutine that will execute in the background. The example used in this section is the Threads1 project in the tutorial's Zip file. It is a very simple project that demonstrates what threads can do for your application as well as how to set up and start a thread. The following is the simplest VB subroutine that I can imagine:

```
Private Sub BackgroundProcess()
    MsgBox("Click OK to close me, or interact with the main form")
End Sub
```

I've named this subroutine BackgroundProcess() because it will execute in the background, in its own thread. To execute some code in a separate thread, you must first create a subroutine with the corresponding code. This subroutine can't accept any arguments.

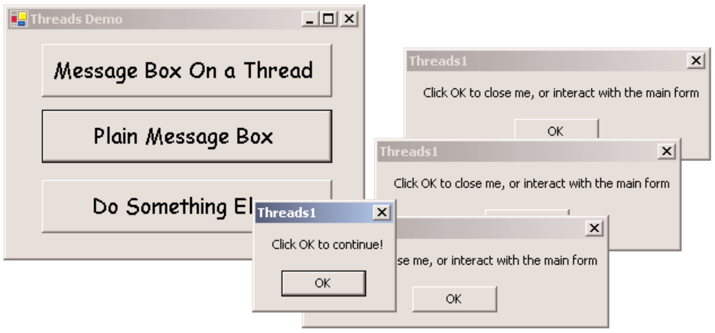
Next you must create a thread in which the BackgroundProcess() subroutine will execute. First, import the namespace `System.Threading` by inserting the following statement at the top of the file:

```
Imports System.Threading
```

To create a new thread, you must instantiate a variable of the Thread type and supply a delegate for the subroutine that you want to execute in the thread. Once the thread is set up and initialized, you can invoke it by calling its Start method. The following statements will start the execution of the BackgroundProcess() subroutine and will immediately return the control to the application:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                      ByVal e As System.EventArgs) Handles Button1.Click
    Dim t As Thread
    t = New Thread(AddressOf Me.BackgroundProcess)
    t.Start()
End Sub
```

If you run this project now, you will see the message box displayed by the BackgroundProcess() subroutine. You can switch to the form of the application and again click the same button that invokes the message box to see an identical message box, as shown in Figure 1.1. You can display any number of message boxes. You can even close the application's form by clicking Close at the top-right corner of the form, and the message boxes will remain in their places on the screen! You will also notice that the project remains in Run mode: you must either close all instances of the BackgroundProcess() subroutine, which are the open message boxes, or click the Stop Debugging button to terminate the application.



Click OK to close me, or interact with the main form

Click OK to continue!

FIGURE 1.1
The message boxes displayed by the Threads1 project are not modal.

Let's add some code to the other two buttons on the form. The Plain Message Box button displays a regular (modal) message box (see Figure 1.1), while the other button prints the current date and time in the Output window. Their code is quite trivial (no threads involved here):

```
Private Sub Button2_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) Handles Button2.Click
    Console.WriteLine(Now)
End Sub

Private Sub Button3_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) Handles Button3.Click
    MsgBox("Click OK to continue!")
End Sub
```

Every time you click the Do Something Else button, the current date is printed in the Output window. This will happen even if a message box is open at the time. If you click the last button, Plain Message Box, the application's main form will freeze. You must close this message box to interact with the form again.

In addition to starting a thread, you can also control it from within your application. The basic methods of the Thread object include the following:

- The Start method begins the execution of the thread.
- The Abort method terminates the execution of a thread. In some cases, calling the Abort method may not actually terminate the thread.

- The Suspend method suspends the execution of a thread (if it's running).
- The Resume method resumes the execution of a suspended thread.
- The Sleep method pauses the execution of the thread for a specified number of milliseconds.
- The Join method blocks the calling procedure either for a specified number of milliseconds or until the thread terminates. Unlike the other methods, this one affects the code that called the thread, not the thread itself.

Using Threads

Don't feel compelled to implement your applications around threads just because it's now possible to write multithreaded applications with Visual Basic .NET. Threads will come in very handy in a few situations, but the majority of applications won't benefit from threads. Multiple threads of execution will complicate the debugging process, not to mention that you may discover that the application has become unacceptably slow because of the overhead introduced by multiple threads. Being able to run several processes in parallel on a single CPU entails some overhead, after all, and this overhead can become substantial.

Switching the CPU from one thread to another is a complicated process; it's handled by the operating system and is called *context switching*. If you have too many threads running at once, the CPU will end up spending too much time switching between their contexts.

An application that animates several shapes (bouncing balls, for example) is a good candidate for a threaded implementation. Let's consider for a moment an application that must animate several balls bouncing on a form and then bouncing off the edges of the form. This application requires that you update the positions of all balls from within a Timer's event, or an endless loop. If you must detect other conditions, such as a ball hitting another or a ball hitting a paddle, the code in this loop can get lengthy and complicated.

The alternative is to use a separate thread for each ball. Once started, the thread knows how to move the ball around, bounce it off the edges of the form, and so on. We'll implement a simple application along these lines in the following section.

Another good use of threads is to implement tight loops that can be interrupted. You'll understand why if you have you ever run into an application that can't quit when you click Cancel because it's too busy to react, or if you have ever had to call the DoEvents method a little too often in a loop that takes too long to execute. (Every other VB6 programmer certainly has!) Consider, for example, an image-processing application. Processing an image means performing a number of calculations for every pixel, and there are hundreds of thousands of pixels in an image. You should provide a method for users to abort the execution of an operation, like a Cancel button. This button should react to the Click event instantly, but this isn't always feasible. The button will react to the Click event only when the DoEvents method is called. (The DoEvents method gives other parts of the application a chance to process their events.) If, however, you insert too many calls to the DoEvents method in your code, then the application's performance suffers because the DoEvents method involves a substantial overhead.

Tip *You can explore the ImageProcessing application from Chapter 14 of* Mastering Visual Basic .NET *to experiment with image-processing techniques. You can insert additional (even unnecessary) calls to the DoEvents method to see how this method affects the overall performance of the application.*

To write applications that involve heavy calculations while still maintaining the flexibility (and luxury) to respond to user events, you can implement computationally intensive segments of code as separate threads. When the user requests the abortion of the current operation, you can call the appropriate thread's Abort method. We'll revise the ImageProcessing application shortly, but first let's look at a simpler application that demonstrates the methods of the Thread object.

Fast-Loading Forms

Do the initialization tasks in some of your applications take too long? Or, do specific forms have to execute an unusually large number of statements that delay the loading of the form? If these scenarios sound familiar, here's an interesting technique. Instead of placing all of the initialization code in the form's Load event handler, you can create a new thread to execute the initialization code. The result is that the form will appear quickly and the application will respond to events even though some code is executing in the background. Not all of the data will be available at the time, but users will be able to interact with the existing data. To demonstrate how this technique works, we'll build an application that must load a million items into a ListBox control in its main form's Load event handler. Of course, a million items are too many for a ListBox control, but you may have to retrieve a substantial number of rows from a database, process them, and then display the results on a control.

We'll call the project for this section FormLoad. This is the code that must be executed in the form's Load event:

```
    Private Sub BackgroundProcess()
        Dim i As Integer = 1
        For i = 1 To 100000
            ListBox1.Items.Add("Item: " + i.ToString)
        Next
        MsgBox("done!")
    End Sub
```

Notice that the code loads 100,000 integers to a ListBox control. Also notice that the code resides in the BackgroundProcess() subroutine, which is the name I use for a thread code. Once the form loads, the thread starts from within the Load event. After that, the form displays as usual and the user can interact with it.

The form's Load event handler is shown next:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
                       ByVal e As System.EventArgs) Handles MyBase.Load
    Dim bgThread As Thread
    bgThread = New Thread(AddressOf Me.BackgroundProcess)
    bgThread.Start()
End Sub
```

The form of the application contains a button, which prints in the Output window the index of the currently selected item on the ListBox control. While the background thread is adding items to the control, you can interact with the control. The process is much smoother than loading the ListBox control from within the Load event's handler, even if you called the DoEvents methods at each iteration. Here's the code behind the button on the form:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) Handles Button1.Click
    Console.WriteLine("ITEM " & ListBox1.SelectedIndex.ToString & _
                      " = " & ListBox1.SelectedItem.ToString)
End Sub
```

If you run this application, you'll see that it works nicely. However, you should never update the controls on a form from within any thread other than the one that created the form. In other words, you should never update your application's interface directly from within a thread you created in your code. All Windows controls expose the Invoke method, which you should use to access a specific method from within a thread. The syntax of the Invoke method is

