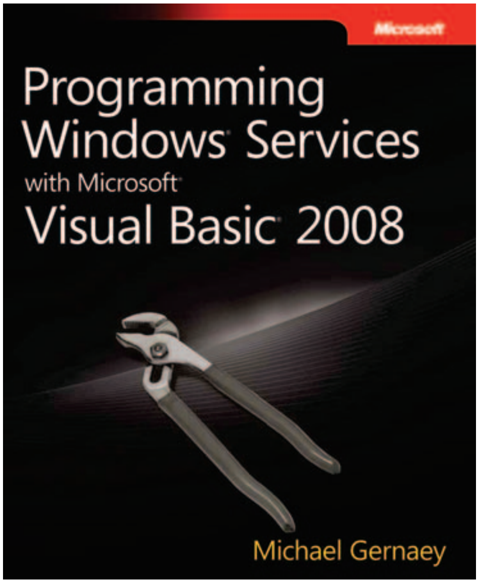

Programming Windows® Services with Microsoft® Visual Basic® 2008

Michael Gernaey

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/11309.aspx

9780735624337

Table of Contents

Part I Defining Windows Services

Part II Creating Interactive Windows Services

Part III Services That Support IT and the Business

Part V Appendices

Chapter 2

Expanding Your Service with Threads

In this chapter:

In the previous chapter we created a simple project using the Microsoft Visual Basic Wizard and templates for Windows Services. Although we were able to install and run the service, it was not very useful—except for demonstration purposes.

You may remember that I said we wanted to avoid doing a large amount of work in the <*OnStart*> method. But if we aren't going to do the work there, where do we do it? This is where *threads* come in. Threads are like mini-processes within the service. Threads allow you to perform multiple actions at the same time within the same application or service.

To determine whether your service requires threads—or how many threads it requires—you have to understand threads a little better. Those of you who already understand how threads work—not just the concept—can skip this section.

Cleaning Up the Service from the Previous Chapter

Before we continue much farther, it is important to note that in many cases we will be continuing the code from the previous chapter. To do this successfully, you should remove the service instance from the previous chapter when you start the next chapter. To uninstall the Tutorials service, open a Visual Studio Command Prompt window and switch to the c:\temp directory. Type **installutil tutorials.exe /u** and press Enter. If the service is still installed, you will see that it has successfully been stopped and uninstalled. If it was not installed, a failure occurs. If you are in the wrong directory, you will receive an error. If you inadvertently deleted the service from the temp directory before you removed it, don't worry. Just rebuild your Chapter 1 code and then remove it.

Understanding Threads

Every application or service has at least one thread. Although the service will usually have much more than just one thread, it has to have at least one thread to perform any work. When an application (or in this case, a service) starts, its primary thread is fired up and begins processing messages from the system. These messages can be mouse clicks, keyboard input, custom events, operating system alerts, and more.

The service we are working with in this book has a primary thread called by the Service Control Manager (SCM). Remember that the SCM states that your service is running only if your <*OnStart*> method returns within 30 seconds of the start request. Imagine, however, if you only used the primary thread to do work. When <*OnStart*> is called, you have potentially a lot of code running that would prevent the <*OnStart*> method from completing in 30 seconds. Therefore the SCM reports back to the user that the service did not start successfully.

With threads, not only can we perform work within the service, but we can also allow the service's primary thread to perform its primary function–coordinating with the SCM through exposed methods such as <*OnStart*> and <*OnStop*>.

For example, suppose you write a game. The game won't perform very well if you expect to draw a large number of graphical aspects at the same time. Like your service, these games require separate threads to perform much of this work.

Determining How Many Threads to Create

You have already been exposed to the minimal integration thread between the SCM and the service, which is the primary SCM thread. For any decent service or real-world application, you will need at least one more thread, which will perform the work while your main thread waits for events triggered by the user.

Note You can trigger a <*ServiceMethod*> that is exposed to the SCM, within your service, but this kind of method is normally triggered by the user through the Administrative Tools, Control Panel, Services panel utility.

The question of how many threads to use is a tricky one. Creating too many threads is dangerous and cumbersome–not only from a coding perspective, but also from an administrative and support perspective. You have to understand that when you add threads, you add complexity to your service or application because of threads' effects on memory, CPU, and other resources.

Thread Synchronization

Many applications and services are written to access data or resources. The developer might want to write code that has multiple threads that can access or share this data at the same time.

Microsoft SQL Server 2005 is a great example, allowing multiple users while sharing data among them. Imagine only allowing one connection and one thread to the entire database system at a time. Ouch!

But what if you were sharing data with dozens of users and all of them tried to update the same record at the same time? This is just not possible—the system has no way to determine how to resolve this situation. Developers must synchronize the order of access to shared data and resources and thereby protect the system from catastrophic anomalies.

