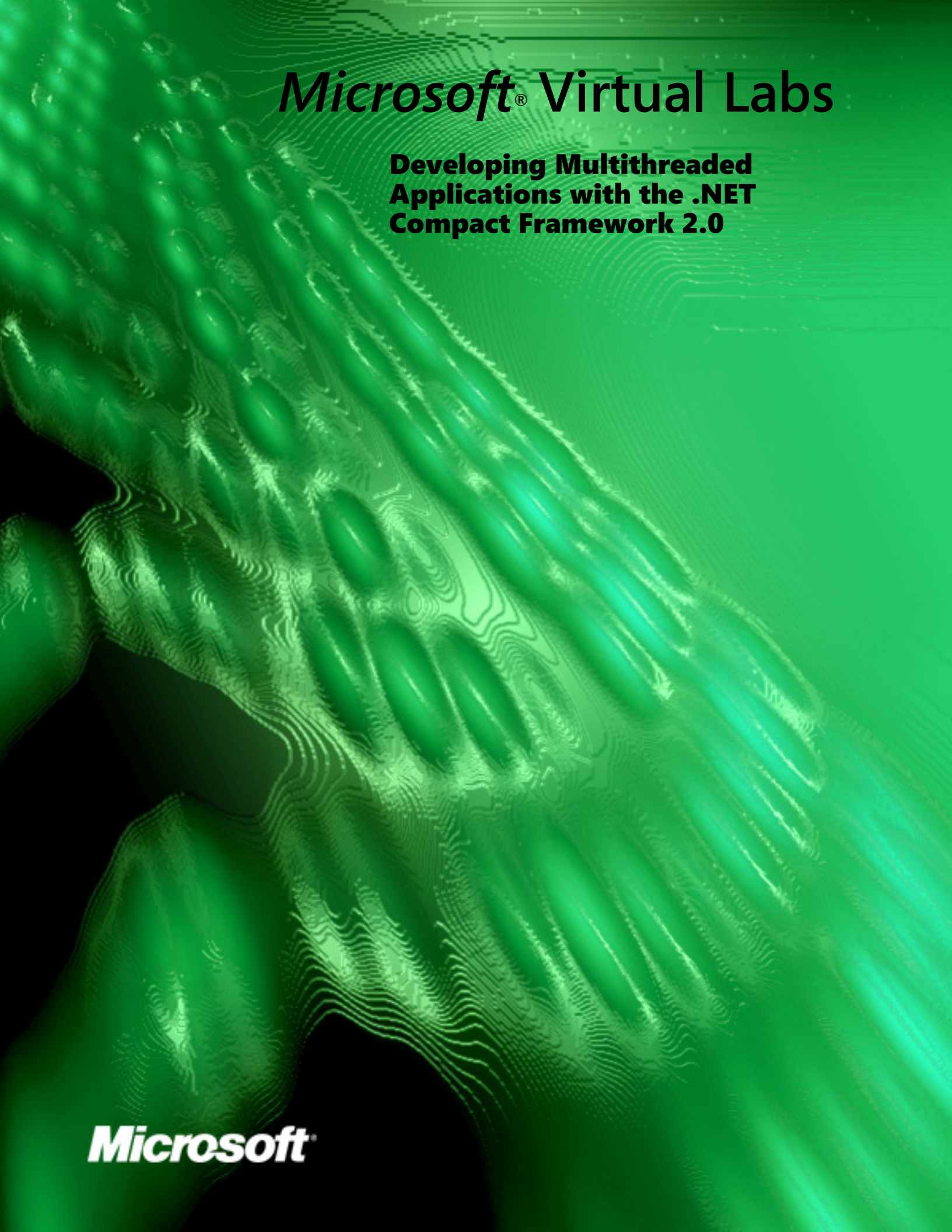




Microsoft® Virtual Labs

**Developing Multithreaded
Applications with the .NET
Compact Framework 2.0**

Microsoft®

Table of Contents

Developing Multithreaded Applications with the .NET Compact Framework 2.0	1
Exercise 1 Creating a Single-Threaded Application by Using the .NET Compact Framework 2.0	2
Exercise 2 Modifying the Multithreaded Application	6
Exercise 3 Examining Thread and ThreadPool	11
Exercise 4 Updating User Interface Controls Inside Threads	14
Exercise 5 Using Synchronization Objects to Synchronize Threads	19
Summary.....	30

Developing Multithreaded Applications with the .NET Compact Framework 2.0

Objectives

The objective of this lab is to demonstrate the multithreading capabilities of the .NET Compact Framework version 2.0 to create responsive applications that target Windows CE version 5.0–based devices or Windows Mobile–based devices. Because completing all exercises in the lab may take more time than you have available, you can select only those exercises that are most useful for you. Each exercise starts with a completely new project, and there is no explicit order in which you have to perform the exercises. However, if you are new to managed multithreaded development, you should start with the first exercise.

In this HOL, you will perform the following exercises:

- Creating a single-threaded application by using the .NET Compact Framework 2.0
- Modifying the multithreaded application
- Examining Thread and ThreadPool
- Updating user interface controls inside threads
- Using synchronization objects to synchronize threads

Estimated Time to Complete This Lab

75 Minutes

Computer used in this Lab



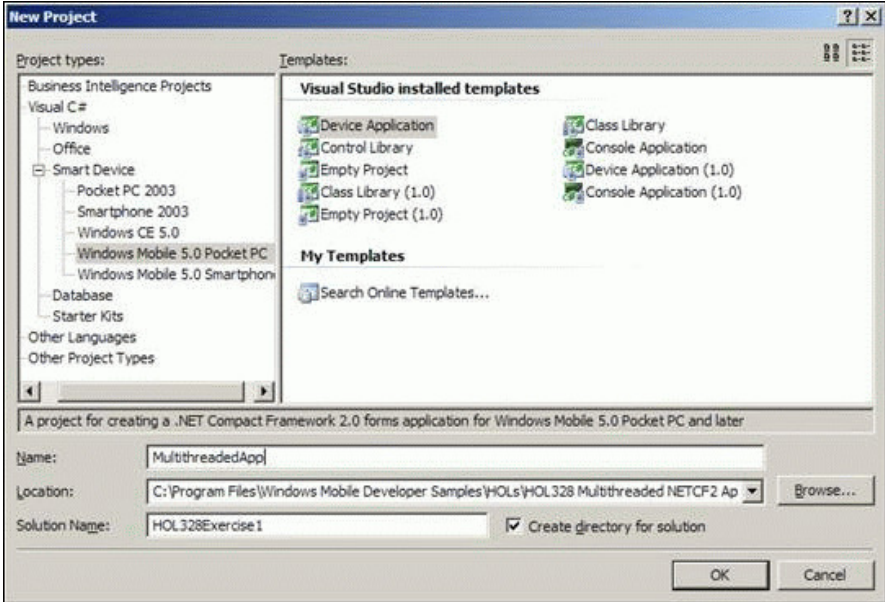
MEDC

Exercise 1

Creating a Single-Threaded Application by Using the .NET Compact Framework 2.0

Scenario

In this exercise, you will build a single-threaded Windows Mobile 5.0–based application that you will transform into a multithreaded application in Exercise 2.