```
Invoke(delegate, arguments)
```

where *delegate* is a method's name and *arguments* is an array with the arguments you want to pass to the method.

The problem occurs when your thread and Windows attempt to refresh the same control at once. When this happens, you'll get an error message indicating that an object is being used by another process. This won't happen with the ListBox example, because the thread is actually manipulating the items of the ListBox control, not the visible interface of the control. With other controls, you should avoid accessing them directly from within a thread's code. In the last example of this chapter, you'll see how to use the Invoke method to update the bitmap of a PictureBox control.

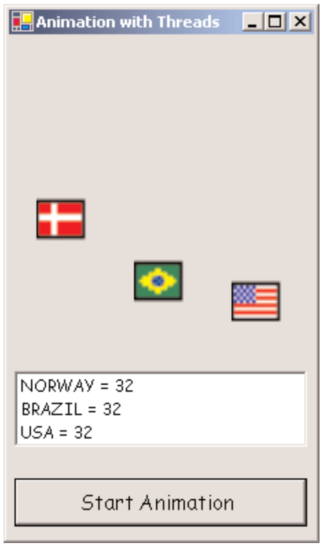
Controlling Threads

The project discussed in this section is the AnimatedFlags project in the tutorial's Zip file. In this project, you will animate three images by bouncing them up and down at different speeds. The images are flag icons displayed on their own PictureBox controls. The animation takes place by changing the coordinates of each control's top-left corner. The code adjusts the flags' location on the form by adding a displacement to the current vertical coordinate of each picture box. This displacement is a random number of pixels (a number between 1 and 5) that is added to the Top property of each PictureBox control. When the flags hit the bottom or the top of the form, the displacement changes, and the flags move in the opposite direction. If you run the project for a while, the flags will bounce up and down. The code doesn't allow them to move all the way to the bottom of the form, because they'll end up covering (or being hidden by) the controls at the bottom of the form. (You can edit the code to bounce them off all four edges of the form, but this will add considerable complexity to the code.)

Each flag is animated from within its own subroutine, which is invoked as a separate thread. Each thread's subroutine is an infinite loop that moves the corresponding PictureBox control to its new location, checks for boundary conditions, and adjusts the displacement accordingly. Then the thread "sleeps" for a number of milliseconds; otherwise, the animation would be too fast.

To start your project, place three PictureBox controls on a new form, set their Image property to a bitmap (I've used some icons that come with Visual Studio), and align them properly on the form. Add a list box and a button on the form, as shown in Figure 1.2, and then switch to the code window.

FIGURE 1.2
The AnimatedFlags project animates three flags. Each animation takes place from within its own thread.



```
NORWAY = 32
BRAZIL = 32
USA = 32
```

First, declare three Thread objects, which will be used to invoke the subroutines that animate each flag (the threads are named after the flags they represent):

```
Dim TNorway As Thread
Dim TBrazil As Thread
Dim TUSA As Thread
```

Next, you must create the animation subroutines. The following subroutine animates the flag of Norway:

```
Sub StartNorway()
    Dim top As Single
    Dim disp As Single
    Dim rnd As New System.Random()
    disp = rnd.Next(1, 5)
    While True
        Console.WriteLine(Thread.CurrentThread.Name)
        top = top + disp / 10
        PBNorway.Top = top
        TNorway.Sleep(1)
        If PBNorway.Top > 200 Then disp = -disp
        If PBNorway.Top < 0 Then disp = -disp
    End While
End Sub
```

In the button's Click event handler, we start the animation by calling each thread's Start method:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                          Handles Button1.Click
    TNorway = New Thread(AddressOf StartNorway)
    TNorway.Name = "Norway"
    TNorway.Priority = ThreadPriority.Lowest
    TNorway.Start()
    TBrazil = New Thread(AddressOf StartBrazil)
    TBrazil.Name = "Brazil"
    TBrazil.Priority = ThreadPriority.Lowest
    TBrazil.Start()
    TUSA = New Thread(AddressOf StartUSA)
    TUSA.Name = "USA"
    TUSA.Priority = ThreadPriority.Lowest
    TUSA.Start()
    Timer1.Enabled = True
End Sub
```

A Timer control on the application's form fires a tick event every 5 milliseconds, and in this event's handler we display the state of each thread on a ListBox control at the bottom of the form. The Timer's Tick event handler is shown next:

```
Private Sub Timer1_Tick(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) _
                        Handles Timer1.Tick
    ListBox1.BeginUpdate()
    ListBox1.Items.Clear()
    ListBox1.Items.Add("NORWAY = " & TNorway.ThreadState.ToString("G"))
    ListBox1.Items.Add("BRAZIL = " & TBrazil.ThreadState.ToString("G"))
    ListBox1.Items.Add("USA = " & TUSA.ThreadState.ToString("G"))
    ListBox1.EndUpdate()
End Sub
```

The status of all threads as it appears on the ListBox is "WaitSleepJoin." When the Tick event takes place, only one of the threads is running and the string "Running" appears on the ListBox control for a very brief moment. You will notice the flickering and be able to see the string "Running," but most of the time the status of all three threads will be "WaitSleepJoin."

In the Click and DoubleClick events of the PictureBox controls with the flags, we can test the basic methods of the Thread class. When a flag is clicked, the program suspends the animation of the clicked flag by calling the Suspend method of the corresponding Thread object. If a suspended flag is clicked again, the animation resumes with a call to the Resume method. Finally, if a flag is double-clicked, it's aborted by a call to the Abort method. Listing 1.1 shows the code behind the Click and DoubleClick events of the picture box with Brazil's flag.

Listing 1.1: Click and DoubleClick Events for the Brazilian Flag Picture Box

```
Private Sub PBBrazil_Click(ByVal sender As System.Object, _
                           ByVal e As System.EventArgs) _
                           Handles PBBrazil.Click
    While TBrazil.ThreadState = ThreadState.WaitSleepJoin
```

```
        End While
        If TBrazil.ThreadState = ThreadState.Running Then
            TBrazil.Suspend()
        Else
            If Not TBrazil.ThreadState = ThreadState.Stopped Then
                TBrazil.Resume()
            End If
        End If
    End Sub

    Private Sub PBBrazil_DoubleClick(ByVal sender As Object, _
                                     ByVal e As System.EventArgs) _
                                     Handles PBBrazil.DoubleClick
        If Not TBrazil.ThreadState = ThreadState.Aborted Then
            If TBrazil.ThreadState = ThreadState.Suspended Then
                TBrazil.Resume()
                TBrazil.Abort()
            End If
        End If
    End Sub
```

Notice that the code calls the Resume method before calling the Abort method, because if the thread happens to be sleeping (and there's a very good chance that the thread will be in the sleep state at the time), it can't be aborted.

Open the AnimatedFlags project and experiment with threads. Notice that you can't restart a thread after aborting it. In each thread's main loop, a Console.WriteLine statement prints the name of the current thread on the Output window.

Normally, you'd expect the three threads to take turns and see a list like the following in the Output window:

```
Norway
Brazil
USA
Norway
Brazil
USA
Norway
Brazil
USA
```

This is what you'll see most of the time, but you may see a thread taking another thread's turn:

```
Norway
Brazil
USA
Norway
Norway
USA
```

Or even something like this:

```
Norway
Brazil
USA
Norway
Norway
Brazil
```

The order in which threads wake up is not deterministic. If two threads should wake up at the same time, the operating system will decide which one wakes up first. The same is true for the Abort method; the thread won't be terminated as soon as you call its Abort method but sometime after that. According to the documentation, calling this method usually terminates the thread. This is a very bold statement, but clearly under some circumstances you may not be able to abort the execution of a thread. I haven't witnessed this behavior so far in my projects, with the exception of the Message Box discussed at the beginning of this tutorial.