Before you determine exactly how many threads you may need or if you need to synchronize your threads (which takes much more effort), you must determine the actions and results you expect to accomplish as well as the data and/or resources required to produce the desired result. If you access other .NET classes or components, it is very important to read up on those components to determine their thread-safe capabilities, which will be defined in the Microsoft Developer Network documentation for Visual Studio 2008. Many are only thread-safe when created as static or shared, depending on the language you create your service in, which will also be specified in MSDN.

Creating Threads

In this chapter we will continue to use code from Chapter 1. The first thing we must do is tell the project that we will be using threads. Visual Basic 2008 supports native threads, which were not supported in earlier versions of Visual Basic.

At the top of the Tutorials.vb file we need to add another import for the .NET *Threading* class, shown in Listing 2-1.

Listing 2-1 Threading Namespace import statement.

```
Imports System.Threading
```

This allows the service to use the classes within the Threading namespace directly, without having to define the namespace for each type declaration. Initially we will only create one thread. Remember, however, that by default the service already has a primary thread. We will use the new thread as a worker thread. Remember also that services are not required to have any specific number of threads—or any extra threads at all. However, to make a service robust we need to use threads that allow us greater control over the tasks required by the service.

Thread Methods

First we must create a thread method, which is used as the starting point of a thread. When you create a thread, it is assigned to a thread method. This method will be used by the thread when it starts. This code can be used by any number of threads. However, the thread itself is isolated, and does not have direct access to other methods or class data members unless they are shared.

The New Code

For this example we will be adding some very simple threading code. The code will write an event into the event log database similar to the way we did in Chapter 1. However, we will use the new thread method to perform the work and we will use the current <*OnStop*> code to clean up the thread and specify that the service is shutting down.

Thread Funtion Code

The first thing we must do is create the thread function or method. In the Tutorials.vb code file, create the method shown in Listing 2-2.

Listing 2-2 Simple thread function.

```
Private Sub ThreadFunc()
   Try
   Catch tab as ThreadAbortException
   Catch ex as Exception
   Finally
   End try
End Sub
```

In this example, I am creating a method called <*ThreadFunc*>. As I mentioned, threads can only access shared data members in a class or must be passed the information directly. In this example, I intentionally do not use what is called an overloaded *parameterized* thread method. I will be using the parameterized method in future chapters. We are going to make a change to the existing code, by adding a log event method that will allow us to write information to the event log but can also be called from the thread.

One important thing to note in the <*ThreadFunc*> method's Try/Catch block is that there are two exception handlers. The second handler catches an exception called Exception. This is a catchall: It will catch any unhandled or thrown exceptions not caught by a previous handler.

The first handler catches an exception called ThreadAbortException. When you want to clean up a thread, your only option is to abort the thread. (I'll discuss this further later in the chapter.) When you abort the thread, it will throw this exception, which allows you to catch the error and perform cleanup before the thread is exited.

The last handler you will see is Finally, which is always called in a Try/Catch scenario, whether an exception occurs or not. Finally allows you to clean up anything you want to clean up before you exit the thread—or potentially before you reach this code again if it is in a loop.

Note To ensure garbage collection of objects created in this thread, you have to understand scope. If you create an object after the Try definition, you cannot clean it up in the Catch or Finally blocks. You must define them outside this scope first.

Event-Logging Code

To properly use event-logging code, we must add an Imports statement under the threading import, as shown in Listing 2-3.

Listing 2-3 Import to interact with event log database and debug .NET classes.

```
Imports System.Diagnostics
```

Now we can create an event logging procedure, shown in Listing 2-4.

Listing 2-4 Shared method for event log database entry creation.

```
Private Shared Sub WriteLogEvent(ByVal pszMessage As String, _
ByVal dwID As Long, ByVal iType As EventLogEntryType, _
ByVal pszSource As String)
  Try
    Dim eLog As EventLog = New EventLog("Application")
    eLog.Source = pszSource
    Dim eInstance As EventInstance = New EventInstance(dwID, 0, iType)
    Dim strArray() As String
    ReDim strArray(1)
    strArray(0) = pszMessage
    eLog.WriteEvent(eInstance, strArray)
    eLog.Dispose()
  Catch ex As Exception
    'We cannot log an event above
    'So we will skip attempting
    'to write this error in the log
    Debug.WriteLine(ex.ToString())
  End Try
End Sub
```

The preceding method is defined as being shared, which means that we can access this method from any instance of or reference to this class, even from a non-shared or static thread method. The <*WriteLogEvent*> will write an event to the Application log. If an exception is generated, we will ignore it for now, because if we can't write to the event log, we can't do much else, except maybe log to a flat file or a database.

The code in Listing 2-4 uses the *WriteEvent* method, which uses the more up-to-date version of the .NET *EventLog* method. You will notice that I am using the *EventInstance* class. This is the more accurate approach to writing events that support the InstanceId property (compared to the older EventID property).

Now that we have an event logging function, let's update the <*OnStop*> and <*OnStart*> methods.

