# Waitable Timers in .NET with C#

(2009-04-01) - Contributed by Jim Mischel

A waitable time is a synchronization object whose state is set to signaled when a specified due time arrives. In this article, Jim Mischel shows you how to get the most of waitable times using C#.

The .NET Framework includes classes that implement almost all of the Windows synchronization objects. Events, mutexes, semaphores, timers, and other Windows synchronization objects are accessible through managed wrappers found in the System.Threading namespace. One object that is noticeably absent is the waitable timer. That's unfortunate, because waitable timers are very useful objects.

A waitable timer is a synchronization object whose state is set to signaled when a specified due time arrives. Think of a .NET Timer object that sets an event. In fact, you can simulate some of what waitable timers do by using a Timer in conjunction with an AutoResetEvent or ManualResetEvent. For example, suppose you wanted to show a splash screen when your program starts up. You want the splash screen to be displayed while you're initializing data structures, but you want to make sure that it's displayed for at least five seconds, but no more than necessary.

One way to do that would be to create a ManualResetEvent and a one-shot timer set for a five second due time. When the timer expires, it sets the event. The code below illustrates that technique, although it uses console output to simulate displaying a splash screen.

```
static void Main(string[] args)
{
   ManualResetEvent splashEvent = new ManualResetEvent(false);
   using (Timer splashTimer = new Timer((s) =>
      {
         splashEvent.Set();
      }, null, 5000, Timeout.Infinite))
   {
      Console.WriteLine("Splash screen");
      // do initialization here.
      // If initialization takes less than 5 seconds,
      // the Wait below will hold splash visible for
      // a minimum of five seconds.
      splashEvent.WaitOne();
   }
   Console.Write("Press Enter:");
   Console.ReadLine();
}
```

If the initialization code takes longer than five seconds, then the timer will have set the event, and the call to splashEvent.WaitOne will not wait. If initialization takes less than five seconds, then the wait will delay until the timer due time arrives.

A waitable timer object combines the timer and the event into a single object so that, if such a thing existed, you could conceivably write:

```
using (WaitableTimer splashTimer = new WaitableTimer(5000))
{
   Console.WriteLine("Splash screen");
   // do initialization here
   splashTimer.WaitOne();
}
```

If that's all you could do with a Windows waitable timer object, then I'd probably just implement my own object that wraps a Timer and a ManualResetEvent, and provides a simple interface. But waitable timers have some other functionality can't easily be duplicated using the existing .NET synchronization objects.

About Waitable Timers

As I mentioned above, a waitable timer is a synchronization object whose state is set to signaled when the due time arrives. As with events, mutexes, and semaphores, creating a waitable timer returns a handle that you can pass to WaitForSingleObject and other synchronization functions.

There are two types of timers that you can create: manual-reset and synchronization (auto-reset). The only difference between the two timers is that, once signaled, a manual-reset timer's state remains signaled until it is explicitly reset by user code. An auto-reset timer's state is reset to unsignaled after one thread performs a wait operation on it.

If you think of a waitable timer as a gate, then signaling a manual-reset timer would be like opening the gate so that people can pass freely through, until the gate is closed again. An auto-reset timer would be akin to having a gate guard who lets one person through and then closes the gate until told to open it again.

Both types of timers can be configured as one-shot or as periodic timers. A one-shot timer is signaled only once. A periodic timer is signaled at a regular user-defined interval, until canceled.

Because waitable timers are system-defined synchronization objects, they can be shared with other processes. All you need to do is create a named waitable timer object. If the named object already exists on the system, then your program will get a handle to that object and it will be shared among the processes. If the object doesn't exist, it's created in such a way that it can be shared among multiple processes.

All of those functions can be simulated by existing .NET synchronization objects. A Timer combined with an EventWaitHandle would work very nicely. Even the cross-process functionality can be mostly simulated by using a named event, although the timer part will exist only in the creating process. But waitable timers can do a little bit more.