Tasks	Detailed Steps
1. Create a project	a. On the desktop computer, click Start All Programs Microsoft Visual Studio 2005 Microsoft Visual Studio 2005. b. In Visual Studio 2005, click File New Project. The New Project dialog box appears. c. Under Project types, ensure that Visual C# Projects Smart Device Windows Mobile 5.0 Pocket PC is selected. d. Under Templates, ensure that Device Application is selected. e. In the Name box, type MultithreadedApp. f. In the Location box, type C:\Program Files\Windows Mobile Developer Samples\HOLs\MEDC06_HOL328. g. Be sure that the Create directory for solution check box is selected, as shown in Figure 1.
	 Figure 1. New Project dialog box h. Click OK to create an empty project. Note: Even though the application that you just created does not do much, it is a complete Windows Mobile 5.0 application, so you can compile it and deploy it to a target device. For all exercises in this lab, you will use the Windows Mobile 5.0

Tasks	Detailed Steps
	<i>Pocket PC emulator.</i> Note: In the following task, you will deploy this first application to the emulator, so you can test the connection to the emulator.
2. Build the project and deploy it to the emulator	a. On the Device toolbar, verify that Windows Mobile 5.0 Pocket PC Emulator is selected as target device. b. On the Device toolbar, click the Connect to Device button to set up a connection to the Windows Mobile 5.0 Pocket PC Emulator. c. You will see the Pocket PC emulator starting. When a connection with Visual Studio 2005 is established, a confirmation message appears in the Connecting dialog box. Now, you are ready to deploy and run your application on the Pocket PC emulator. Note: If the connection to the emulator initially fails because the device is not ready, click the Close button, and then click again on the Connect to Device button. d. Close the Connecting dialog box by clicking Close. e. Start the application in debug mode by pressing F5 or by clicking Debug Start Debugging. f. In the Deploy MultithreadedApp dialog box, be sure that the Windows Mobile 5.0 Pocket PC Emulator device is selected, and then click Deploy, as shown in Figure 2. Figure 2. Deploy MultithreadedApp dialog box Note: The status bar in the Visual Studio 2005 integrated development environment (IDE) informs you about the progress of deployment. g. Be sure that the emulator is visible by clicking its button on the Windows taskbar. Note: After a while, you will see the empty application running inside the Windows Mobile 5.0 Pocket PC Emulator. This application is not of much use right now, so, in the next task, you will add some functionality to it.
3. Add functionality to the application	a. Press SHIFT+F5 or click Debug Stop Debugging to close the application from within Visual Studio 2005. b. Change the title of the application to Multithreaded App by clicking somewhere in the form and changing the Text property. If the Properties pane is not visible, you can show it by clicking View Properties Window. c. To make it easier to close the application, change the Minimize Box property of the form to False. In this way, it is possible to close the application instead of smart minimizing it.

Tasks	Detailed Steps
	<i>Note: Now, you are ready to add functionality to the application. For this first exercise, you are going to add two buttons to the form and a method that simulates background processing.</i> d. In the Toolbox, select a Button control from the All Device Controls group, and then drag the control to the form. e. Click the button, and then change the caption of the button to Begin processing by changing its Text property. You may resize the button to accommodate this text by clicking and dragging one of its handles. f. Double-click the Begin processing button in the form to add a click event handler. g. Add the following code to the button1_Click event handler. <pre>button1.Enabled = false; processingDone = false; BackgroundProcessing();</pre> h. In the form designer in Visual Studio, select another Button control in the Toolbox, and then drag it to the form. i. Click the button, and then change the caption of the button to End processing by changing its Text property. j. Double-click the End processing button in the form to add a click event handler. k. Add the following code to the button2_Click event handler. <pre>processingDone = true; button1.Enabled = true;</pre> l. Create an alias for the "System.Threading" namespace by adding the following statement immediately under the other using statements at the beginning of the Form1.cs file (this is the file into which all code in this exercise will be added). <pre>using System.Threading;</pre> m. Add a Boolean instance variable called processingDone to the Form1 class by adding the following code. <pre>private bool processingDone = false;</pre> n. Add a new method called BackgroundProcessing to the Form1 class by adding the following code. <pre>private void BackgroundProcessing() { while (! processingDone) { Thread.Sleep(0); } }</pre> o. Start the application in debug mode by pressing F5 or by clicking Debug Start Debugging. p. In the Deploy MultithreadedApp dialog box, be sure that the Windows Mobile 5.0 Pocket PC Emulator device is selected, and then click Deploy. q. Be sure that the emulator is visible by clicking its button on the Windows taskbar. <i>Note: If you didn't have any syntax errors after compiling the code, the application</i>

Tasks	Detailed Steps
	<i>appears in the Windows Mobile 5.0 Pocket PC emulator.</i> r. In the emulator, click the Begin processing button of the application. Note: You will see that the Begin processing button is disabled. There is no further indication that the BackgroundProcessing function is running. s. In the emulator, click the End processing button of the application. Note: You might expect the Begin processing button to be enabled again, but that does not happen. In fact, the application stops responding. Because all code for this application runs inside the same thread, a problem occurs. After the code in the BackgroundProcessing method is executed, the button2_Click event handler cannot be called. The application does not respond to your button clicks, resulting in the processingDone variable remaining false. In other words, the application will not close properly, as shown in Figure 3. Figure 3. The application is running, but it can't be stopped Note: To solve this problem, you will change the application so it will run the BackgroundProcessing method in a separate thread. t. Press SHIFT+F5 or click Debug Stop Debugging to close the running application from within Visual Studio 2005. You can also use the Stop button on the Debug toolbar of Visual Studio 2005 if it is visible.

Exercise 2

Modifying the Multithreaded Application

Scenario

In this exercise, you will modify the application that you created in Exercise 1. You will create a new thread in which the **BackgroundProcessing** method will execute. Creating a thread in the .NET Compact Framework is simply a matter of instantiating a class of type **Thread** and passing a reference to a function to the class. The function is the entry point of the new thread, and after the function exits, the thread automatically is terminated.

Tasks	Detailed Steps
1. Create a separate thread in which BackgroundProcessing executes	a. Add an instance variable of type Thread called workerThread to the Form1 class immediately under the processingDone variable by adding the following code. <pre>private Thread workerThread;</pre> b. Locate the method call to BackgroundProcessing() inside the button1_Click event handler, and then replace it by using the following statements. <pre>workerThread = new Thread(BackgroundProcessing); workerThread.Start();</pre> <i>Note: You have just used a new C# 2.0 feature called delegate inference. Delegate inference allows you to make a direct assignment of a method name to a delegate variable—without wrapping it first with a delegate object. In C# 1.0 you would create your worker thread by using the following statement:</i> <pre>workerThread = new Thread(new ThreadStart(BackgroundProcessing));</pre> <i>Note: As you can see, the way to create a new thread object in C# 2.0 is much more intuitive.</i> <i>Note: That is all you need to do to have the BackgroundProcessing method running in a different thread. It is important to call the Start method of the thread after creation; otherwise, the thread will not run. Now, it is time to run the application again and to notice the difference from the first version you ran.</i> c. Start the application in debugging mode by pressing F5 or by clicking Debug Start Debugging. d. In the Deploy MultithreadedApp dialog box, be sure that the Windows Mobile 5.0 Pocket PC Emulator device is selected, and then click Deploy. <i>Note: If you didn't have any syntax errors after compiling the code, the application appears in the Windows Mobile 5.0 Pocket PC Emulator, which you can make visible by clicking the Windows taskbar button.</i> e. In the emulator, click the Begin processing button of the application. <i>Note: The Begin processing button is disabled. There is no further indication that the BackgroundProcessing function is running.</i> f. In the emulator, click the End processing button of the application. <i>Note: The Begin processing button is enabled again, effectively meaning that the thread on which the BackgroundProcessing method is executing has terminated. So</i>

Tasks	Detailed Steps
	this time, you remain in control of the application. Leave the application running in the emulator for the moment. Note: In the .NET Compact Framework 2.0, there is a distinction between foreground threads and background threads. You will experience the difference in the behavior of both. To do so, you need to start the worker thread in the application again.
2. Properly terminate created threads	a. In the emulator, click the Begin processing button of the application. b. In the upper-right corner of the application, click ok. Note: It seems that the application is closing because the application's form is closed. However, if you look at Visual Studio, you might notice that the application is still running because the debugger is still active. The reason is that the thread you have created is a foreground thread. As long as a foreground thread is still running in an application, the application will not close—even though the form has been closed. To properly close the application, you will need to add some extra functionality. It is, for instance, possible to add a form-closing event handler that checks whether the worker thread is still active. In that case, a message can be given to the user to first terminate the worker thread. c. Press SHIFT+F5 or click Debug Stop Debugging to close the running application from within Visual Studio 2005. You can also click the Stop button on the Debug toolbar of Visual Studio 2005 if it is visible. d. In Visual Studio 2005, click the Form1.cs [Design] tab, and then click somewhere on the form outside the buttons. e. In the Properties pane, click the Events button. f. Double-click the Closing event, as shown in Figure 4. You now have created a Form1_Closing event handler. Figure 4. Adding a form closing event handler g. In the Form1_Closing event handler, add the following code. <pre> if (!processingDone) { MessageBox.Show(</pre>