Communicating with Threads

Threads aren't isolated from the calling application. A mechanism must be in place, though, for a thread to pass information to the calling application, or at least to notify the application that it has completed its execution. As you may have noticed in the examples so far, the Start method doesn't accept any arguments, which means that you can't pass any data to a thread when you invoke it. The procedure that you pass to the thread constructor can't have any arguments, or a return value, either.

Because the thread's code resides in the same file as the calling application, you can always use global variables to exchange information among the threads, but this approach can lead to horrible bugs. For example, the threads are running independently of one another, and you can't control when each thread is reading, or setting, a global variable. They could end up reading the global variables at the wrong time.

Passing Arguments to a Thread

A better approach is to implement a class for each thread. In the following example, we'll wrap the procedure in a class and then add fields and/or properties to the class. The calling application can set these fields and then start the execution of the thread. The code of the thread can then read the fields, which are local variables. Each instance of the class will have its own set of variables and will not interfere with the variables of any other instance. This will allow you to set some initial values (sort of passing arguments through the Start method) and read back the results.

The following class provides two public fields and a function that will be executed in a separate thread:

```
Public Class ThreadedCalculator
    Public Value1 As Double
    Public Value2 As Double
    Public Sub Calculate()
        ' the procedure's statements
        ' you can read and set the values of the public fields
        Value2 = <return_value>
    End Sub
End Class
```

The calling thread must set *Value1* to an initial value before starting the thread of the Calculate() subroutine.

Following is the code that will create and start a new thread of execution for the Calculate() subroutine:

```
Dim TCalc As New ThreadedCalculator()
TCalc.Value1 = 99.09
Dim oThread As New Thread(AddressOf TCalc.Calculate)
OThread.Start()
```

After the execution of the thread, *Value2* can be read to retrieve the result of the calculations. But how do we know that a thread has completed its execution? The safest approach is to raise an event from within the thread's code, just prior to its termination. To do so, you must add the event declaration in the class and raise the event from within the Calculate() subroutine, as shown here:

```
    Public Event ThreadComplete(ByVal result As Double)
    Public Sub Calculate()
        ' the procedure's statements
        ' read or set the values of the public fields
        RaiseEvent ThreadComplete(<return_value>)
    End Sub
```

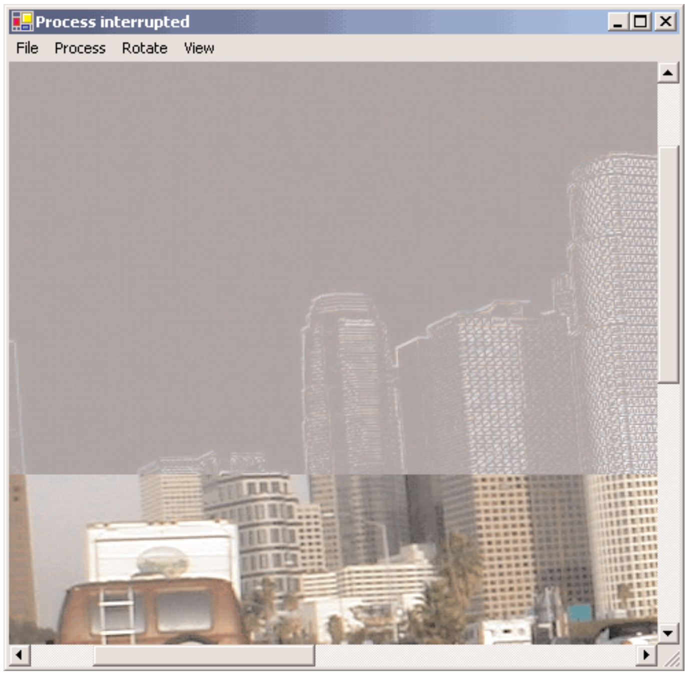
Notice that you no longer need to set any of the class's fields; instead, you can pass the appropriate values to the calling application through the event's argument list. The calling application must declare the *TCalc* variable with the WithEvents keyword and provide a handler for this event:

```
Dim WithEvents TCalc As ThreadedCalculator

Sub ThreadHandler(ByVal result As Double) _
        Handles TCalc.ThreadComplete
    Console.WriteLine("The thread returned the value " & result.ToString)
End Sub
```

Let's look at a more advanced example that demonstrates some of the practical aspects of multithreading. Chapter 14 of *Mastering VB.NET* discusses an image-processing application in which you can load an image and then select one of the image-processing algorithms from the Process menu to process the image. In the following section, we'll revise the ImageProcessing application so that the processing of the image takes place in a separate thread (see Figure 1.3 for an example). The main form will be quite responsive, although there's not much you can do from the main form while an image is being processed, short of stopping the current processing or resizing the image being processed.

FIGURE 1.3
A partially embossed image



Revising the ImageProcessing Application

The application's form has the usual File menu with commands to load an image (and to save the processed image back to a file), the Rotate and View menus (with commands to zoom in and out, rotate and flip the image), and the Process menu, which contains the image-processing commands. The following image-processing algorithms are implemented: Emboss, Sharpen, Smooth, and Diffuse. These algorithms are explained in detail in Chapter 14 of *Mastering VB.NET*, but here's a quick outline of an image-processing algorithm for those of you who haven't read the chapter.

Each image-processing technique is implemented by a separate subroutine, which goes through all the pixels in the image, transforms them, and stores their new values to another image. The transformation involves the values of the neighboring pixels, which means that we can't overwrite the pixels of the original image after transforming them. That's why we use another Image object to store the processed pixel values. When we're done, we display the new image.

The transformation of each pixel's value involves basic arithmetic operations. To smooth the image, for example, we take the average of the pixels surrounding the current pixel. The wider the area over which we calculate the average, the smoother the image will be. To smooth an image, we scan each pixel (pixel after pixel in the same row, one row at a time) and calculate the average of a square (3 × 3 or 5 × 5) centered over the current image. This is the value of the pixel at the same location in the smoothed image, and it's stored to another Image object (we don't want the smoothed pixels to influence the calculations of their neighboring pixels). If you calculate the average over a much larger square, you'll remove all the details (this technique is frequently used in TV to hide someone's face).

All of the image-processing algorithms in the ImageProcessing application scan the pixels of the original image with two nested loops, transform the current pixel, and store the transformed value to another image. Instead of waiting for the image-processing algorithm to complete before displaying the transformed image, we copy the original image and update the display so that users can watch the transformation of the image as it happens. Updating the display is a rather costly operation in terms of CPU cycles, so we update the display after transforming a number of rows of pixels. You can also update the display pixel by pixel, but the program will be spending more time updating the display than doing calculations.

The original image-processing application is single-threaded. You start the processing of the image by selecting a command from the Process menu and waiting for the algorithm to complete. The subroutine that implements the operation runs in the foreground; the rest of the application can't react to external events instantly. Now we'll convert the single-threaded application into a multi-threaded application by executing the image-processing subroutines in a separate thread.

To execute each processing routine in a separate thread, you must create a new subroutine and place the appropriate code in it. The program supports four types of processing: smoothing, sharpening, embossing, and diffusing. We therefore need four subroutines with the appropriate code. These four subroutines are quite similar to the original ones, but they're implemented to be executed in a separate thread. The difference is that none of these subroutines directly updates the image on the PictureBox control. Instead, they call a delegate, which copies the transformed pixels onto the PictureBox.

In the main application, declare a Thread object, which will be used to start any of the image-processing operations, as well as a Bitmap object that holds the original pixels of the image:

```
Dim T As Thread
Dim TStart As ThreadStart
Dim _bitmap As Bitmap
```

We have only one thread; all processing subroutines will run in this thread. It doesn't make any sense to have two procedures operating on the same bitmap, anyway. You'll want to see the results of a process before applying another one to the same image. Besides, if one of the threads involves more calculations than the other, that one may start later than the other thread but still get ahead of it.