Updating the <*OnStart*> Method

As shown in Listing 2-5, we want to change the current <*OnStart*> method code to use the new <*WriteLogEvent*> method

Listing 2-5 Modifications to <*OnStart*> method to use the <*WriteLogEvent*> method.

```
Dim StartLog As EventLog = New EventLog("Application")
StartLog.Source = "Tutorials"
StartLog.WriteEntry("Tutorials Starting", EventLogEntryType.Information,
 1000)
StartLog.Dispose()
```

Replace the code in Listing 2-5 with the following:

```
WriteLogEvent("Tutorials Starting", 1000, EventLogEntryType.Information,
 "Tutorials")
```

This code allows the <*OnStart*> method to call the newly created shared method and to write to the event log. This makes the <*OnStart*> code cleaner. Next we must update the <*OnStop*> method.

Updating the <*OnStop*> Method

Listing 2-6 shows the current <*OnStop*> method.

Listing 2-6 Modifications to the <*OnStop*> method to use the <*WriteLogEvent*> method.

```
Dim StopLog As EventLog = New EventLog("Application")
StopLog.Source = "Tutorials"
StopLog.WriteEntry("Tutorials Stopping", EventLogEntryType.Information,
 1001)
StopLog.Dispose()
```

Replace the code in Listing 2-6 with the following code:

```
WriteLogEvent("Tutorials Stopping", 1001,
EventLogEntryType.Information, "Tutorials")
```

Updating the <*OnPause*> Method

Listing 2-7 shows the current <*OnPause*> method.

Listing 2-7 Modifications to the <*OnPause*> method to use the <*WriteLogEvent*> method.

```
Dim StopLog As Event log database = New EventLog("Application")
StopLog.Source = "Tutorials"
StopLog.WriteEntry("Tutorials Pausing", EventLogEntryType.Information,
 1001)
StopLog.Dispose()
```

Replace the code in Listing 2-7 with the following code:

```
WriteLogEvent("Tutorials Pausing", 1002, EventLogEntryType.Information,
 "Tutorials")
```

Updating the <*OnContinue*> Method

Listing 2-8 shows the current <*OnContinue*> method.

Listing 2-8 Modifications to the <*OnContinue*> method to use the <*WriteLogEvent*> method.

```
Dim StopLog As Event log database = New EventLog("Application")
StopLog.Source = "Tutorials"
StopLog.WriteEntry("Tutorials Continuing",
EventLogEntryType.Information, 1003)
StopLog.Dispose()
```

Replace the code in Listing 2-8 with the following code:

```
WriteLogEvent("Tutorials Continuing", 1003, _
 EventLogEntryType.Information, "Tutorials")
```

Updating the Thread Method

Now we can add code to the <*ThreadFunc*> method, which will make the thread useful and demonstrate its ability to communicate with the shared <*WriteLogEvent*> method.

Listing 2-9 shows what the entire method code will look like.

Listing 2-9 *Thread* method code with event-logging support.

```
Private Sub ThreadFunc()
Try
  WriteLogEvent("Thread Function Information - " + Now.ToString, 1005, _
 EventLogEntryType.Information, "Tutorials")
Catch tab As ThreadAbortException '
Catch ex As Exception
  WriteLogEvent("Thread Function Error - " + Now.ToString, 1005, _
 EventLogEntryType.Error, "Tutorials")
End Try

End Sub
```

The code shown in Listing 2-9 will attempt to write an event to the Application log. In the event of an exception, the code again tries to write an event to the Application log. I have added this code only for demonstration purposes so that you can see how TheadAbortException is raised. However, here we are using another method to make the call and that method has its own error handlers, so no unhandled exceptions should occur here.

Executing the Thread

Once all the code is in place, we have to create a thread, assign it to use the thread method, and then start the thread. Threads have several properties, which I will describe in this section.

> **Note** Remember that we are not using a parameterized thread for this example. Therefore the event log entry contains a static message.

Because the desired functionality of the service is to have the <*OnStart*> method do minimal work and then return back to the SCM, we will modify the <*OnStart*> method to create an instance of a thread, assign it to the thread method, set the priority, and then execute or start the thread. Once these steps have been completed, the thread will write to the Application log its standard started message and return control back to the SCM. Because we create the thread in its own space, the <*OnStart*> method is not blocked by its creation or the work it performs. Therefore, minimal time is required to set up and execute a thread.

Updating <*OnStart*>

We need to update the <*OnStart*> method to create and run a thread that will execute the new <*ThreadFunc*> code. Listing 2-10 shows what the finished method should look like.

Listing 2-10 Updated <*OnStart*> method with thread support.

```
Protected Overrides Sub OnStart(ByVal args() As String)
'Add code here to start your service. This method should set things
'in motion so your service can do its work.
  Try
    Dim tmpThread As New Thread(AddressOf ThreadFunc)
    tmpThread.Name = "Tutorials Worker Thread"
    tmpThread.Priority = ThreadPriority.Normal
    tmpThread.Start()
    WriteLogEvent("Tutorials Starting", 1000, _
EventLogEntryType.Information, "Tutorials")
  Catch ex As Exception
    'We Catch the Exception
    'to avoid any unhandled errors
    'and we will stop the service if any occur here
    Me.Stop()
  End Try
End Sub
```

Once you have completed implementing the changes to the <*OnStart*>, save and build the project.