Tasks	Detailed Steps
	<pre data-bbox="544 193 1398 317">"Terminate the worker thread before closing the application"); e.Cancel = true; }</pre> h. Press F5 or click Debug Start Debugging to start the application in debugging mode. i. In the Deploy MultithreadedApp dialog box, be sure that the Windows Mobile 5.0 Pocket PC Emulator device is selected, and then click Deploy. j. In the emulator, click the Begin processing button of the application. k. In the upper-right corner of the application, click ok. <i>Note: A message informs you that the BackgroundProcessing thread is still running and indicates that the application's form will not be closed unless the BackgroundProcessing thread is terminated.</i> l. Click the End Processing button to close the application, and then click ok in the upper-right corner of the application. m. In the .NET Compact Framework 2.0, you can change the worker thread to a background thread by setting the IsBackground property of the Thread class to true. The common language runtime automatically terminates threads as soon as the last foreground thread that belongs to the same process ends. To see the behavior of a background thread, you need to remove the Form1_Closing event handler that you just created. n. In Visual Studio 2005, click the Form1.cs [Design] tab, and then click somewhere on the form outside the buttons. o. In the Properties pane, click the Events button. p. Right-click the Closing event, and then click Reset. This action causes the event handler to be detached from the form. q. The code for the event handler might still be in your source file. If that is the case, simply remove the entire Form1_Closing method from the Form1.cs source file. r. Locate the button1_Click event handler inside Form1.cs, and then add the following statement between the statements that create and start the thread. <pre data-bbox="544 1281 1019 1310">workerThread.IsBackground = true;</pre> s. Press F5 or click Debug Start Debugging to start the application in debugging mode. t. In the Deploy MultithreadedApp dialog box, be sure that the Windows Mobile 5.0 Pocket PC Emulator is selected, and then click Deploy. u. In the emulator, click the Begin processing button of the application. v. In the upper-right corner of the application, click ok. <i>Note: This time, the application closes properly, even if the BackgroundProcessing thread is active.</i> <i>Note: In some scenarios, it is absolutely necessary for an application to know when a thread has been terminated. For those situations, the Thread.Join method is available in the .NET Compact Framework 2.0. This method blocks the calling thread until a thread terminates. To make the behavior of Thread.Join clear, you are going to add some time-consuming termination code to the worker thread.</i>
3. Find out if a worker thread has really terminated	a. Locate the BackgroundProcessing method in the Form1.cs source file. Immediately under the while loop, add the following statement. <pre data-bbox="544 1883 818 1913">Thread.Sleep(2000);</pre>

Tasks	Detailed Steps
	b. Locate the button2_Click event handler. Immediately under the statement processingDone = true, add the following statement. <pre>workerThread.Join();</pre> c. Compile, deploy, and then run the application. d. Start the worker thread by clicking the Begin processing button, and then terminate the thread again by clicking the End processing button. <i>Note: You may have noticed that it takes a few seconds for the Begin processing button to be re-enabled after you click the End processing button. The reason is that the main thread now blocks processing in the button2_Click event handler until the worker thread terminates. Because you have added an extra Thread.Sleep of two seconds in the worker thread, the worker thread takes an additional two seconds to terminate.</i> e. In the upper-right corner of the application, click ok to close the application. <i>Note: So far, you have used a Boolean flag in combination with a while loop to stop the worker thread. In the .NET Compact Framework 2.0, there is an alternative way to terminate worker threads: You can use the Thread.Abort method in combination with an exception handler. In cases where some thread deinitialization code must always be executed (for example, closing a connection to a database), Thread.Abort can be useful. Calling Thread.Abort raises a ThreadAbortException in the thread on which it is invoked, to begin the process of terminating the thread.</i>
4. Terminate a thread by using Thread.Abort	a. Locate the statement processingDone = true in the button2_Click event handler in the Form1.cs source file, and then replace it with the following statement. <pre>workerThread.Abort();</pre> b. Replace all code of the BackgroundProcessing method by using the following code (you can type this code or copy the code from this document and paste it into Form1.cs). <pre>private void BackgroundProcessing() { try { while (!processingDone) { Thread.Sleep(0); } } catch (ThreadAbortException e) { MessageBox.Show(e.Message); } finally { // This deinitialization code must always be // executed, // so be sure to put it in a finally clause. Thread.Sleep(2000); } }</pre> c. Compile, deploy, and then run the application.

Tasks	Detailed Steps
	d. Start the worker thread by clicking the Begin processing button, and then terminate the thread again by clicking the End processing button. <i>Note: You will see that a ThreadAbortException has been raised and that the worker thread terminates after handling the exception.</i> <i>Note: Because exception handling is a relatively expensive operation in managed code, in normal situations, it might be better to use a Boolean variable in combination with a <code>while</code> loop to properly terminate worker threads. Also, Thread.Abort might not function correctly if your worker thread is platform invoking into native code where it is waiting for a native thread synchronization object.</i> e. In the upper-right corner of the application, click ok to close the application. <i>Note: In a multithreaded operating system like Windows CE, threads can have different priorities. Assigning different priorities to different threads can dramatically influence the behavior of your application. Threads with a high priority will always run before threads with a lower priority. When multiple threads of the same priority exist in Windows CE, they will share the processor time equally. To see the impact of changing a thread priority, you will give your worker thread a higher priority than the main thread. First, though, you are going to modify the thread method itself so that it continuously uses the processor</i> <i>Note: Never perform this procedure inside a real application.</i>
5. Use thread priorities	a. Locate the statement <code>Thread.Sleep(0)</code> in the BackgroundProcessing method in the <code>Form1.cs</code> source file, and then replace it with an empty statement (just a semicolon). b. Compile, deploy, and then run the application. c. Start the worker thread by clicking the Begin processing button, and then terminate the thread again by clicking the End processing button. d. In the upper-right corner of the application, click ok to close the application. <i>Note: Even though the worker thread wants to run continuously now, you can still terminate it because threads with the same priority share the processor equally. But now, watch what happens if you change the worker thread priority.</i> e. Locate the statement <code>workerThread.Start()</code> in the button1_Click event handler in the <code>Form1.cs</code> source file, and then add the following statement above it. <pre>workerThread.Priority = ThreadPriority.AboveNormal;</pre> f. Compile, deploy, and then run the application. g. Start the worker thread by clicking the Begin processing button, and then terminate the thread again by clicking the End processing button. <i>Note: As you can see, it is not possible anymore to terminate the worker thread—and even more dramatically, it is impossible to close the application. The reason is that a thread with a higher priority is now continuously running, not allowing the main thread to execute. Because the main thread is responsible for all of the user interface (UI) actions, the user can no longer control the application. The only way to close the application is to stop debugging from inside Visual Studio 2005.</i> h. Press SHIFT+F5 or click Debug Stop Debugging to close the running application from within Visual Studio 2005. You can also use the Stop button on the Debug toolbar of Visual Studio 2005 if it is visible. i. Close Visual Studio 2005. <i>Note: So far, you have created a multithreaded application, seen the difference between foreground and background threads, and learned how to properly terminate threads and multithreaded applications. You have also seen the impact that changing thread priorities can have. In the next exercise, you will compare Thread objects with</i>

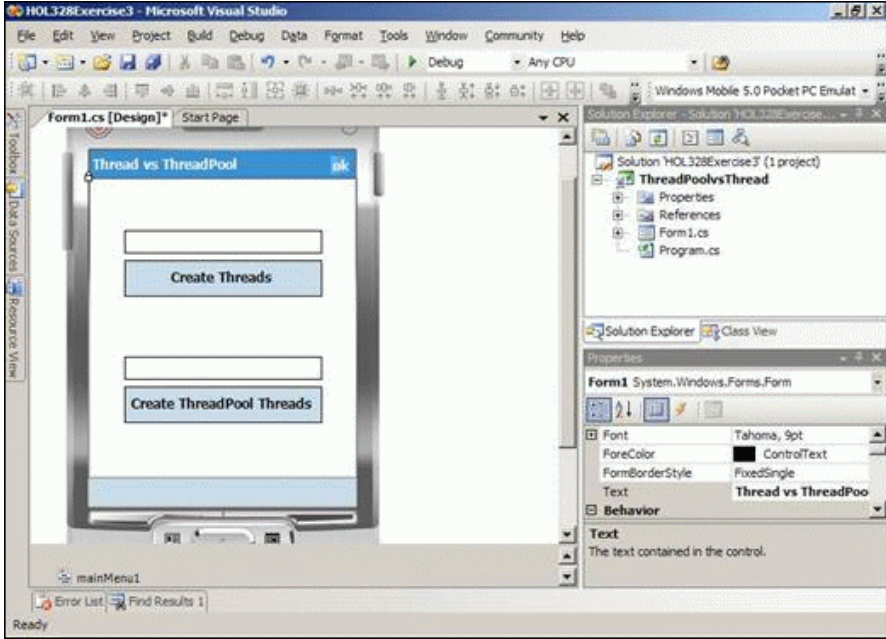
Tasks	Detailed Steps
	<i>ThreadPool objects.</i>

Exercise 3

Examining Thread and ThreadPool

Scenario

In this exercise, you will create an application, again using the Windows Mobile 5.0 Pocket PC Emulator, to see the difference between using Threads and ThreadPool threads. For this exercise, you will use an already created solution that contains the user interface for the application that you are going to build.