As with the .NET Timer object, a waitable timer can be configured to execute a callback function when the due time arrives. Unlike the Timer callback, which is executed on a thread pool thread, the waitable timer's callback is executed on the thread that created the timer, using the asynchronous procedure call (APC) mechanism. This feature makes for some interesting possibilities but also requires that the thread be in what's called an "alertable wait state." In general, a thread enters an alertable wait state when it calls Sleep, WaitForSingleObject, or one of the other wait functions.

The last thing that a waitable timer can do is a real good one: it can wake your computer from a suspended state in order to signal the event and execute the callback function. Provided, of course, that you have the proper permissions and that your computer actually supports the restore functionality (i.e. has advanced power management).

Working with the API

Whenever I make Windows API subsystems available to managed code, I do it in two steps. First I create a class that contains managed prototypes for the unmanaged functions and make sure that I can use the API directly. Then I build .NET wrapper classes that make a more object-oriented interface that is easier to use from my C# programs.

The waitable timers API consists of just a handful of functions to create and modify timers. The functions are:CreateWaitableTimerCreates a timer with default access.CreateWaitableTimerExCreates a timer, allowing you to request specific access.OpenWaitableTimerOpens an existing waitable timer.SetWaitableTimerSets the due time, period, and callback, and activates the timer.CancelWaitableTimerDeactivates the timer.

The API also includes a handful of constants, and it references a few other API functions (CloseHandle, for example) that I included in the interface class. The entire class, Win32WaitableTimer, is available in the code download for this article.

You'll notice that I used SafeWaitHandle rather than IntPtr in the API functions that expect or return handles. For example, the Windows API definition for CreateWaitableTimer is:

```
HANDLE WINAPI CreateWaitableTimer(
  __in      LPSECURITY_ATTRIBUTES lpTimerAttributes,
  __in      BOOL bManualReset,
  __in      LPCTSTR lpTimerName
);
```

My corresponding managed prototype is:

```
public static extern SafeWaitHandle CreateWaitableTimer(
    SECURITY_ATTRIBUTES timerAttributes,
    bool manualReset,
    string timerName);
```

This is in keeping with the .NET design guidelines. Safe handles help avoid a number of problems that can occur when working with naked handles, and SafeWaitHandle simplifies working with handles that are used with wait functions. For more information about why you should use safe handles, see the .NET Security Blog entry.

Creating and starting a timer is a two-step process. First you create the timer, specifying its type (manual- or auto-reset) and access permissions, and optionally assigning it a name. After the timer is created, you call SetWaitableTimer to set the due time, the period, the callback function and state, and specify whether you want the timer to wake the system from a suspended state.

Creating a simple timer is straightforward. For example, the code below creates a local (non-shared) manual-reset timer using default access and security permissions, and then disposes it when finished.

```
SafeWaitHandle myTimer =
    Win32WaitableTimer.CreateWaitableTimer(null, true, null);
if (myTimer.IsInvalid)
{
    Console.WriteLine("Windows API error creating timer ({0})",
        Marshal.GetLastWin32Error());
}
try
{
    // do processing here.
}
```

```
finally
{
   myTimer.Dispose();
}
```

Like all synchronization objects, a waitable timer is a system resource. You should release them when you're done using them. SafeWaitHandle makes that very easy because it implements IDisposable. As you can see from the code above, I wrap the code in a try/finally block, and call Dispose on the returned timer handle when I'm done processing.

Once you've created a waitable timer, you have to give it a due time before it can do any work. That's done by calling SetWaitableTimer: a function that has many different options. The function's prototype is:

```
[DllImport("kernel32", SetLastError = true)]
public static extern bool SetWaitableTimer(
   SafeHandle hTimer,
   ref long dueTime,
   int period,
   TimerAPCProc completionRoutine,
   IntPtr completionArg,
   bool fResume);
```

The first parameter is easy enough. It's the timer handle returned by CreateWaitableTimer. The next parameter, dueTime is a 64 bit integer that represents the time when the event should fire. There are two ways to specify the time: as an absolute time (1:59 AM on March 14), or as a relative time (one hour from now). SetWaitableTimer tells the difference by the sign of the argument. If dueTime is positive, it's interpreted as an absolute Universal time (i.e. UTC). If dueTime is negative, it's interpreted as a relative time.