To start the *T* thread, initialize it by calling its constructor with the delegate of the appropriate image-processing subroutine. You must first make sure, however, that the thread isn't already running. If a process is running in the thread at the time, you must first abort it, create a new thread, and then start it. The following statements are executed when the Emboss item in the Process menu is selected:

```
Private Sub ProcessEmboss_Click(ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
                                Handles ProcessEmboss.Click
    If Not T Is Nothing Then
        If T.ThreadState = ThreadState.Running Then
            T.Abort()
            T.Join()
        End If
    End If
    processCaption = "Embossing"
   _bitmapwidth = PictureBox1.Image.Width
   _bitmapheight = PictureBox1.Image.Height
    TStart = New ThreadStart(AddressOf BGEmboss)
    T = New Thread(TStart)
    T.Name = "Emboss"
    T.Priority = ThreadPriority.BelowNormal
    T.Start()
End Sub
```

The Abort method doesn't take effect immediately, so you can't assume that the thread has been terminated as soon as you call its Abort method. The code calls the Abort method and immediately after that the Join method. The Join method will not return unless the thread has actually terminated.

Then it sets up a new thread to start the BGEmboss() subroutine, sets the thread's priority, and starts it. You can experiment with the settings of the Priority property to find out how it affects the responsiveness of the application. The more time you give to the thread (by assigning a higher priority to it), the less responsive the entire application gets.

The code in the BGEmboss subroutine is straightforward, except for the statements that update the display. Listing 1.2 shows the code for the BGEmboss subroutine.

Listing 1.2: The BGEmboss Subroutine

```
Sub BGEmboss()
   _bitmap = New Bitmap(PictureBox1.Image)
    PictureBox1.Image = _bitmap
    Dim tempbmp As New Bitmap(PictureBox1.Image)
    Dim pixels(tempbmp.Width) As Color
    Dim i, j As Integer
    Dim DispX As Integer = 1, DispY As Integer = 1
    Dim red, green, blue As Integer
    Dim args(2) As Object
    With tempbmp
        For i = 0 To .Height - 2
            For j = 0 To .Width - 2
                Dim pixel1, pixel2 As System.Drawing.Color
                pixel1 = .GetPixel(j, i)
                pixel2 = .GetPixel(j + DispX, i + DispY)
                red = Math.Min(Math.Abs(CInt(pixel1.R) - _
                         CInt(pixel2.R)) + 128, 255)
                green = Math.Min(Math.Abs(CInt(pixel1.G) - _
                         CInt(pixel2.G)) + 128, 255)
                blue = Math.Min(Math.Abs(CInt(pixel1.B) - _
                         CInt(pixel2.B)) + 128, 255)
                pixels(j) = Color.FromArgb(red, green, blue)
            Next
            args(0) = i
            args(1) = pixels
            If (i Mod 10) = 0 Then
                args(2) = True
            Else
                args(2) = False
            End If
            Me.Invoke(_displayCallBack, args)
        Next
    End With
    Me.Invoke(_doneCallback)
End Sub
```

Every time the subroutine completes the processing of a row of pixels, it calls the displayCallBack method passing a couple of arguments through the *args* array. The displayCallBack method must be declared with the following statement:

```
Private _displayCallBack As ImageDisplayCallback = _
           New ImageDisplayCallback(AddressOf ShowPixels)
```

where *ShowPixels* is the name of the subroutine that actually displays the processed pixels. The ShowPixels() subroutine accepts as arguments a row of pixels (an array of Color values) and the number of the row in the original image it should update. It also accepts a third argument, which is a Boolean value indicating whether the image on the PictureBox control should be invalidated or not. The image is updated each time a set of 10 rows has been transformed. You can set the third argument to True, so that the image on the PictureBox control will be updated with every row of pixels, to see the effect of frequent updates to the overall speed of the application. The following code shows the implementation of the ShowPixels() subroutine:

```
Sub ShowPixels(ByVal Y As Integer, ByVal pixels() As Color, _
               ByVal Refresh As Boolean)
    Dim i As Integer
    For i = 0 To _bitmap.Width - 1
        _bitmap.SetPixel(i, Y, pixels(i))
    Next
    Dim ratio As Single
    ratio = PictureBox1.Height / _bitmapheight
    If Refresh Then
       PictureBox1.Invalidate(New Region(New Rectangle(0, 0,_
                      PictureBox1.Width - 1, Y * ratio + 10)))
    End If
End Sub
```

This subroutine copies the color values passed through the Pixels() array to the appropriate pixels of the original bitmap, invalidates the PictureBox control (if specified by the last argument) and then updates the form's caption to indicate the percentage of the work done so far.

With the ShowPixels() subroutine in place, you can declare a delegate to relate the displayCall-Back method to the ShowPixels() subroutine:

```
Public Delegate Sub ImageDisplayCallback(ByVal Y As Integer, _
                     ByVal pixels() As Color, ByVal Refresh As Boolean)
```

The delegate has the same signature as the ShowPixels() subroutine. To update the display through the ShowPixels() subroutine, the BGEmboss subroutine sets up an array of objects. Each element of this array holds one of the arguments expected by the ShowPixels() subroutine, in the same order as they appear in the declaration of the subroutine. Then it invokes the delegate with the following statement:

```
Me.Invoke(_displayCallBack, args)
```

It's a bit complicated, but keep in mind that the thread should not attempt to update the user interface on the main form directly. The form's visible interface must be updated only from within the thread that owns it. The thread, in effect, raises an event telling the application to update its display.

Open the ImageProcessing application to see the implementation of the BGEmboss method. The code involves calculations with pixel values, which are discussed in detail in Chapter 14 of *Mastering VB.NET*.

The revised ImageProcessing application demonstrates how to start and abort a thread in order to carry out a rather complicated task. One last feature of the application is that you can abort the current process by pressing the Esc key. To capture the Esc key, you must set the form's KeyPreview property to True and insert the following statements in the form's KeyUp event handler:

```
Private Sub Form1_KeyUp(ByVal sender As Object, _
                        ByVal e As System.Windows.Forms.KeyEventArgs) _
                        Handles MyBase.KeyUp
    If e.KeyCode = Keys.Escape Then
        t.Abort()
        Me.Text = "Process interrupted"
    End If
End Sub
```

The code is trivial, because only one thread can be executing at a time. For a more advanced multithreaded image-processing application, you should implement an MDI application that can process multiple images. Normally, the type of processing you'll select from the Process menu applies to the active child form. If you use multithreading, you can process multiple images and watch the transformation of all images in parallel.

If you attempt to terminate the application by clicking the Close button of the form while an image is being transformed, the following exception will be raised:

```
Cannot access a disposed object named "Form1"
```

Do you see why this happened? The main application is shutting down and it releases the Form1 object. The thread, however, is still executing, and when it's time to update the display, it attempts to call the form's Invoke method. But this method no longer exists, because the form itself no longer exists. To prevent this exception, you must terminate the thread from within the form's Closing event with the following statements:

```
Private Sub Form1_Closing(ByVal sender As Object, _
                   ByVal e As System.ComponentModel.CancelEventArgs) _
                   Handles MyBase.Closing
    T.Abort()
    T.Join()
    End
End Sub
```

These statements are commented out in the project's code so that you can watch the behavior of the thread. It continues to exist even after the termination of the project that started it.

You can also experiment with the various settings of the thread's Priority property. The code sets this property to ThreadPriority.BelowNormal. You can zoom in to and out of the image as it's being processed, resize the form on the desktop, and everything will work fine. If you set the priority to AboveNormal, the application will become sluggish and not nearly as responsive. Updating a large bitmap doesn't happen instantly and Windows doesn't get the time it needs to update the display. Before the operating system has a chance to update the display, the CPU switches to the thread, and after a while the PictureBox must be updated again. Of course, a lot depends on the computer on which you're running the application. On my system, the Normal setting worked nicely, even though I used an 850 MHz CPU to develop and test this project. Notice that the overall speed of the application isn't affected significantly by lowering the priority of the thread that does all the work. The reason is that the main application doesn't require much of the CPU time. As soon as the display is updated, the thread kicks in and it takes all of the remaining CPU time anyway.

Listings 1.3 and 1.4 show the code for two of this tutorial's projects.