Tasks	Detailed Steps
1. Open the ThreadPool versus the Thread solution	a. Open Visual Studio 2005. b. In Visual Studio 2005, click File Open Project/Solution. c. In the Open Project dialog box, browse to C:\Program Files\Windows Mobile Developer Samples\HOLs\HOL328_Multithreaded_NETCF2\HOL328Exercise3. d. Select HOL328Exercise3.sln, and then click Open to open the solution in Visual Studio 2005. e. Open the Form1.cs file in the form designer mode by double clicking on it in Solution Explorer. <i>Note: To save time, the user interface is already created for you. It should look like Figure 5.</i>  Figure 5. User interface for the ThreadvsThreadPool application f. Double-click both buttons in the form designer to add click event handlers. You have to switch back to the form designer after you add the first click event handler

Tasks	Detailed Steps
	to be able to add the second click event handler.
2. Add functionality to the ThreadPoolsThread application	a. Create an alias for the "System.Threading" namespace by adding the following statement immediately under the other <code>using</code> statements at the beginning of the <code>Form1.cs</code> file (this is the file into which all code in this exercise will be added). <pre>using System.Threading;</pre> b. Add the following instance variables to the Form1 class by adding the following code. <pre>private AutoResetEvent doneEvent; private int threadCounter; private int threadPoolCounter;</pre> c. Add the following code to the button1_Click event handler, assuming the first button you double-clicked to create an event handler was the one labeled Create Threads. <pre>button1.Enabled = false; threadCounter = 0; doneEvent = new AutoResetEvent(false); textBox1.Text = ""; int elapsedTime = Environment.TickCount; for (int i = 0; i < 200; i++) { Thread workerThread = new Thread(MyWorkerThread); workerThread.Start(); } doneEvent.WaitOne(); elapsedTime = Environment.TickCount - elapsedTime; textBox1.Text = "Creating threads: " + elapsedTime.ToString() + " msec"; button1.Enabled = true;</pre> <i>Note: The code you just added to the button1_Click event handler creates 200 different worker threads that all have a very short lifetime. In fact, the only functionality of all worker threads is to check if a particular thread is the two hundredth. In that case, an event is set to inform the main thread that the test run is complete, so the main thread can update the user interface with timing information. All worker threads share the same functionality that exists inside the MyWorkerThread method that you will add.</i> d. Add a new MyWorkerThread method to the Form1 class by using the following code. <pre>private void MyWorkerThread() { Interlocked.Increment(ref threadCounter); if (threadCounter == 200) doneEvent.Set(); }</pre> <i>Note: The working of Interlocked is explained in detail in Exercise 5 of this lab.</i> e. Add the following code to the button2_Click event handler, assuming the first button you double-clicked to create an event handler was the one labeled Create Threads.

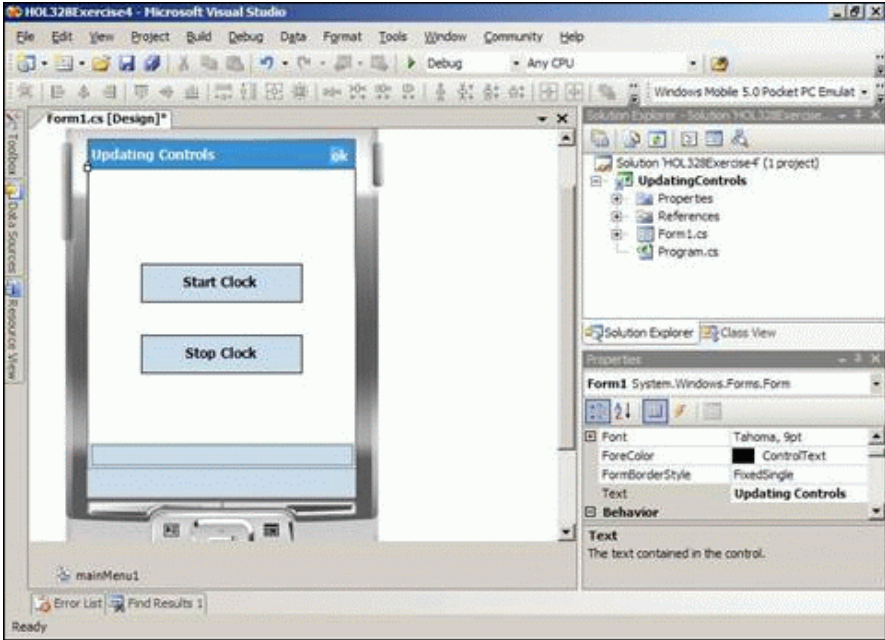
Tasks	Detailed Steps
	<pre data-bbox="544 233 1263 674"> button2.Enabled = false; threadPoolCounter = 0; doneEvent = new AutoResetEvent(false); textBox2.Text = ""; int elapsedTime = Environment.TickCount; for (int i = 0; i < 200; i++) { ThreadPool.QueueUserWorkItem(MyWaitCallback); } doneEvent.WaitOne(); elapsedTime = Environment.TickCount - elapsedTime; textBox2.Text = "Creating threads: " + elapsedTime.ToString() + " msec"; button2.Enabled = true; </pre> Note: The code of the button2_Click event handler is almost identical to the code of the button1_Click event handler, with one important exception. This time, inside the for loop, no new threads are created, but instead a callback method is added to the ThreadPool class. ThreadPool is in fact a collection of reusable threads. Using ThreadPool eliminates the overhead of creating a new thread class and setting up its resources to be able to blend in seamlessly with the operating system. Because creating a worker thread is a relatively expensive operation, the difference in performance is dramatic. Note: Because ThreadPool threads are shared resources, you should not run long-lasting threads inside them. After all, if no thread is available in ThreadPool when the QueueUserWorkItem method is called, the specified delegate executes only after a thread becomes available. Because a ThreadPool thread is a ready-to-run real foreground thread, you need to properly terminate it before the application closes (for more information, see "To properly terminate created threads" in Exercise 2). f. Add a new MyWaitCallback method to the Form1 class by using the following code. <pre data-bbox="544 1241 1250 1430"> private void MyWaitCallback(object stateInfo) { Interlocked.Increment(ref threadPoolCounter); if (threadPoolCounter == 200) doneEvent.Set(); } </pre> Note: This method executes as a separate thread, but the ThreadPool object creates the thread for you or reuses an existing thread. When a thread is available in the ThreadPool, it immediately starts running your method, so there is no need to explicitly start the thread. g. Compile, deploy, and then run the application. h. Click the two buttons, waiting for each to complete its process, and compare the difference in timing between creating and running 200 threads to using 200 ThreadPool threads. Note: Creating 200 threads might take up to a minute or more of execution time. i. In the upper-right corner of the application, click ok to close the application. j. Close Visual Studio 2005.