Install and Test Your Service

Copy the tutorials.exe from the bin\Release directory to the c:\temp directory, replacing the Tutorials.exe we created in Chapter 1. Open a Visual Studio Command Prompt window and switch to the c:\temp directory. Type **installutil tutorials.exe** and press Enter. After the service installs correctly, open the Services control panel utility by clicking Start, clicking Administrative Tools, clicking Control Panel, and then clicking Services. Then start the Tutorials service by right-clicking it and selecting Start. Using the Event Viewer, you will see the events from the <*OnStart*> and <*ThreadFunc*> *methods*. Stop the service and you will now see the event from the <*OnStop*> method.

Note In subsequent chapters, I won't explain how to install and remove your service—I will simply indicate when you need to do so.

What Is Thread Cleanup?

We have now expanded the code so that we can call methods on the class and/or code within thread functions. However, we have not yet made it possible to clean up the threads if the user were to stop the service while the threads were actively processing. We need to have control over the cleanup of the threads because unlike applications that run on the client, services are required to either shut down quickly or update the Service Control Manager with an estimate of how much longer the services need to shut down. If the Services does not request more time, the SCM will consider the service to be in a hung or unresponsive state, which will most likely require a reboot of the system. In some cases you can look in Task Manager or use the Windows Resource Kit to terminate the rogue service. Over time, however, attempting to forcibly terminate a process can cause operating system or application instability.

Thread Cleanup Availability

We need to make the threads that we create accessible to the rest of the service methods, not just the one that starts or creates the thread. You can do this in many ways—by creating a pool of threads or a class of thread-exposing objects, for example—but for now we're going to use the simplest way possible. We will be adding a list, or collection of threads, that is available privately to the service but is not available to external processes.

Threads and Accessibility

By default, threads have limited access to other members or data in your service. Each thread only has access to either shared data members or methods and—if you are using a parameterized thread method—the object that is passed to it. Threads—which run in different scopes to be able to share data—must be coded in a way that protects your data. (For more information, see "Thread Synchronization" earlier in this chapter.) From a UI perspective, it is possible to create delegates which can be used to participate with other threads. In this case you do not need synchronization to use the delegates themselves, but you may need synchronization within the method executed on the delegate, to protect the actions being performed on the delegate's behalf.

A Problem with the Current <*OnStart*> Thread

If you look back at the previous <*OnStart*> method, you'll notice that we are creating a thread. The problem is that we are creating the thread with a local variable instance, which makes it local-scope only. Once the thread has started, it will continue to run. However, the thread variable, or pointer to the thread, goes out of scope, and we .now have no way to directly access the thread and stop it from doing its work—or clean it up. This lack of access would be a huge problem if a thread became unstable, or worse if it exhibited rogue behavior such as accessing off-limits data or causing a memory leak, CPU spike, or other resource issue.

Fixing the Thread-Scope Issue

To resolve the thread-scope issue we have to make the thread available to either the global scope of the application or to some part of the service that allows us to clean up threads. In this example—because we are only using a single thread—we are going to create a private data member of type *Thread* that is global to the service class, which we are calling *Tutorials*.

Directly after the class definition code, add the code shown in Listing 2-11.

Listing 2-11 Code to add a private thread member variable to the service.

```
Public Class Tutorials
  Private m_WorkerThread As Thread = Nothing
```

The code in Listing 2-11 will create a variable that will store a thread pointer after we create it. This variable doesn't store anything yet. We have to assign it something before we can use it. When we add the code shown in Listing 2-12, the class definition allows this variable to be available to any method in the service, except directly by the threads we create because those threads require differently scoped variables. Again this variable is intended to be available to the service, not just by the threads we create, so that we can clean it up later. The variable is not required to be available to the threads themselves because a thread can clean itself up.

Creating the Thread in <*OnStart*>

We need to change the current <*OnStart*> method so that it no longer uses a local variable for the thread. In the <*OnStart*> method we will change the code shown in the first part of Listing 2-12 with the bolded code that follows it.

Listing 2-12 Modifications to <*OnStart*> to fix the thread-scope issue.

```
Dim tmpThread As New Thread(AddressOf ThreadFunc)
tmpThread.Name = "Tutorials Worker Thread"
tmpThread.Priority = ThreadPriority.Normal
tmpThread.Start()
m_WorkerThread = New Thread(AddressOf ThreadFunc)
m_WorkerThread.Name = "Tutorials Worker Thread"
m_WorkerThread.Priority = ThreadPriority.Normal
m_WorkerThread.Start()
```

Now that we are creating a thread using the private class variable, we have to worry about cleaning it up.

Thread Cleanup

Before we get into the code itself, you have to understand that like other .NET variables, thread variables have a scope. Global variables are just that—global—and can be accessed by other methods. Originally the thread variable was only local to the <*OnStart*> method. Now it is not.