Listing 1.3: Project AnimatedFlag

```
Imports System.Threading
Public Class Form1
    Inherits System.Windows.Forms.Form

#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)
    If disposing Then
        If Not (components Is Nothing) Then
            components.Dispose()
        End If
    End If
    MyBase.Dispose(disposing)
End Sub

'Required by the Windows Form Designer
Private components As System.ComponentModel.IContainer

'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.
Friend WithEvents PBNorway As System.Windows.Forms.PictureBox
Friend WithEvents PBBrazil As System.Windows.Forms.PictureBox
Friend WithEvents PBUSA As System.Windows.Forms.PictureBox
Friend WithEvents Button1 As System.Windows.Forms.Button
Friend WithEvents ListBox1 As System.Windows.Forms.ListBox
Friend WithEvents Timer1 As System.Windows.Forms.Timer
<System.Diagnostics.DebuggerStepThrough()> _
                   Private Sub InitializeComponent()
    Me.components = New System.ComponentModel.Container()
    Dim resources As System.Resources.ResourceManager = _
                New System.Resources.ResourceManager(GetType(Form1))
    Me.PBNorway = New System.Windows.Forms.PictureBox()
    Me.PBBrazil = New System.Windows.Forms.PictureBox()
    Me.PBUSA = New System.Windows.Forms.PictureBox()
    Me.Button1 = New System.Windows.Forms.Button()
    Me.ListBox1 = New System.Windows.Forms.ListBox()
    Me.Timer1 = New System.Windows.Forms.Timer(Me.components)
    Me.SuspendLayout()
    '
    'PBNorway
    '
    Me.PBNorway.Image = CType(resources.GetObject("PBNorway.Image"), _
               System.Drawing.Bitmap)
    Me.PBNorway.Location = New System.Drawing.Point(20, 4)
    Me.PBNorway.Name = "PBNorway"
    Me.PBNorway.Size = New System.Drawing.Size(36, 42)
    Me.PBNorway.SizeMode = _
               System.Windows.Forms.PictureBoxSizeMode.StretchImage
    Me.PBNorway.TabIndex = 0
    Me.PBNorway.TabStop = False
    '
    'PBBrazil
    '
    Me.PBBrazil.Image = CType(resources.GetObject("PBBrazil.Image"), _
                        System.Drawing.Bitmap)
    Me.PBBrazil.Location = New System.Drawing.Point(91, 4)
    Me.PBBrazil.Name = "PBBrazil"
    Me.PBBrazil.Size = New System.Drawing.Size(36, 42)
    Me.PBBrazil.SizeMode = _
                System.Windows.Forms.PictureBoxSizeMode.StretchImage
    Me.PBBrazil.TabIndex = 1
    Me.PBBrazil.TabStop = False
    '
    'PBUSA
    '
    Me.PBUSA.Image = CType(resources.GetObject("PBUSA.Image"), _
                     System.Drawing.Bitmap)
    Me.PBUSA.Location = New System.Drawing.Point(162, 4)
    Me.PBUSA.Name = "PBUSA"
    Me.PBUSA.Size = New System.Drawing.Size(36, 42)
    Me.PBUSA.SizeMode = _
                 System.Windows.Forms.PictureBoxSizeMode.StretchImage
    Me.PBUSA.TabIndex = 2
    Me.PBUSA.TabStop = False
    '
    'Button1
    '
    Me.Button1.Font = New System.Drawing.Font("Comic Sans MS", 11.25!, _
                     System.Drawing.FontStyle.Regular, _
                     System.Drawing.GraphicsUnit.Point, CType(0, Byte))
    Me.Button1.Location = New System.Drawing.Point(4, 322)
    Me.Button1.Name = "Button1"
    Me.Button1.Size = New System.Drawing.Size(213, 35)
    Me.Button1.TabIndex = 5
    Me.Button1.Text = "Start Animation"
    '
    'ListBox1
    '
    Me.ListBox1.Font = New System.Drawing.Font("Comic Sans MS", 9.0!, _
                   System.Drawing.FontStyle.Regular, _
                   System.Drawing.GraphicsUnit.Point, CType(0, Byte))
    Me.ListBox1.ItemHeight = 17
    Me.ListBox1.Location = New System.Drawing.Point(4, 244)
    Me.ListBox1.Name = "ListBox1"
    Me.ListBox1.Size = New System.Drawing.Size(213, 55)
    Me.ListBox1.TabIndex = 6
    '
    'Timer1
    '
    Me.Timer1.Interval = 5
    '
    'Form1
```

```
        '
        Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
        Me.ClientSize = New System.Drawing.Size(222, 365)
        Me.Controls.AddRange(New System.Windows.Forms.Control() _
                       {Me.ListBox1, Me.Button1, Me.PBUSA, _
                        Me.PBBrazil, Me.PBNorway})
        Me.Name = "Form1"
        Me.Text = "Animation with Threads"
        Me.ResumeLayout(False)
    End Sub

    #End Region

    Dim TNorway As Thread
    Dim TBrazil As Thread
    Dim TUSA As Thread

    Private Sub Button1_Click(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) _
                              Handles Button1.Click
        TNorway = New Thread(AddressOf StartNorway)
        TNorway.Name = "Norway"
        TNorway.Priority = ThreadPriority.Lowest
        TNorway.Start()
        TBrazil = New Thread(AddressOf StartBrazil)
        TBrazil.Name = "Brazil"
        TBrazil.Priority = ThreadPriority.Lowest
        TBrazil.Start()
        TUSA = New Thread(AddressOf StartUSA)
        TUSA.Name = "USA"
        TUSA.Priority = ThreadPriority.Lowest
        TUSA.Start()
        Timer1.Enabled = True
    End Sub

    Sub StartNorway()
        Dim top As Single
        Dim disp As Single
        Dim rnd As New System.Random()
        disp = rnd.Next(1, 5)
        While True
            Console.WriteLine(Thread.CurrentThread.Name)
            top = top + disp / 20
            PBNorway.Top = top
            TNorway.Sleep(1)
            If PBNorway.Top > 200 Then disp = -disp
            If PBNorway.Top < 0 Then disp = -disp
        End While
```

```
End Sub

Sub StartBrazil()
    Dim top As Single
    Dim disp As Single
    Dim rnd As New System.Random()
    disp = rnd.Next(1, 5)
    While True
        Console.WriteLine(Thread.CurrentThread.Name)
        top = top + disp / 20
        PBBrazil.Top = top
        TBrazil.Sleep(2)
        If PBBrazil.Top > 200 Then disp = -disp
        If PBBrazil.Top < 0 Then disp = -disp
    End While
End Sub

Sub StartUSA()
    Dim top As Single
    Dim disp As Single
    Dim rnd As New System.Random()
    disp = rnd.Next(1, 5)
    While True
        Console.WriteLine(Thread.CurrentThread.Name)
        top = top + disp / 20
        PBUSA.Top = top
        TUSA.Sleep(3)
        If PBUSA.Top > 200 Then disp = -disp
        If PBUSA.Top < 0 Then disp = -disp
    End While
End Sub


Private Sub PBBrazil_Click(ByVal sender As System.Object, _
                           ByVal e As System.EventArgs) _
                           Handles PBBrazil.Click
    While TBrazil.ThreadState = ThreadState.WaitSleepJoin
    End While
    If TBrazil.ThreadState = ThreadState.Running Then
        TBrazil.Suspend()
    Else
        If Not TBrazil.ThreadState = ThreadState.Stopped Then
            TBrazil.Resume()
        End If
    End If
End Sub

Private Sub PBBrazil_DoubleClick(ByVal sender As Object, _
                                 ByVal e As System.EventArgs) _
                                 Handles PBBrazil.DoubleClick
    If Not TBrazil.ThreadState = ThreadState.Aborted Then
        If TBrazil.ThreadState = ThreadState.Suspended Then
            TBrazil.Resume()
            TBrazil.Abort()
        End If
    End If
End Sub

Private Sub PBNorway_Click(ByVal sender As System.Object, _
                           ByVal e As System.EventArgs) _
                           Handles PBNorway.Click
    While TNorway.ThreadState = ThreadState.WaitSleepJoin
    End While
    If TNorway.ThreadState = ThreadState.Running Then
        TNorway.Suspend()
    Else
        If Not TNorway.ThreadState = ThreadState.Stopped Then
            TNorway.Resume()
        End If
    End If
End Sub

Private Sub PBNorway_DoubleClick(ByVal sender As Object, _
                                 ByVal e As System.EventArgs) _
                                 Handles PBNorway.DoubleClick
    If Not TNorway.ThreadState = ThreadState.Aborted Then
        If TNorway.ThreadState = ThreadState.Suspended Then
            TNorway.Resume()
            TNorway.Abort()
        End If
    End If
End Sub

Private Sub PBUSA_Click(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) _
                        Handles PBUSA.Click
    While TUSA.ThreadState = ThreadState.WaitSleepJoin
    End While
    If TUSA.ThreadState = ThreadState.Running Then
        TUSA.Suspend()
    Else
        If Not TUSA.ThreadState = ThreadState.Stopped Then
            TUSA.Resume()
        End If
    End If
End Sub
```

```
Private Sub PBUSA_DoubleClick(ByVal sender As Object, _
                              ByVal e As System.EventArgs) _
                              Handles PBUSA.DoubleClick
    If Not TUSA.ThreadState = ThreadState.Aborted Then
        If TUSA.ThreadState = ThreadState.Suspended Then
            TUSA.Resume()
            TUSA.Abort()
        End If
    End If
End Sub

Private Sub Timer1_Tick(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) _
                        Handles Timer1.Tick
    ListBox1.BeginUpdate()
    ListBox1.Items.Clear()
    ListBox1.Items.Add("NORWAY = " & TNorway.ThreadState.ToString("G"))
    ListBox1.Items.Add("BRAZIL = " & TBrazil.ThreadState.ToString("G"))
    ListBox1.Items.Add("USA = " & TUSA.ThreadState.ToString("G"))
    ListBox1.EndUpdate()
End Sub

End Class
```