Exercise 4

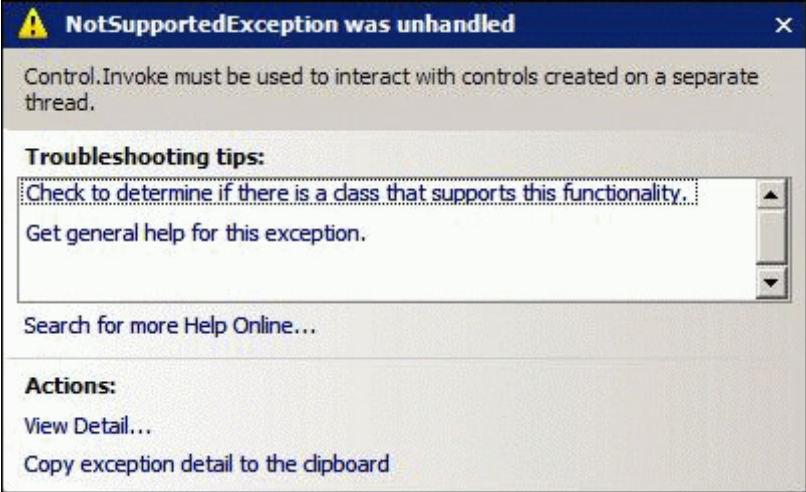
Updating User Interface Controls Inside Threads

Scenario

A common mistake that many developers make is trying to update or access user interface controls directly from within worker threads. This action results in unexpected behavior; frequently, the application stops responding. To see this problem in action, you are going to use yet another Windows Mobile 5.0 application to update a user interface control the wrong way. Later, you will modify the code so the application will work properly.

Tasks	Detailed Steps
1. Open the UpdatingControls solution	a. Open Visual Studio 2005. b. In Visual Studio 2005, click File Open Project/Solution. c. In the Open Project dialog box, browse to C:\Program Files\Windows Mobile Developer Samples\HOLs\HOL328_Multithreaded_NETCF2\HOL328Exercise4. d. Select HOL328Exercise4.sln, and then click Open to open the solution in Visual Studio 2005. e. Open the Form1.cs file in the form designer by double clicking it in Solution Explorer. <i>Note: To save time, the user interface is already created for you. It should look like Figure 6.</i>  Figure 6. User interface for the UpdatingControls application f. Double-click both buttons in the form designer to add click event handlers. You have to switch back to the form designer after you add the first click event handler to add the second click event handler. <i>Note: In the next task, you will add functionality to the application to display the</i>

Tasks	Detailed Steps
	<i>current time continuously on the status bar.</i>
2. Add functionality to the UpdatingControls application	a. Create an alias for the "System.Threading" namespace by adding the following statement immediately under the other <code>using</code> statements at the beginning of the Form1.cs file. <pre>using System.Threading;</pre> b. Add the following instance variables to the Form1 class. <pre>private Thread myThread; private bool workerThreadDone;</pre> c. Add the following code to the button1_Click event handler. <pre>button1.Enabled = false; button2.Enabled = true; statusBar1.Text = ""; workerThreadDone = false; myThread = new Thread(MyWorkerThread); myThread.Start();</pre> <i>Note: In the code you just added to button1_Click, you have instantiated and started a new worker thread. You will add the functionality of the worker thread itself later in this lab.</i> d. Add the following code to the button2_Click event handler. <pre>workerThreadDone = true; button2.Enabled = false; button1.Enabled = true;</pre> <i>Note: The code you just added is responsible for terminating the worker thread. It also re-enables the Start Clock button to allow running the clock again.</i> e. Add a new MyWorkerThread method to the Form1 class by using the following code. <pre>private void MyWorkerThread() { while (!workerThreadDone) { statusBar1.Text = DateTime.Now.ToLongDateString() + " - " + DateTime.Now.ToLongTimeString(); Thread.Sleep(0); } }</pre> f. Compile, deploy, and then run the application. g. Click the Start Clock button. <i>Note: A few seconds after you click the Start Clock button, the application throws an exception, as shown in Figure 7. The exception is thrown because you tried to update a user interface control from within a thread that did not create the user interface control.</i>

Tasks	Detailed Steps
	 Figure 7. The application throws an exception h. Press SHIFT+F5 or click Debug Stop Debugging to stop the debugger within Visual Studio 2005. You can also use the Stop button on the Debug toolbar of Visual Studio 2005 if it is visible. <i>Note: Even though an exception occurred, this is a huge improvement over the behavior of the .NET Compact Framework version 1.0, where the application might stop responding for no apparent reason. To solve the problem, commit yourself to the following rule: Only the thread that creates a UI control can safely update that control.</i> <i>Note: If you need to update a control inside a worker thread, you should always use the Control.Invoke method. This method executes a specified delegate on the thread that owns the control's underlying window handle, in other words, the thread that created the control. The .NET Compact Framework 1.0 supported only synchronous updates of user interface controls from within worker threads. The .NET Compact Framework 2.0, however, supports asynchronous updates of user interface controls. Another limitation of the .NET Compact Framework 1.0 was the lack of support to pass parameters by using Control.Invoke. With version 2.0, it is possible to pass parameters by using Control.Invoke. In the next procedure, you will explore synchronous updates of user interface controls inside worker threads.</i>
3. Update controls by using Control.Invoke	a. Locate the declaration of the myThread instance variable in the Form1.cs source file, and then add the following code to declare a delegate with the correct signature for updating the status bar. <pre>private delegate void UpdateTime(string dateTimeString);</pre> b. Scroll to the end of the Form1.cs source file, and then add the following method (a method with exactly the same signature as the delegate that actually updates the status bar) to the Form1 class. <pre>private void UpdateTimeMethod(string dateTimeString) { statusBar1.Text = dateTimeString; }</pre> c. Modify the MyWorkerThread method as follows so it calls the Invoke method instead of directly updating the status bar itself. <pre>private void MyWorkerThread()</pre>

Tasks	Detailed Steps
	<pre data-bbox="544 195 1372 604"> { UpdateTime timeUpdater = UpdateTimeMethod; string currentTime; while (!workerThreadDone) { currentTime = DateTime.Now.ToLongDateString() + " - " + DateTime.Now.ToLongTimeString(); this.Invoke(timeUpdater, new object[] {currentTime}); Thread.Sleep(0); } } </pre> Note: As you can see in the modified code, a local string inside MyWorkerThread is set to the current date/time value. The Invoke method of Form1 is called to update the StatusBar control on behalf of MyWorkerThread, but it runs in the context of the main thread (the thread that created the StatusBar control). d. Compile, deploy, and then run the application. e. Click the Start Clock button. The status bar is now continuously updated with date/time information. f. Stop the worker thread by clicking the Stop Clock button, and then close the application by clicking ok. Note: To determine when the worker thread terminates, you need to add a Thread.Join method. You then can, for instance, inform the user on the status bar about worker thread termination. Note: If you want to learn more about the Thread.Join method, see Exercise 2 in this lab.
4. Synchronously update the user interface and understand the risk of deadlock	a. Locate the button2_Click event handler in the Form1.cs file, and then add the following statement immediately after workerThreadDone = true. <pre data-bbox="544 1213 776 1245">myThread.Join();</pre> b. Inside MyWorkerThread, add the following statements at the end of the method. <pre data-bbox="544 1329 1323 1388">string statusInfo = "MyWorkerThread terminated!"; this.Invoke(timeUpdater, new object[] { statusInfo });</pre> c. Compile, deploy, and then run the application. d. Click the Start Clock button. The status bar is now continuously updated with date/time information. e. Try to stop the worker thread by clicking the Stop Clock button. Note: When you click the Stop Clock button, the timer stops, but you don't see the message, "MyWorkerThread terminated!" on the status bar. Even worse, the application stops responding. The fact that you added the myThread.Join() statement in the button2_Click event handler blocks the main thread until the worker thread has terminated. Because you also set workerThreadDone = true in the same event handler, the worker thread continues running after the while loop, where it wants to update the status bar one more time. Note: Remember that Control.Invoke executes on the main thread. However, the main thread is blocked because it is waiting for the worker thread to terminate, so Control.Invoke cannot execute. Because Control.Invoke is a synchronous operation, it returns only when the delegate passed by Control.Invoke finishes executing. There is