Why is this distinction so important? You should never just create threads that your application can't clean up. When an application exits, the threads and resources the application allocated should be released, even if those resources are no longer visible to the application itself

However, a service works a little differently. When a service shuts down, it expects you to have cleaned up any existing threads. If you didn't, and the service recognizes this fact—and you haven't told the service to wait for you to complete cleanup—the service will cause the Service Control Manager to throw back an error to the user. You will often need to use Task Manager to terminate the now rogue and abandoned application service.

Cleaning Up the <*OnStop*> Method

In the <*OnStop*> method, we will not only write an event to the Application log, but also shut down the thread. It's important to note that <*OnStop*> is similar to <*OnStart*> in that it can only take so much time before returning. However, in the <*OnStop*> method you can request more time from the SCM to continue cleaning up. We won't need to do this because we only have two actions and both are extremely simple.

The top part of Listing 2-13 shows the current code, which we will replace with the bolded code that follows it. This is the new cleanup code.

Listing 2-13 Modifications to <*OnStop*> to support new thread scope.

```
Try
WriteLogEvent("Tutorials Stopping", 1001, EventLogEntryType.Information,
 "Tutorials")
Catch ex As Exception
WriteLogEvent(ex.ToString, 1001, EventLogEntryType.Error, "Tutorials")
End Try
Try
  If Not m_WorkerThread Is Nothing Then
    Try
      m_WorkerThread.Abort()
      m_WorkerThread = Nothing
    Catch ex As Exception
       m_WorkerThread = Nothing
    End Try
  End If
  WriteLogEvent("Tutorials Stopping", 1001, _
EventLogEntryType.Information, "Tutorials")
Catch ex As Exception
  WriteLogEvent(ex.ToString, 1001, EventLogEntryType.Error, "Tutorials")
End Try
```

Now we are able to control how the thread is terminated and when, because the thread is a data member of the service class. This means that any method that is part of the service class can terminate that thread at any time. You should always ensure that the thread still exists before attempting to abort it.

About Thread Abort

When you call Abort on an active thread, inside that thread's instance of the thread method it will throw a ThreadAbortException. So we will add a handler for this exception. Currently the service thread function completes its task so quickly that we will not see the abort exception. The thread exits normally before we could call the <*OnStop*> method.

Note When you abort a thread, there is no guarantee that the thread will terminate or even throw the ThreadAbortException immediately.

Making Thread Cleanup Useful

Although we have added in thread cleanup code, we still have a problem. The current thread method implementation actually performs only a single action and then exits. Although this doesn't make the cleanup code totally useless, its value is questionable because the thread has already exited and cleaned itself up.

To remedy this, we will modify the thread method to do two things:

- Add a handler for the ThreadAbortException
- Add code to keep the thread alive so that the cleanup code will execute

Adding Code to ThreadAbortException

Listing 2-14 shows the code we add to ensure that the ThreadAbortException handler is being used properly.

Listing 2-14 Modifications to <*ThreadFunc*> to handle ThreadAbortException.

```
Try
  WriteLogEvent("Thread Function Information - " + Now.ToString, 1005, _
 EventLogEntryType.Information, "Tutorials")
Catch tab As ThreadAbortException 'this must be listed first as
Catch ex As Exception
  WriteLogEvent("Thread Function Error - " + Now.ToString, 1005, _
EventLogEntryType.Error, "Tutorials")
End Try
Try
 WriteLogEvent("Thread Function Information - " + Now.ToString, 1005, _
 EventLogEntryType.Information, "Tutorials")
'this must be listed first as Exception is the master catch
Catch tab As ThreadAbortException
‘Clean up the thread here
WriteLogEvent("Thread Function Abort Error - " + Now.ToString, 1006, _
 EventLogEntryType.Error, "Tutorials") Catch ex As Exception
 WriteLogEvent("Thread Function Error - " + Now.ToString, 1005, _
 EventLogEntryType.Error, "Tutorials")
End Try
```

In Listing 2-14 we added ThreadAbortException, which will attempt to write an event to the Application log, alerting us about a request to abort the thread. The only problem is that this can't happen because the code runs only once and then exits. By the time you do start the service it has probably run this code and exited. To fix this we need to make sure that the thread continues to run long enough for the cleanup code and this new exception to be executed.

Keeping the Thread Alive

To resolve the issue of the thread exiting too quickly, we will add in a loop that will run the code to write an event over and over. In many cases threads will do the same work repeatedly. However, this doesn't mean that the thread will be active all the time—instead, it will have a sleep interval before it continues its work or starts over again.

We will define a constant called THREAD_WAIT, shown in Listing 2-15. This constant is just below m_WorkerThread at the top of Tutorials.vb.

Listing 2-15 Creating the thread loop wait variable.

```
Public Class Tutorials

    Private m_WorkerThread As Thread = Nothing
    Private Const THREAD_WAIT As Integer = 5000
```

We will wrap the current code in a Do/Loop and use a *Thread* class's *Sleep* method to pause the thread.