In either case, dueTime is expressed as a number of 100-nanosecond ticks. Fortunately, DateTime and TimeSpan objects both have a Ticks property that returns the number of 100-nanosecond ticks that represent the value.

So, if you want the timer's due time to be 1:59 AM on March 14, here's how you'd create the dueTime value:

```
DateTime dtDue = new DateTime(2009, 3, 14, 1, 59, 00);
long dueTime = dtDue.ToUniversalTime().Ticks;
```

f you want the timer to expire an hour from now, you create a TimeSpan that represents that relative time, convert it to ticks, and negate it:

```
TimeSpan tsDue = TimeSpan.FromHours(1);
long dueTime = -(tsDue.Ticks);
```

If you forget to negate the value that you're using for a relative time, SetWaitableTimer will interpret it as a date in the past and your timer will never expire.

By the way, the dueTime parameter must be passed by reference due to historical reasons. It appears that the convention in the Windows API when these objects were created was to pass 64 bit values as pointers (references) rather than by value. The documentation says that it's treated as a constant, though, so the API shouldn't change the value.

The period parameter is used to create a periodic timer. If period is 0, then the timer signals once and is deactivated. If it's greater than zero, then the timer will signal every period milliseconds. Passing a negative value for period will cause SetWaitableTimer to fail.

The next two parameters, completionRoutine and completionArg are used to specify the optional timer callback function that will be called when the timer signals, and the argument to pass to the callback function. The completionRoutine is a TimerAPCProc delegate:

```
public delegate void TimerAPCProc(
   IntPtr completionArg,
   UInt32 timerLowValue,
   UInt32 timerHighValue);
```

The first argument is the completionArg that you passed to SetWaitableTimer. The second and third arguments make up a 64 bit integer that represents the time (in UTC) when the timer signaled. You can convert that to a .NET DateTime in the current time zone like this:

```
long timerValue = timerHighValue;
timerValue = (timerValue << 32) | timerLowValue;
DateTime signalTime = new DateTime(timerValue).ToLocalTime();
```

Remember that the completion routine is placed in the APC queue of the thread that called SetWaitableTimer, and won't be dispatched until the thread enters an alertable wait state.

The completionArg can be difficult to use from .NET, since it involves getting a pointer to a structure and converting it to an IntPtr, or, if you want to pass a reference type, creating a GCHandle and pinning the object in memory so that it doesn't move around.

If the last parameter to SetWaitableTimer, fResume is True, then a system in a suspended power conservation mode will be restored when the timer is signaled. If the system doesn't support restore, the call to SetWaitableTimer will still succeed, but the Windows error code (obtained by calling Marshal.GetLastWin32Error) will be set to ERROR_NOT_SUPPORTED (50).

The most common use of SetWaitableTimer will probably be to set a one-shot or periodic timer, ignoring the completion routine and the resume state. For example, to set a timer to signal five seconds from the current time, you would write:

```
TimeSpan tsDue = TimeSpan.FromSeconds(5);
long dueTime = -(tsDue.Ticks);
if (!Win32WaitableTimer.SetWaitableTimer(myTimer, ref dueTime, 0, null, IntPtr.Zero, false))
{
   Console.WriteLine("Windows API error setting timer ({0})",
      Marshal.GetLastWin32Error());
}
```

In the Windows API, you wait for an object to be signaled by calling WaitForSingleObject, or one of the other wait functions. If we put all of those code samples together, we can duplicate the functionality of the first splash screen example, this time using a Windows waitable timer object:

```
try
{
   // Create the timer
```

```
    SafeWaitHandle myTimer =
        Win32WaitableTimer.CreateWaitableTimer(null, true, null);
    if (myTimer.IsInvalid)
    {
        throw new IOException(
            string.Format("Windows API error creating timer ({0})",
            Marshal.GetLastWin32Error()));
    }

    try
    {
        TimeSpan tsDue = TimeSpan.FromSeconds(5);
        long dueTime = -(tsDue.Ticks);
        if (!Win32WaitableTimer.SetWaitableTimer(myTimer, ref dueTime,
            0, null, IntPtr.Zero, false))
        {
            throw new IOException(
                string.Format("Windows API error setting timer ({0})",
                Marshal.GetLastWin32Error()));
        }

        Console.WriteLine("Splash screen");
        // do processing here.

        Win32WaitableTimer.WaitForSingleObject(myTimer,
            Win32WaitableTimer.INFINITE);

    }
    finally
    {
        myTimer.Dispose();
    }
}
catch (IOException ex)
{
    Console.WriteLine(ex.Message);
}
```

Granted, it's a bit ugly, but it works. We'll pretty it up once we wrap the waitable timer into a nice .NET class.