Tasks	Detailed Steps
	<i>no way for the worker thread to continue.</i> Note: This is a typical deadlock situation. To solve this problem, you can update user interface controls by using the asynchronous way that is available in the .NET Compact Framework 2.0. f. Press SHIFT+F5 or click Debug Stop Debugging to close the application from within Visual Studio 2005. You can also use the Stop button on the Debug toolbar of Visual Studio 2005 if it is visible.
5. Asynchronously update user interface controls inside worker threads	a. Locate the <code>this.Invoke(timeUpdater, new object[] { statusInfo });</code> statement in the MyWorkerThread method, and then replace it with the following statement. <pre data-bbox="545 548 1398 611">this.BeginInvoke(timeUpdater, new object[] { statusInfo });</pre> Note: This is the asynchronous version of Control.Invoke. On calling this method, the worker thread continues to run. The main thread updates the control on behalf of the worker thread only when a thread switch occurs. Most often, Control.BeginInvoke is used in combination with Control.EndInvoke. The latter method retrieves the return value of the asynchronous operation represented by the IAsyncResult object passed. However, in this application, you have to ignore the result of the asynchronous operation because the worker thread is finished before the main thread can return the result of the operation. Even worse, adding Control.EndInvoke to the worker thread might again result in a deadlock situation or in an ObjectDisposedException. b. Compile, deploy, and then run the application. c. Click the Start Clock button, and then wait until the clock is updating information on the status bar. d. Stop the worker thread by clicking the Stop Clock button. Note: This time, the worker thread is terminated and the message, “MyWorkerThread terminated!” appears on the status bar. e. In the upper-right corner of the form, click ok to close the application. Note: You have created a multithreaded application; you have seen the difference between Thread and ThreadPool; and you also have learned how to update user interface controls from within worker threads. In the last exercise of this lab, you will learn how to synchronize different threads by using various synchronization objects. f. Close Visual Studio 2005.

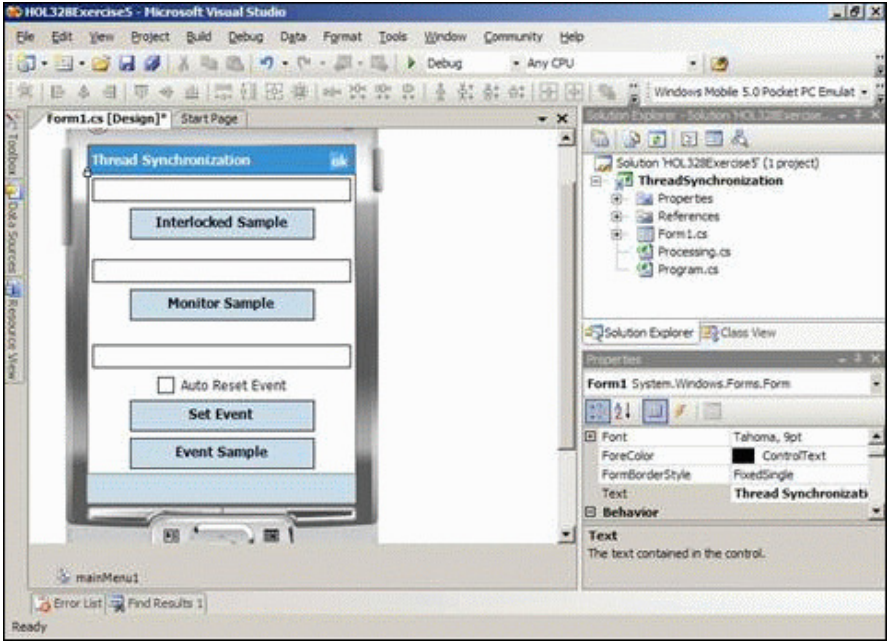
Exercise 5

Using Synchronization Objects to Synchronize Threads

Scenario

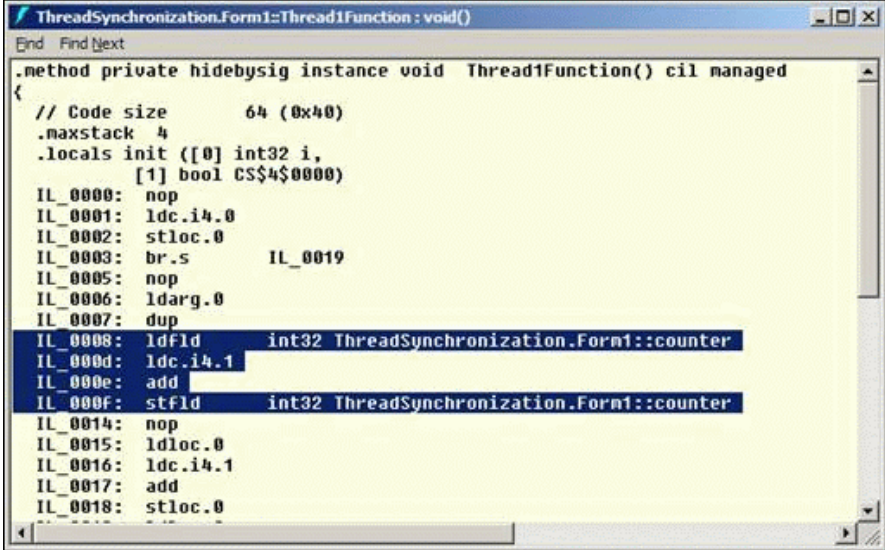
Even though the Windows CE scheduler is responsible for scheduling threads, it seems that all threads in a multithreaded application run autonomously unless special attention is paid to thread synchronization. Especially in cases where multiple threads access shared data, it is absolutely necessary to synchronize access of that data.

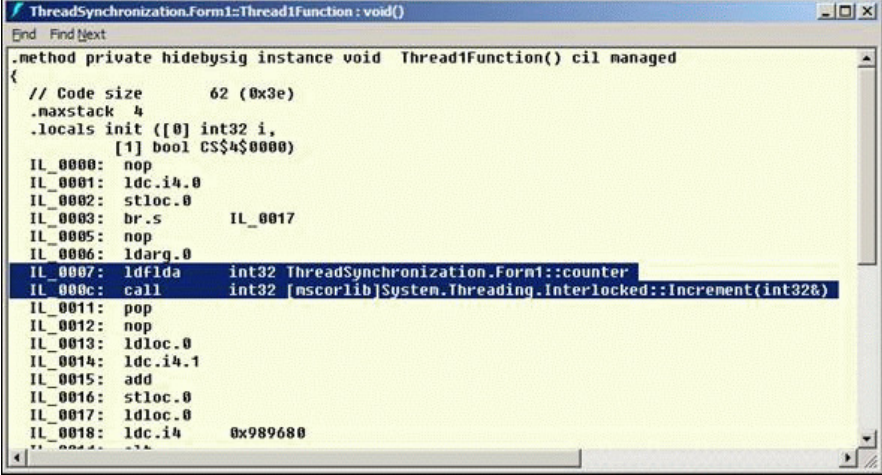
Thread synchronization is a very important subject. To cover as much as possible, this exercise is longer than the previous exercises. You should, therefore, copy and paste most of the code from this document into the application that you are going to create.

Tasks	Detailed Steps
1. Open the ThreadSynchronizati on solution	a. Open Visual Studio 2005. b. In Visual Studio 2005, click File Open Project/Solution. c. In the Open Project dialog box, browse to C:\Program Files\Windows Mobile Developer Samples\HOLs\HOL328_Multithreaded_NETCF2\HOL328Exercise5. d. Select HOL328Exercise5.sln, and then click Open to open the solution in Visual Studio 2005. e. Open the Form1.cs file in the form designer by double clicking it in Solution Explorer. <i>Note: To save time, the user interface is already created for you. It should look like Figure 8.</i>  Figure 8. User interface of the ThreadSynchronization application f. Double-click each button in the form designer to add click event handlers. Start from the top button and work your way to the bottom button. You have to switch

Tasks	Detailed Steps
	back to the form designer after you add each click event handler. <i>Note: In this part of Exercise 5, you will see the need for synchronization between threads. The application starts two different worker threads. Both worker threads access the same variable. One thread increments the variable, whereas the other thread decrements it. Each thread will loop 10 million times. When both worker threads have finished executing, the value of the variable should be zero. The end result is shown in the read-only text box.</i>
2. Use the Interlocked class	a. Create an alias for the "System.Threading" namespace by adding the following statement immediately under the other <code>using</code> statements at the beginning of the Form1.cs file. <pre>using System.Threading;</pre> b. Add the following instance variables to the Form1 class by adding the following code. <pre>private int counter; private bool thread1Running; private bool thread2Running;</pre> c. Add the following code to the button1_Click event handler. <pre>textBox1.Text = "Worker Threads started"; button1.Enabled = false; counter = 0; Thread workerThread1 = new Thread(Thread1Function); Thread workerThread2 = new Thread(Thread2Function); thread1Running = true; thread2Running = true; workerThread1.Start(); workerThread2.Start();</pre> d. Create methods for both worker threads and a method that indicates proper thread termination of both threads by adding the following code. <pre>// Worker thread that repeatedly updates an instance // variable private void Thread1Function() { for (int i = 0; i < 10000000; i++) { counter++; } thread1Running = false; this.Invoke(new EventHandler(WorkerThreadsFinished)); } // Worker thread that repeatedly updates the same instance // variable private void Thread2Function() { for (int i = 0; i < 10000000; i++) { counter--; } }</pre>