LISTING 1.4: PROJECT IMAGEPROCESSING

```
Imports System.Threading
Public Class Form1
    Inherits System.Windows.Forms.Form

#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)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub
    Friend WithEvents PictureBox1 As System.Windows.Forms.PictureBox
    Friend WithEvents MainMenu1 As System.Windows.Forms.MainMenu
    Friend WithEvents FileMenu As System.Windows.Forms.MenuItem
    Friend WithEvents FileOpen As System.Windows.Forms.MenuItem
    Friend WithEvents OpenFileDialog1 As System.Windows.Forms.OpenFileDialog
    Friend WithEvents ProcessMenu As System.Windows.Forms.MenuItem
    Friend WithEvents ProcessEmboss As System.Windows.Forms.MenuItem
    Friend WithEvents ProcessSharpen As System.Windows.Forms.MenuItem
    Friend WithEvents ProcessSmooth As System.Windows.Forms.MenuItem
    Friend WithEvents ViewMenu As System.Windows.Forms.MenuItem
    Friend WithEvents ViewNormal As System.Windows.Forms.MenuItem
    Friend WithEvents ViewZoomIn As System.Windows.Forms.MenuItem
    Friend WithEvents ViewZoomOut As System.Windows.Forms.MenuItem
    Friend WithEvents MenuItem1 As System.Windows.Forms.MenuItem
    Friend WithEvents MenuItem2 As System.Windows.Forms.MenuItem
    Friend WithEvents MenuItem4 As System.Windows.Forms.MenuItem
    Friend WithEvents ProcessDiffuse As System.Windows.Forms.MenuItem
    Friend WithEvents FlipH As System.Windows.Forms.MenuItem
    Friend WithEvents FlipV As System.Windows.Forms.MenuItem
    Friend WithEvents RotateLeft As System.Windows.Forms.MenuItem
    Friend WithEvents RotateRight As System.Windows.Forms.MenuItem
    Friend WithEvents RotateMenu As System.Windows.Forms.MenuItem

    'Required by the Windows Form Designer
    Private components As System.ComponentModel.Container

    '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.
    <System.Diagnostics.DebuggerStepThrough()> _
               Private Sub InitializeComponent()
        Me.ViewNormal = New System.Windows.Forms.MenuItem()
        Me.RotateLeft = New System.Windows.Forms.MenuItem()
        Me.OpenFileDialog1 = New System.Windows.Forms.OpenFileDialog()
        Me.FlipV = New System.Windows.Forms.MenuItem()
        Me.ProcessMenu = New System.Windows.Forms.MenuItem()
        Me.ProcessEmboss = New System.Windows.Forms.MenuItem()
        Me.ProcessSharpen = New System.Windows.Forms.MenuItem()
        Me.ProcessSmooth = New System.Windows.Forms.MenuItem()
        Me.ProcessDiffuse = New System.Windows.Forms.MenuItem()
        Me.ViewZoomIn = New System.Windows.Forms.MenuItem()
        Me.RotateRight = New System.Windows.Forms.MenuItem()
        Me.ViewMenu = New System.Windows.Forms.MenuItem()
        Me.ViewZoomOut = New System.Windows.Forms.MenuItem()
        Me.MenuItem1 = New System.Windows.Forms.MenuItem()
        Me.MenuItem2 = New System.Windows.Forms.MenuItem()
        Me.RotateMenu = New System.Windows.Forms.MenuItem()
        Me.FlipH = New System.Windows.Forms.MenuItem()
        Me.PictureBox1 = New System.Windows.Forms.PictureBox()
        Me.FileMenu = New System.Windows.Forms.MenuItem()
        Me.FileOpen = New System.Windows.Forms.MenuItem()
        Me.MenuItem4 = New System.Windows.Forms.MenuItem()
        Me.MainMenu1 = New System.Windows.Forms.MainMenu()
        Me.SuspendLayout()
        '
        'ViewNormal
        '
        Me.ViewNormal.Index = 0
        Me.ViewNormal.Text = "Normal"
        '
        'RotateLeft
        '
        Me.RotateLeft.Index = 0
        Me.RotateLeft.Text = "Rotate Left"
        '
        'FlipV
        '
        Me.FlipV.Index = 3
        Me.FlipV.Text = "Flip Vertical"
        '
        'ProcessMenu
        '
        Me.ProcessMenu.Index = 1
        Me.ProcessMenu.MenuItems.AddRange(New _
                        System.Windows.Forms.MenuItem() _
                        {Me.ProcessEmboss, Me.ProcessSharpen, _
                         Me.ProcessSmooth, Me.ProcessDiffuse})
        Me.ProcessMenu.Text = "Process"
        '
        'ProcessEmboss
        '
        Me.ProcessEmboss.Index = 0
        Me.ProcessEmboss.Text = "Emboss"
        '
        'ProcessSharpen
        '
        Me.ProcessSharpen.Index = 1
        Me.ProcessSharpen.Text = "Sharpen"
```

```
'
'ProcessSmooth
'
Me.ProcessSmooth.Index = 2
Me.ProcessSmooth.Text = "Smooth"
'
'ProcessDiffuse
'
Me.ProcessDiffuse.Index = 3
Me.ProcessDiffuse.Text = "Diffuse"
'
'ViewZoomIn
'
Me.ViewZoomIn.Index = 1
Me.ViewZoomIn.Text = "Zoom In"
'
'RotateRight
'
Me.RotateRight.Index = 1
Me.RotateRight.Text = "Rotate Right"
'
'ViewMenu
'
Me.ViewMenu.Index = 3
Me.ViewMenu.MenuItems.AddRange(New _
               System.Windows.Forms.MenuItem() _
              {Me.ViewNormal, Me.ViewZoomIn, Me.ViewZoomOut})
Me.ViewMenu.Text = "View"
'
'ViewZoomOut
'
Me.ViewZoomOut.Index = 2
Me.ViewZoomOut.Text = "Zoom Out"
'
'MenuItem1
'
Me.MenuItem1.Index = 1
Me.MenuItem1.Text = "Save As"
'
'MenuItem2
'
Me.MenuItem2.Index = 3
Me.MenuItem2.Text = "Exit"
'
'RotateMenu
'
Me.RotateMenu.Index = 2
Me.RotateMenu.MenuItems.AddRange(New _
          System.Windows.Forms.MenuItem() _
          {Me.RotateLeft, Me.RotateRight, Me.FlipH, Me.FlipV})
Me.RotateMenu.Text = "Rotate"
'
'FlipH
'
Me.FlipH.Index = 2
Me.FlipH.Text = "Flip Horizontal"
'
'PictureBox1
'
Me.PictureBox1.Name = "PictureBox1"
Me.PictureBox1.Size = New System.Drawing.Size(624, 360)
Me.PictureBox1.SizeMode = _
         System.Windows.Forms.PictureBoxSizeMode.StretchImage
Me.PictureBox1.TabIndex = 0
Me.PictureBox1.TabStop = False
'
'FileMenu
'
Me.FileMenu.Index = 0
Me.FileMenu.MenuItems.AddRange(New _
              System.Windows.Forms.MenuItem() _
              {Me.FileOpen, Me.MenuItem1, _
               Me.MenuItem4, Me.MenuItem2})
Me.FileMenu.Text = "File"
'
'FileOpen
'
Me.FileOpen.Index = 0
Me.FileOpen.Text = "Open"
'
'MenuItem4
'
Me.MenuItem4.Index = 2
Me.MenuItem4.Text = "-"
'
'MainMenu1
'
Me.MainMenu1.MenuItems.AddRange(New _
                      System.Windows.Forms.MenuItem() _
                      {Me.FileMenu, Me.ProcessMenu, _
                       Me.RotateMenu, Me.ViewMenu})
'
'Form1
'
Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
Me.AutoScroll = True
```

```
        Me.ClientSize = New System.Drawing.Size(624, 377)
        Me.Controls.AddRange(New _
                    System.Windows.Forms.Control() {Me.PictureBox1})
        Me.KeyPreview = True
        Me.Menu = Me.MainMenu1
        Me.Name = "Form1"
        Me.Text = "Image Processing"
        Me.ResumeLayout(False)

    End Sub

#End Region

' The priority of the thread that processes the image is "BelowNormal"
' because we want to make sure the operating system gets
' a chance to update the window as we resize it at run time.

    Dim T As Thread
    Dim TStart As ThreadStart
    Dim _bitmap As Bitmap
    Dim _bitmapwidth As Integer, _bitmapheight As Integer

    Public Delegate Sub ImageDisplayCallback(ByVal Y As Integer, _