Tip The *Thread* class's *Sleep* method uses milliseconds to represent its sleep time. You must convert seconds, minutes, or your *TimeSpan* into milliseconds.

Listing 2-16 shows the new thread method code.

Listing 2-16 New thread method code implementing keep-alive logic.

```
Do
  Try
    WriteLogEvent("Thread Function Information - " + Now.ToString, 1005, _
 EventLogEntryType.Information, "Tutorials")
'this must be listed first as Exception is the master catch
  Catch tab As ThreadAbortException
'Clean up the thread here
    WriteLogEvent("Thread Function Abort Error - " + Now.ToString, 1006, _
 EventLogEntryType.Error, "Tutorials")
  Catch ex As Exception
    WriteLogEvent("Thread Function Error - " + Now.ToString, 1005, _
 EventLogEntryType.Error, "Tutorials")
  End Try
  Thread.Sleep(THREAD_WAIT)
Loop
```

The thread will run the code, sleep for five seconds, and then run the code again. When the thread is aborting in the <*OnStop*> method, it will cause the ThreadAbortException to be called, in which case it will log another event, letting us know that the thread was aborted. We could perform any necessary cleanup there. However, do not make a habit of spending long amounts of time in the <*OnStop*> method or you could possibly lock up the service.

Install and Verify

Before you compile and install your new service version, make sure to remove the old one. When you run the service, you will see the information event logging the time every five seconds until you stop the service.

Now you have an understanding of how to use threads in your services. There is no real limit to how many you can use. However, you should use threads and resources wisely. Threads that run out of control can hang your systems, lock up your processors or your data, block users from retrieving information and, even worse, crash your system.

Extending <*OnPause*> and <*OnContinue*>

Remember that we created a way to indirectly control the flow of service. Using <*OnPause*> and <*OnContinue*>, we can write code that will allow us to either block threads from doing work or make them intuitive enough to know whether they should exit or merely delay their processing responsibilities.

Ways to Control Thread Processing

We will implement a couple of different ways to control what the threads do.

Thread Suspension

Our first attempt at thread control will be to use the built-in *Thread* class method called *Suspend*. This method will allow us to stop a thread in its tracks, or at least attempt to. A thread suspension can fail, in which case we would end up in a situation that we must code for—a rogue thread. However, for the purposes of this example, we will implement the Thread Suspension and Thread Resume features of the *Thread* class.

Updating <*OnPause*>

We will use the <*OnPause*> method to suspend the thread; we will use <*OnContinue*> to resume the thread. Listing 2-17 shows the updated code for <*OnPause*>.

Listing 2-17 Modifications to <*OnPause*> to support thread suspension.

```
Protected Overrides Sub OnPause()
    Try
        If (Not m_WorkerThread Is Nothing) Then
            Try
                m_WorkerThread.Abort()
            Catch ex As Exception
                'we do not care about this
                'exception as we are shutting it down
                'anyway
                m_WorkerThread = Nothing
            End Try
        End If

        WriteLogEvent("Tutorials Pausing", 1002,
 EventLogEntryType.Information, "Tutorials")
    Catch ex As Exception
        'We Catch the Exception
        'to avoid any unhandled errors
        'since we are pausing and
        'logging an event is what failed
        'we will merely write the output
        'to the debug window
        Debug.WriteLine("Error pausing service: " + ex.ToString())
        Me.Stop()
    End Try
End Sub
```

Updating <*OnContinue*>

We now have to update the <*OnContinue*> method to allow us to resume the thread. Listing 2-18 shows the updated code.

Listing 2-18 Modifications to <*OnContinue*> to support continuing the thread.

```
Protected Overrides Sub OnContinue()
    Try
        Try
            'Create a new thread
            'and start it just like
            'in the OnStart
            m_WorkerThread = New Thread(AddressOf ThreadFunc)
            m_WorkerThread.Name = "Tutorials Worker Thread"
            m_WorkerThread.Priority = ThreadPriority.Normal
            m_WorkerThread.Start()
        Catch ex As Exception
            WriteLogEvent("Tutorials Unable to Continue:" + vbCrLf +
 ex.ToString(), 1010, EventLogEntryType.Information, "Tutorials")
            m_WorkerThread = Nothing
        End Try

        WriteLogEvent("Tutorials Continuing", 1003,
EventLogEntryType.Information, "Tutorials")
    Catch ex As Exception
        'We Catch the Exception
        'to avoid any unhandled errors
        'since we are resuming and
        'logging an event is what failed
        'we will merely write the output
        'to the debug window
        Debug.WriteLine("Error resuming service: " + ex.ToString())
        Me.Stop()
    End Try
End Sub
```

In Listing 2-18, we use the *Resume* method to start the thread back where it was suspended. You have to be careful about how your code handles being suspended. While the thread is suspended, you will receive no processing notifications. Therefore, thread suspension is not always the best solution. Let's look at another scenario.