Tasks	Detailed Steps
	<pre data-bbox="544 195 1365 793"> } thread2Running = false; this.Invoke(new EventHandler(WorkerThreadsFinished)); } // Delegate that is called whenever one of the worker // threads is finished. // If both are finished, the processing result is // presented to the user. private void WorkerThreadsFinished(object sender, System.EventArgs e) { if (!thread1Running && !thread2Running) { button1.Enabled = true; textBox1.Text = "The value of counter = " + counter.ToString(); } } </pre> Note: You should note the way both worker threads indicate that they are finished: They use Control.Invoke to call a delegate. The reason to use Control.Invoke is that both the button and the text box are updated after both threads are finished, and the threads themselves update the user interface controls. e. Compile, deploy, and then run the application. f. Click the Interlocked Sample button. After a while, you will see the result in the text box. Note: Most of the time, the end result after running both threads will be zero. But if you try running the Interlocked sample often enough, the end result sometimes is a random number. Even if you don't see unexpected results when testing the application, you should be aware of the potential danger of simultaneously updating variables in more than one thread. The reason is that each of the threads is running with the same priority. They both get equal amounts of processor time and execute in a round-robin fashion. Note: Even though accessing the counter variable in the worker threads seems to be a single statement (<code>counter++</code> or <code>counter--</code>), if you look in the generated Microsoft intermediate language (MSIL) in Figure 9, you will see that this single C# statement consists of several lines of MSIL code. The highlighted lines of MSIL code are the ones that help explain the problem that you might experience. If both threads are running for a longer period, one might be scheduled out while having read the current value of the counter. The other worker thread is scheduled in by that time, reading the counter as well, and then modifying it and storing the updated value of the counter back. When the first worker thread resumes running, it continues exactly where it stopped; it does not read the counter again, but simply increments the original value and stores it back into memory. With that, all of the changes that the other worker threads made to the counter are destroyed, which results in unexpected values of the counter after both worker threads terminate.

Tasks	Detailed Steps
	 Figure 9. Generated intermediate language for Thread1Function <i>Note: To prevent against the potential problem in the previous situation, you should protect the data that both worker threads are accessing. In the case of simple increment, decrement, or compare operations, you can protect the data by using the Interlocked class. This class has methods to increment, decrement, and compare object types. The Interlocked class guarantees that the increment or decrement operations are atomic operations because variables are passed by reference, not by value.</i> g. In the upper-right corner of the form, click ok to close the application. h. To use Interlocked, simply change the code of both worker threads. i. Change the increment/decrement operators in both worker threads (Thread1Function and Thread2Function, respectively) to the following. <pre>Interlocked.Increment(ref counter); Interlocked.Decrement(ref counter);</pre> j. Compile, deploy, and then run the application. k. Click the Interlocked Sample button. After a while, you will see the result in the text box, which now consistently is zero. <i>Note: Looking at the generated MSIL in Figure 10, you can see that it is no longer relevant if a thread switch occurs during the increment/decrement operation because Interlocked takes a reference to the variable that needs to be updated, and methods in the Interlocked object cannot be interrupted by other threads. There is however a performance penalty because the counter variable is now passed by reference instead of by value. However, the variable is guaranteed to contain the right value all of the time.</i>

Tasks	Detailed Steps
	 Figure 10. Generated intermediate language for Thread1Function with Interlocked I. In the upper-right corner of the form, click ok to close the application. Note: Imagine the following scenario. The application creates two worker threads, and both worker threads use methods of another class, called Processing. Those methods are updating instance data of the Processing class. Both worker threads have access to all methods in the Processing class. The Processing class has two methods that each update a counter value in a loop. Both functions loop the same number of times. You are now going to extend the application to implement this functionality.
3. Use the Monitor class	a. Open the already existing source file, Processing.cs, in Code view in Visual Studio 2005, and then replace all of the code in it with the following. <pre data-bbox="545 1098 1357 1896">using System; using System.Collections.Generic; using System.Text; using System.Threading; namespace ThreadSynchronization { // Processing class that can be accessed by multiple threads. // This class demonstrates the use of Monitor. public class Processing { private const int nrLoops = 10; private int counter1; private int counter2; // Constructor of the Processing class. public Processing() { counter1 = 0; counter2 = 0; } // Access some data that Monitor protects. public void Function1() </pre>

Tasks	Detailed Steps
	<pre> { // Monitor.Enter(this); for (int i = 0; i < nrLoops; i++) { int localCounter = counter1; Thread.Sleep(0); localCounter++; Thread.Sleep(0); counter1 = localCounter; Thread.Sleep(0); } // Monitor.Exit(this); } // Access some data that Monitor protects. public void Function2() { // Monitor.Enter(this); for (int i = 0; i < nrLoops; i++) { int localCounter = counter2; localCounter++; counter2 = localCounter; } // Monitor.Exit(this); } // Return the difference in the two counter variables // that have been accessed by multiple threads. public int Counter { get { return counter1 - counter2; } } } </pre> Note: The functionality of the Processing class is very simple. It contains two different functions that each increment a variable for a constant number of times. The class also contains a read-only property that returns the difference of both counters. You might also note that you are using a local variable in both Function1 and Function2 to update an instance variable, and that Function1 contains a number of calls to the Thread.Sleep method. This code adds functionality that you would not use in a real application; this code simulates what happens if several threads try to update the same variables independently from each other. Note: In addition, note that the source code of the Processing class contains a number of calls to methods of a class called Monitor. As you can see in the code example, these methods are commented out. Leave these methods commented out for the time being. Note: To use this class, you will create two different worker threads in the Form1.cs source file. Each of these threads calls both functions of the Processing class, but in an opposite order. When the threads have finished running, you will get the final

Tasks	Detailed Steps
	<i>result by calling the Processing.Counter property and showing it in the text box.</i> b. Add the following instance variables in the source file Form1.cs. <pre>private Processing processing; private bool workerThread1Done; private bool workerThread2Done;</pre> c. Add the following code to the button2_Click event handler (this should be the handler for the button labeled Monitor Sample). <pre>button2.Enabled = false; textBox2.Text = "Monitor sample running"; processing = new Processing(); workerThread1Done = false; workerThread2Done = false; Thread thread1 = new Thread(Thread1Monitor); Thread thread2 = new Thread(Thread2Monitor); thread1.Start(); thread2.Start();</pre> d. Add the following code to create methods for both worker threads and a method that indicates proper thread termination of both threads. <pre>private void Thread1Monitor() { processing.Function1(); processing.Function2(); Thread.CurrentThread.Priority = ThreadPriority.AboveNormal; workerThread1Done = true; this.Invoke(new EventHandler(WorkerThreadsDone)); } private void Thread2Monitor() { processing.Function1(); processing.Function2(); Thread.CurrentThread.Priority = ThreadPriority.AboveNormal; workerThread2Done = true; this.Invoke(new EventHandler(WorkerThreadsDone)); } private void WorkerThreadsDone(object sender, System.EventArgs e) { if (workerThread1Done && workerThread2Done) { textBox2.Text = "Processing result: " + processing.Counter.ToString(); button2.Enabled = true; } }</pre> e. Compile, deploy, and then run the application.