                               ByVal pixels() As Color, _
                               ByVal Refresh As Boolean)
    Public Delegate Sub DoneProcessingCallback()
    Private _displayCallBack As ImageDisplayCallback = _
               New ImageDisplayCallback(AddressOf ShowPixels)
    Private _doneCallback As DoneProcessingCallback = _
               New DoneProcessingCallback(AddressOf DoneProcessing)

    Sub DoneProcessing()
        Me.Text = "Done processing"
        PictureBox1.Invalidate()
        T.Abort()
        T.Join()
    End Sub

    Sub ShowPixels(ByVal Y As Integer, ByVal pixels() As Color, _
                 ByVal Refresh As Boolean)
        Dim i As Integer
        For i = 0 To _bitmap.Width - 1
            _bitmap.SetPixel(i, Y, pixels(i))
        Next
        Dim ratio As Single
        ratio = PictureBox1.Height / _bitmapheight
        If Refresh Then
            PictureBox1.Invalidate(New Region(New Rectangle(0, 0,_
                       PictureBox1.Width - 1, Y * ratio + 10)))
        End If
    End Sub

    Private Sub FileOpen_Click(ByVal sender As System.Object, _
                 ByVal e As System.EventArgs) Handles FileOpen.Click
        If Not T Is Nothing Then
            If T.ThreadState = ThreadState.Running Then
                Beep()
                Exit Sub
            End If
        End If
        OpenFileDialog1.Filter = "Images|*.GIF;*.TIF;*.JPG"
        OpenFileDialog1.ShowDialog()
        If OpenFileDialog1.FileName = "" Then Exit Sub
        PictureBox1.Image = Image.FromFile(OpenFileDialog1.FileName)
        PictureBox1.Width = PictureBox1.Height * _
                  PictureBox1.Image.Width / PictureBox1.Image.Height
        Me.Text = "(" & PictureBox1.Image.Width.ToString & ", " & _
                        PictureBox1.Image.Height.ToString & ")"
    End Sub

    Private Sub ProcessEmboss_Click(ByVal sender As System.Object, _
              ByVal e As System.EventArgs) Handles ProcessEmboss.Click
        If Not T Is Nothing Then
            If T.ThreadState = ThreadState.Running Then
                T.Abort()
                T.Join()
            End If
        End If
        Me.Text = "Embossing"
        _bitmapwidth = PictureBox1.Image.Width
        _bitmapheight = PictureBox1.Image.Height
        TStart = New ThreadStart(AddressOf BGEmboss)
        T = New Thread(TStart)
        T.Name = "Emboss"
        T.Priority = ThreadPriority.BelowNormal
        T.Start()
    End Sub

    Private Sub ProcessSharpen_Click(ByVal sender As System.Object, _
              ByVal e As System.EventArgs) Handles ProcessSharpen.Click
        If Not T Is Nothing Then
            If T.ThreadState = ThreadState.Running Then
                T.Abort()
                T.Join()
            End If
        End If
        Me.Text = "Sharpening"
        _bitmapwidth = PictureBox1.Image.Width
        _bitmapheight = PictureBox1.Image.Height
        TStart = New ThreadStart(AddressOf BGSharpen)
        T = New Thread(TStart)
        T.Name = "Sharpen"
        T.Priority = ThreadPriority.BelowNormal
        T.Start()
    End Sub

    Private Sub ProcessSmooth_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles ProcessSmooth.Click
        If Not T Is Nothing Then
            If T.ThreadState = ThreadState.Running Then
                T.Abort()
                T.Join()
            End If
        End If
        Me.Text = "Smoothing"
        _bitmapwidth = PictureBox1.Image.Width
        _bitmapheight = PictureBox1.Image.Height
        TStart = New ThreadStart(AddressOf BGSmooth)
        T = New Thread(TStart)
        T.Name = "Smooth"
        T.Priority = ThreadPriority.BelowNormal
        T.Start()
    End Sub

    Private Sub ProcessDiffuse_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles ProcessDiffuse.Click
        If Not T Is Nothing Then
            If T.ThreadState = ThreadState.Running Then
                T.Abort()
                T.Join()
            End If
        End If
        Me.Text = "Diffusing"
        _bitmapwidth = PictureBox1.Image.Width
        _bitmapheight = PictureBox1.Image.Height
        TStart = New ThreadStart(AddressOf BGDiffuse)
        T = New Thread(TStart)
        T.Name = "Diffuse"
        T.Priority = ThreadPriority.Lowest
        T.Start()
    End Sub

    Private Sub ViewNormal_Click(ByVal sender As System.Object, _
                  ByVal e As System.EventArgs) Handles ViewNormal.Click
        PictureBox1.Width = PictureBox1.Image.Width
        PictureBox1.Height = PictureBox1.Image.Height
    End Sub

    Private Sub ViewZoomOut_Click(ByVal sender As System.Object, _
                  ByVal e As System.EventArgs) Handles ViewZoomOut.Click
        PictureBox1.Width = PictureBox1.Width / 1.25
        PictureBox1.Height = PictureBox1.Height / 1.25
    End Sub

    Private Sub ViewZoomIn_Click(ByVal sender As System.Object, _
                 ByVal e As System.EventArgs) Handles ViewZoomIn.Click
        PictureBox1.Width = PictureBox1.Width * 1.25
        PictureBox1.Height = PictureBox1.Height * 1.25
    End Sub

    Private Sub RotateLeft_Click(ByVal sender As System.Object, _
                 ByVal e As System.EventArgs) Handles RotateLeft.Click
        If Not T Is Nothing Then
            If T.ThreadState = ThreadState.Running Then
                Beep()
                Exit Sub
            End If
        End If
        PictureBox1.Image.RotateFlip(RotateFlipType.Rotate270FlipNone)
        PictureBox1.Width = PictureBox1.Height * _
                  PictureBox1.Image.Width / PictureBox1.Image.Height
    End Sub

    Private Sub RotateRight_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles RotateRight.Click
        If Not T Is Nothing Then
            If T.ThreadState = ThreadState.Running Then
                Beep()
                Exit Sub
            End If
        End If
        PictureBox1.Image.RotateFlip(RotateFlipType.Rotate90FlipNone)
        PictureBox1.Width = PictureBox1.Height * _
                   PictureBox1.Image.Width / PictureBox1.Image.Height
    End Sub

    Private Sub FlipH_Click(ByVal sender As System.Object, _
                 ByVal e As System.EventArgs) Handles FlipH.Click
        If Not T Is Nothing Then
            If T.ThreadState = ThreadState.Running Then
                Beep()
                Exit Sub
            End If
        End If
        PictureBox1.Image.RotateFlip(RotateFlipType.RotateNoneFlipX)
        PictureBox1.Invalidate()
    End Sub

    Private Sub FlipV_Click(ByVal sender As System.Object, _
                   ByVal e As System.EventArgs) Handles FlipV.Click
        If Not T Is Nothing Then
            If T.ThreadState = ThreadState.Running Then
                Beep()
                Exit Sub
            End If
        End If
        PictureBox1.Image.RotateFlip(RotateFlipType.RotateNoneFlipY)
        PictureBox1.Invalidate()