Using Thread State Control

Another way to control thread processing is through state variables. You can use individual variables for each state, or—as in the following example—you can use a type that will affect each thread.

Listing 2-19 shows the code we will add to the thread definition at the top of the class definition.

Listing 2-19 Code to create a thread state capability.

```
Private Const THIRTY_SECONDS As Long = 30000
Private Const TIME_OUT As Long = 15000
Private Structure Thread_Action_State
   Private m_Pause As Boolean
   Private m_Stop As Boolean
   Public Property Pause() As Boolean
     Get
       Return m_Pause
     End Get
     Set(ByVal value As Boolean)
       m_Pause = value
     End Set
   End Property
   Public Property StopThread() As Boolean
     Get
       Return m_Stop
     End Get
     Set(ByVal value As Boolean)
       m_Stop = value
     End Set
    End Property
End Structure
Private Shared m_ThreadAction As New Thread_Action_State
```

This code adds the following features to the service:

First we are adding a constant called THIRTY_SECONDS. I mentioned earlier the service can request additional shutdown time from the SCM. In this case we are going to request an additional 30 seconds to complete processing and cleanup before we exit.

Next we add the TIME_OUT constant. Because we are now using multiple threads, we need to make sure that the processing threads are completed before the primary thread says we are done by exiting the <*OnStop*> method. To do this, we will use the processing thread's *Join* method, which will allow us to either block indefinitely for that thread to complete or wait for a specified period of time. In this case we will wait 15 seconds for the thread to complete its task or shut itself down.

Next we add the Thread_Action_State structure. This structure has two properties: one states whether the service is paused, telling the processing threads to pause, and the other tells the threads that the service is shutting down and they need to exit.

Last we add the m_ThreadAction variable. This variable is shared, or static, meaning that all threads can see these values and there is no need to pass it around to each thread. This also means that the variable can be accessed directly without creating an instance of the class itself. The thread class instance is created automatically when you start the service so you don't need to instantiate it by any other means. The default values are false, so the threads will neither be stopped nor paused when the service starts up.

Updating <*OnPause*>

Now that we have added new state controls, we need to modify <*OnPause*>, <*OnStop*>, and <*OnContinue*> to reflect the state changes. In the previous <*OnPause*> we suspended the thread. We need to remove or comment that code out and add in the new state change code. <*OnPause*> should now look like the code shown in Listing 2-20.

Listing 2-20 Modifications to <*OnPause*> to use thread state management.

```
Protected Overrides Sub OnPause()
  Try
    m_ThreadAction.Pause = True
    WriteLogEvent("Tutorials Pausing", 1002, EventLogEntryType.Information, _
 "Tutorials")
  Catch ex As Exception
    'We Catch the Exception
    'to avoid any unhandled errors
    'since we are pausing and
    'logging an event is what failed
    'we will merely write the output
    'to the debug window
    Debug.WriteLine("Error pausing service: " + ex.ToString())
    Me.Stop()
  End Try
End Sub
```

This code will merely change the state of the threads to Paused. Although we have more work to do, controlling the thread's state is much simpler here than suspending the threads—and safer, too.

Updating <*OnContinue*>

As with <*OnPause*>, <*OnContinue*> must reflect a similar change, shown in Listing 2-21.

Listing 2-21 Modifications to <*OnContinue*> to support thread state management.

```
Protected Overrides Sub OnContinue()
  Try
    m_ThreadAction.Pause = False
    WriteLogEvent("Tutorials Continuing", 1003, EventLogEntryType.Information,
Tutorials")
  Catch ex As Exception
    'We Catch the Exception
    'to avoid any unhandled errors
    'since we are resuming and
    'logging an event is what failed
    'we will merely write the output
    'to the debug window
    Debug.WriteLine("Error resuming service: " + ex.ToString())
    Me.Stop()
  End Try
End Sub
```

You will notice that just as <*OnPause*> set the state to True, we now set it to False so that the threads can continue their work.

Updating <*OnStop*>

Last we will update <*OnStop*>, which requires a bit more work than updating the previous two methods. Not only do we need to tell the threads to stop, but we also want to attempt to wait for them so that we know they are completed and cleaned up before exiting. Listing 2-22 shows the updated code.

Listing 2-22 Modifications to <*OnStop*> to support thread state management.

```
Protected Overrides Sub OnStop()
    ' Add code here to perform any tear-down
     'necessary to stop your service.
    Try
        If (Not m_WorkerThread Is Nothing) Then
            Try
                Me.RequestAdditionalTime(THIRTY_SECONDS)
                m_WorkerThread.Join(TIME_OUT)
                m_ThreadAction.StopThread = True
            Catch ex As Exception
                'Do Nothing
            End Try
        End If

        WriteLogEvent("Tutorials Stopping", 1001,
EventLogEntryType.Information, "Tutorials")
    Catch ex As Exception
        'We Catch the Exception
        'to avoid any unhandled errors
        'since we are stopping and
        'logging an event is what failed
        'we will merely write the output
        'to the debug window
        m_WorkerThread = Nothing
        Debug.WriteLine("Error stopping service: " + ex.ToString())
    End Try
End Sub
```

First, I replaced the Abort and Nothing lines of code. Because we aren't going to directly clean up the thread, instead telling it when to clean itself up, we have to remove these.