Tasks	Detailed Steps
	f. Click the Monitor Sample button, and then watch the result that appears after both worker threads have finished their jobs. <i>Note: The worker threads are completely free to update data whenever they want. They both call Processing.Function1 and Processing.Function2 in the same order. After both threads are finished, the code asks the Processing class for the result (simply returning <code>counter1 - counter2</code>). Because both counters have the same number of increments in the Processing class, a correct return value would be zero, but when you don't think about synchronization, unexpected results might be returned (as you just experienced). To solve this problem, you need to use the Monitor class to protect the data in the Processing class from simultaneous access by multiple threads.</i> <i>Note: Besides Monitor, another class is available for synchronization with comparable functionality: the Mutex class. Because of limited time, you are not going to use Mutex in this lab, but it is important to be aware of both classes. Both Monitor and Mutex protect regions of code that only one thread at a time can execute. Using these synchronization objects adds some responsibility to you as a developer. Prior to accessing the region of code that needs to be protected, you will call either Mutex.WaitOne or Monitor.Enter, depending on which synchronization object you are using. These methods give you immediate access to the protected region of code when no other thread is executing that particular code. However, when another thread already executes that code, Mutex.WaitOne and Monitor.Enter block the requesting thread until the thread currently executing the protected region of code calls Mutex.Release or Monitor.Exit, respectively. Omitting the release of previously obtained Monitor or Mutex objects might lead to unexpected results, including deadlocks. Therefore, it is extremely important to properly use these objects. In cases where protected regions of code might catch exceptions, you should be sure to add exception handling and release the synchronization object in a finally block.</i> g. In the upper-right corner of the form, click ok to close the application. <i>Note: In Visual Studio 2005, you now need to edit the source file Processing.cs.</i> h. Find all occurrences of Monitor.Enter and Monitor.Exit in the source file and uncomment them (there should be four occurrences). i. Compile, deploy, and then run the application. j. Click the Monitor Sample button, and then look at the result that appears after both worker threads have finished their jobs. k. This time, the result that is displayed is always zero. Adding the calls to the Monitor class ensures access to the Processing.Function1 and Processing.Function2 methods by only one thread at a time. l. In the upper-right corner of the form, click ok to close the application. <i>Note: In a managed multithreaded environment, an event is an object that can be used to signal a thread that something has happened. In the context of this lab, events are synchronization objects—not to be confused with UI events that are associated with event handlers. In the .NET Compact Framework, two different types of events are available: AutoResetEvent and ManualResetEvent. The main difference is that an AutoResetEvent event is set and immediately reset again when a thread is waiting on the event. A ManualResetEvent event remains set until it is explicitly reset again. To show the difference between the two different types of events in action, you are going to add a little more code to the application that you have worked on in this exercise.</i> <i>Note: In the following task, you are going to create one worker thread that becomes active when an event is set. You will set the event from the user interface by clicking the Set Event button. By using a CheckBox control, you can choose between AutoResetEvent and ManualResetEvent, and study the different behaviors. Each time you set the event, the number of times the worker thread was executed appears in the text box.</i>

Tasks	Detailed Steps
	Note: The code you are going to add has some logic to differentiate between AutoResetEvent and ManualResetEvent. It also has some functionality to enable or disable relevant buttons.
4. Use Events	a. Switch to the code view of Form1.cs file by clicking on its tab. b. Add the following instance variables to the Form1 class. <pre>private int eventWorkerThreadCounter = 0; private bool eventWorkerThreadDone = true; private Thread eventWorkerThread = null; private AutoResetEvent runOnceEvent = null; private ManualResetEvent runManyEvent = null; private bool useAutoResetEvent = false; private bool eventSampleRunning = false;</pre> c. Add the following code to the eventSampleButton_Click handler. <pre>if (eventSampleRunning) { // The user requests the demo to stop, so be sure to // terminate the worker thread properly. eventSetButton.Enabled = false; eventWorkerThreadDone = true; // Because the worker thread is continuously waiting // for events // before it starts processing, you need to set the // event one // final time if you want to terminate the worker // thread. if (useAutoResetEvent) { runOnceEvent.Set(); } else { runManyEvent.Set(); } eventWorkerThread.Join(); textBox3.Text = ""; checkBox1.Enabled = true; eventSampleButton.Text = "Event Sample"; eventSetButton.Text = "Set Event"; eventSampleRunning = false; } else { // The user requests the demo to start, so create a // worker thread that simply counts the number of // times it loops. eventWorkerThreadDone = false; eventWorkerThreadCounter = 0; checkBox1.Enabled = false; if (useAutoResetEvent)</pre>

Tasks	Detailed Steps
	<pre data-bbox="544 195 1398 604"> { runOnceEvent = new AutoResetEvent(false); } else { runManyEvent = new ManualResetEvent(false); } eventWorkerThread = new Thread(MyEventWorkerThread); eventWorkerThread.Start(); eventSetButton.Enabled = true; eventSampleButton.Text = "Stop Event Sample"; eventSampleRunning = true; } </pre> Note: You have just added some functionality to properly create a worker thread and instantiate a new AutoResetEvent event or a new ManualResetEvent event, depending on the state of the check box. Because a Pocket PC has a limited screen size, you reuse the same button to properly terminate the worker thread after you have finished with the sample. d. Add the following code to the eventSetButton_Click handler. <pre data-bbox="544 852 1398 1455"> textBox3.Text = "Worker thread loop: " + eventWorkerThreadCounter.ToString(); if (useAutoResetEvent) { runOnceEvent.Set(); } else { if (eventSetButton.Text == "Set Event") { eventSetButton.Text = "Reset Event"; runManyEvent.Set(); } else { eventSetButton.Text = "Set Event"; runManyEvent.Reset(); } } } </pre> Note: This code sets an event by clicking the button. In case of a ManualResetEvent event, the button is reused to reset the event. Note: You need one more UI control event handler to monitor the state change of the check box that you added. e. In the form designer, click the check box. f. In the Properties pane, click the Events button. g. Double-click the CheckStateChanged event. h. Add the following code to the CheckStateChanged handler. <pre data-bbox="544 1801 1398 1831"> useAutoResetEvent = checkBox1.Checked; </pre> i. Create the worker thread by adding the following code under the CheckStateChanged method.

Tasks	Detailed Steps
	<pre data-bbox="544 233 1209 863"> private void MyEventWorkerThread() { WaitHandle nextEvent = useAutoResetEvent ? (WaitHandle)runOnceEvent : (WaitHandle)runManyEvent; Thread.CurrentThread.Priority = ThreadPriority.BelowNormal; while (!eventWorkerThreadDone) { nextEvent.WaitOne(); if (! eventWorkerThreadDone) { eventWorkerThreadCounter++; Thread.Sleep(10); } } } </pre> Note: The first statement of the worker thread assigns a WaitHandle object to either a runOnceEvent event (an instantiation of AutoResetEvent) or a runManyEvent event (an instantiation of ManualResetEvent). AutoResetEvent and ManualResetEvent are both derived from WaitHandle. Making use of this fact, you can simply wait in your worker thread for a WaitHandle object to be set. If there had been a chance to pass parameters to a worker thread, you would have passed it a WaitHandle object, making the worker thread completely unaware of the type of event it is waiting on. To mimic that, you now use a local WaitHandle object and assign it to the correct instance variable. Note: The thread itself is extremely simple. Every time it returns from the WaitOne method (which happens when an event is set), a counter is incremented and the thread sleeps for 10 milliseconds. If you select an AutoResetEvent event in the user interface, it becomes clear that the worker thread runs one time for each time the Set Event button is clicked. If you change to a ManualResetEvent event by clearing the check box and then start the worker thread again, the behavior will be different. After you click the Set Event button, the worker thread starts running and continues to do so until the Reset Event button is clicked. j. Compile, deploy, and then run the application. k. Click the Event Sample button. l. Click the Set Event button several times, and then note the behavior of the worker thread. m. Click the Stop Event Sample button, and then change the state of the check box. n. Start the sample again, and note the difference in behavior when you click the Set Event button several times. o. Stop the sample again, and then click ok to close the application.

Summary

In this lab, you performed the following exercises:

- Creating a single-threaded application by using the .NET Compact Framework 2.0
- Modifying the multithreaded application
- Examining Thread and ThreadPool
- Updating user interface controls inside threads
- Using synchronization objects to synchronize threads

In this lab, you created a number of applications to explore the multithreading capabilities of the .NET Compact Framework 2.0. You learned how to properly terminate multithreaded applications. You also found out how to update user interface controls from within worker threads. Using synchronization objects, you learned to synchronize threads to safely update shared data.