    End Sub

    Private Sub Form1_KeyUp(ByVal sender As Object, _
                     ByVal e As System.Windows.Forms.KeyEventArgs) _
                     Handles MyBase.KeyUp
        If e.KeyCode = Keys.Escape And T.ThreadState = _
                           ThreadState.Running Then
            T.Abort()
            T.Join()
            Me.Text = "Process interrupted"
        End If
    End Sub

    Sub BGSharpen()
        _bitmap = New Bitmap(PictureBox1.Image)
        PictureBox1.Image = _bitmap
        Dim tempbmp As New Bitmap(PictureBox1.Image)
        Dim pixels(tempbmp.Width) As Color
        Dim DX As Integer = 2, DY As Integer = 2
        Dim red, green, blue As Integer
        Dim args(2) As Object
        Dim i, j As Integer
        With tempbmp
            For i = DX To .Height - DX - 1
                For j = DY To .Width - DY - 1
                    red = CInt(.GetPixel(j, i).R) + 0.5 * _
                          CInt((.GetPixel(j, i).R) - _
                          CInt(.GetPixel(j - DX, i - DY).R))
                    green = CInt(.GetPixel(j, i).G) + 0.7 * _
                          CInt((.GetPixel(j, i).G) - _
                          CInt(.GetPixel(j - DX, i - DY).G))
                    blue = CInt(.GetPixel(j, i).B) + 0.5 * _
```

```
                       CInt((.GetPixel(j, i).B - _
                       CInt(.GetPixel(j - DX, i - DY).B)))
                red = Math.Min(Math.Max(red, 0), 255)
                green = Math.Min(Math.Max(green, 0), 255)
                blue = Math.Min(Math.Max(blue, 0), 255)
                pixels(j) = Color.FromArgb(red, green, blue)
            Next
            args(0) = i
            args(1) = pixels
            If (i Mod 10) = 0 Then
                args(2) = True
            Else
                args(2) = False
            End If
            Me.Invoke(_displayCallBack, args)
        Next
    End With
    Me.Invoke(_doneCallback)
End Sub

Sub BGSmooth()
    _bitmap = New Bitmap(PictureBox1.Image)
    PictureBox1.Image = _bitmap
    Dim tempbmp As New Bitmap(PictureBox1.Image)
    Dim pixels(tempbmp.Width) As Color
    Dim args(2) As Object

    Dim DX As Integer = 2, DY As Integer = 2
    Dim red, green, blue As Integer
    Dim i, j As Integer
    With tempbmp
        For i = DX To .Height - DX - 1
            For j = DY To .Width - DY - 1
                red = CInt((CInt(.GetPixel(j - 1, i - 1).R) + _
                       CInt(.GetPixel(j - 1, i).R) + _
                        CInt(.GetPixel(j - 1, i + 1).R) + _
                        CInt(.GetPixel(j, i - 1).R) + _
                        CInt(.GetPixel(j, i).R) + _
                        CInt(.GetPixel(j, i + 1).R) + _
                        CInt(.GetPixel(j + 1, i - 1).R) + _
                        CInt(.GetPixel(j + 1, i).R) + _
                        CInt(.GetPixel(j + 1, i + 1).R)) / 9)
                green = CInt((CInt(.GetPixel(j - 1, i - 1).G) + _
                        CInt(.GetPixel(j - 1, i).G) + _
                        CInt(.GetPixel(j - 1, i + 1).G) + _
                        CInt(.GetPixel(j, i - 1).G) + _
                        CInt(.GetPixel(j, i).G) + _
                       CInt(.GetPixel(j, i + 1).G) + _
```

```
                    CInt(.GetPixel(j + 1, i - 1).G) + _
                    CInt(.GetPixel(j + 1, i).G) + _
                    CInt(.GetPixel(j + 1, i + 1).G)) / 9)

                blue = CInt((CInt(.GetPixel(j - 1, i - 1).B) + _
                    CInt(.GetPixel(j - 1, i).B) + _
                    CInt(.GetPixel(j - 1, i + 1).B) + _
                    CInt(.GetPixel(j, i - 1).B) + _
                    CInt(.GetPixel(j, i).B) + _
                    CInt(.GetPixel(j, i + 1).B) + _
                    CInt(.GetPixel(j + 1, i - 1).B) + _
                    CInt(.GetPixel(j + 1, i).B) + _
                    CInt(.GetPixel(j + 1, i + 1).B)) / 9)
                red = Math.Min(Math.Max(red, 0), 255)
                green = Math.Min(Math.Max(green, 0), 255)
                blue = Math.Min(Math.Max(blue, 0), 255)
                pixels(j) = Color.FromArgb(red, green, blue)
            Next
            args(0) = i
            args(1) = pixels
            If i Mod 10 = 0 Then
                args(2) = True
            Else
                args(2) = False
            End If
            Me.Invoke(_displayCallBack, args)
        Next
    End With
    Me.Invoke(_doneCallback)
End Sub

Sub BGEmboss()
    _bitmap = New Bitmap(PictureBox1.Image)
    PictureBox1.Image = _bitmap
    Dim tempbmp As New Bitmap(PictureBox1.Image)
    Dim pixels(tempbmp.Width) As Color
    Dim i, j As Integer
    Dim DispX As Integer = 1, DispY As Integer = 1
    Dim red, green, blue As Integer
    Dim args(2) As Object
    With tempbmp
        For i = 0 To .Height - 2
            For j = 0 To .Width - 2
                Dim pixel1, pixel2 As System.Drawing.Color
                pixel1 = .GetPixel(j, i)
                pixel2 = .GetPixel(j + DispX, i + DispY)
                red = Math.Min(Math.Abs(CInt(pixel1.R) - _
                      CInt(pixel2.R)) + 128, 255)
                green = Math.Min(Math.Abs(CInt(pixel1.G) - _
                      CInt(pixel2.G)) + 128, 255)
                blue = Math.Min(Math.Abs(CInt(pixel1.B) - _
                      CInt(pixel2.B)) + 128, 255)
                pixels(j) = Color.FromArgb(red, green, blue)
            Next
            args(0) = i
            args(1) = pixels
            If (i Mod 10) = 0 Then
                args(2) = True
            Else
                args(2) = False
            End If
            Me.Invoke(_displayCallBack, args)
        Next
    End With
    Me.Invoke(_doneCallback)
End Sub

Sub BGDiffuse()
    _bitmap = New Bitmap(PictureBox1.Image)
    PictureBox1.Image = _bitmap
    Dim tempbmp As New Bitmap(PictureBox1.Image)
    Dim pixels(tempbmp.Width) As Color
    Dim i, j As Integer
    Dim DX As Integer = 1, DY As Integer = 1
    Dim red, green, blue As Integer
    Dim args(2) As Object
    With tempbmp
        For i = 3 To .Height - 3
            For j = 3 To .Width - 3
                DX = Rnd() * 4 - 2
                DY = Rnd() * 4 - 2
                red = .GetPixel(j + DX, i + DY).R
                green = .GetPixel(j + DX, i + DY).G
                blue = .GetPixel(j + DX, i + DY).B
                pixels(j) = Color.FromArgb(red, green, blue)
            Next
            args(0) = i
            args(1) = pixels
            If i Mod 10 = 0 Then
                args(2) = True
            Else
                args(2) = False
            End If
            Me.Invoke(_displayCallBack, args)
        Next
    End With
```

```
        Me.Invoke(_doneCallback)
    End Sub

    Private Sub MenuItem2_Click(ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
                                Handles MenuItem2.Click
        End
    End Sub
End Class
```