As mentioned when I described the new <*OnStop*> code, I will first request an additional 30 seconds from the SCM so that it doesn't believe we are being unresponsive.

Next I will add the *Join* method to the worker thread. This is like saying to the thread, "I am waiting for you to exit. Let me know when you are done." However, although I could wait forever for the service to exit, I have asked for only 30 extra seconds from the SCM. Therefore I have set a 15-second time-out for the thread to exit before I move on, so that I don't cause the SCM to consider the service unresponsive to the stop request.

Next I set the state of the threads to stopped. You may wonder why I didn't do this first. I could have but then, by time I got to the *Join* method, the thread could be invalidated and cause an exception. I'm prepared for that possibility, but I prefer to avoid it.

The last step is to write the original event, exit, and return control to the SCM.

Note If for some reason the thread I join with is not cleaned up in the allotted time, it will get cleaned up when the process exits. However, if this doesn't happen we may need to reboot or terminate the service in Task Manager.

Updating <*ThreadFunc*>

At this point all the code I added is useless unless I first modify the <*ThreadFunc*> method to handle these state changes. I usually don't directly add state change checks to my threads without wrapping them in a call that returns a bool. I find that sometimes I want to do more than just exit or pause a thread when the state changes. Because I want to have multiple state change checks in my thread—which could make it quite large or complex—I wouldn't want to copy and paste this excessive code all over. I don't need to add a wrapper method for every state change possible because I already defined these state changes in the the StopThread and Pause properties of the Thread.

Listing 2-23 shows the new <*ThreadFunc*> code.

Listing 2-23 Modifications to thread method to support thread state management.

```
Private Sub ThreadFunc()
  While Not m_ThreadAction.StopThread
    If Not m_ThreadAction.Pause Then
      Try
        WriteLogEvent("Thread Function Information - " + Now.ToString, _
1005, EventLogEntryType.Information, "Tutorials")
      Catch tab As ThreadAbortException _
'this must be listed first as Exceptionis the master catch
        'Clean up the thread here
        WriteLogEvent("Thread Function Abort Error - " + Now.ToString, _
 1006, EventLogEntryType.Error, "Tutorials")
      Catch ex As Exception
        WriteLogEvent("Thread Function Error - " + Now.ToString, 1005, _
 EventLogEntryType.Error, "Tutorials")
      End Try
    End If
   Thread.Sleep(THREAD_WAIT)
  End While
End Sub
```

This function has changed quite a bit. First, I changed from a Do Loop to a While End While loop. I am using an outer loop to validate that the thread is not in a stopped State. If I hit this state, I will exit the While loop and exit the thread, hence shutting it down.

I am validating that the service is not in a Paused state. If the service is in a Paused state, none of the code is executed and it will merely hit Thread.Sleep. This causes the service thread to sleep for five seconds. Then it attempts to do the While loop and If check again. This process will continue until someone stops the service, in which case it will exit. Pausing will not exit—it will just stop it from doing any active processing.

When this thread exits the *Join* that the main thread attached to in the <*OnStop*> method, the event is released and <*OnStop*> method processing continues, causing the service itself to finally exit and the SCM to report it has stopped.

Importance of the THREAD_WAIT Value

Because we are only doing a 15-second join with this thread to validate that it has shut down, if we set the THREAD_WAIT beyond 10 or 12 seconds, the thread could very likely still be asleep before it validates that it was supposed to shut down. Therefore you should check before you go to sleep whether you were supposed to stop. Pausing is not a big deal, but if you were to have a process internal to the thread that took more than 5 to 10 seconds to complete, and it had just started, and then you went to sleep for 5 or more seconds, you could easily exceed this 15-second join. For this reason, update your <*ThreadFunc*> by wrapping your Sleep call as shown in Listing 2-24.

Listing 2-24 Update to thread method thread sleep call code.

```
If Not m_ThreadAction.StopThread Then
  Thread.Sleep(THREAD_WAIT)
End If
```

You may think this is merely adding extra code, but it isn't. It is saving us from missing our required deadline of 15 seconds.

Summary

We have successfully created a fairly extensive multi-threaded service. At this point you might not see its value because we only have one worker thread. However, the abilities we have included in this version can be easily extended in upcoming chapters, where we will use more than just one worker thread. Controlling thread states is of key importance to services for stability and usability.

- Threads are a way to allow applications to perform multiple actions in a concurrent manner.
- Microsoft Visual Basic 2008 supports threads natively with the *System.Threading* class.
- Microsoft Visual Basic 2008 makes the creation of threads simple, but you must also be careful when using this powerful programming technique.
- Use caution when creating threads. Overuse or improper use can cause severe side effects and instability.