You can share timers with other processes by using named timers. If you call CreateTimer or CreateTimerEx and pass a non-null value for the timerName parameter, a system timer is created and any process that has sufficient permissions (meaning most local processes) can access that timer by referring to it by name. There are two ways that a process can open an existing timer.

The OpenWaitableTimer API function allows you to open a waitable timer that you know already exists. If the timer exists and your process has sufficient access rights, OpenWaitableTimer will return a handle that refers to it. If the timer doesn't exist or if you don't have sufficient access rights, the returned handle will be invalid.

The other way to open an existing timer is to call CreateTimer or CreateTimerEx, passing it the name of the timer. If a timer with that name already exists, then a valid handle is returned and the Windows error code will be set to ERROR_ALREADY_EXISTS.

We'll look more at sharing timers once we've developed the .NET WaitableTimer class.

Finally, the CancelWaitableTimer method sets a waitable timer to inactive. It stops the timer and, if there are any pending callbacks it removes them from the APC queue. Canceling the timer does not change the signaled state of the

timer. So if there are threads waiting on the timer and you cancel it, those threads will continue to wait until the timer is reactivated and signaled. Similarly, if the timer is already in the signaled state, it remains signaled.

The important thing to remember about canceling a timer is that doing so could cause a deadlock if other threads are depending on it.

By now you should have a pretty good idea of what waitable timers do and how to work with them through the API. I find working directly with the API to be incredibly frustrating and much prefer the .NET class interface to Windows synchronization objects. With that in mind, it's time to start building the managed WaitableTimer class.
{mospagebreak title=Design Decisions}

Design Decisions

When I started this project, I had hoped to create a WaitableTimer class with an interface that's as simple and easy to use as the .NET Timer class. Unfortunately, the many different ways you can use a waitable timer makes that goal impossible to reach. However, with a little thought I was able to at least simplify the most common cases and make the full functionality available when necessary.

I've attempted to follow the interface conventions for the other .NET synchronization objects as closely as possible, including object security. Following those conventions should make it easy for anybody familiar with the .NET synchronization objects to use this new object.

We'll start with the constructors. At minimum, I need one that allows all parameters to be specified and, following the convention of the other synchronization objects, includes a security attributes parameter as well as an <tt>out</tt> parameter that tells whether the object was created new. The "specify everything" constructor for our object looks like this:

```
public WaitableTimer(
   bool manualReset,
   string name,
   out bool createdNew,
   WaitableTimerSecurity timerSecurity)
```

The last two parameters have meaning only when creating a named timer, so it makes sense to have an overloaded constructors that eliminate them. This, too, is in keeping with the interfaces for EventWaitHandle, Mutex, and Semaphore.

```
public WaitableTimer(bool manualReset)
public WaitableTimer(bool manualReset, string name)
```

One thing that bothers me about working with waitable timers in the API is that I have to create the object and then call SetWaitableTimer to start it. That separation can be useful at times, but most of the time when I create a timer I want it to start immediately. So it seems reasonable to add constructors that start the timer. But there are so many options to SetWaitableTimer that supplying reasonable constructors for all cases would be too confusing. So I decided to support the ways that I typically use the objects. I don't often use the callback or resume functionality, so the only parameters I really need are the due time and period.

As you recall, SetWaitableTimer interprets the dueTime parameter in two different ways: as an absolute UTC date and

time, or as a relative number of ticks from the current time. I need overloads to handle each of those cases, which means four additional constructors:

```
public WaitableTimer(bool manualReset, DateTime dueTime, int period)
public WaitableTimer(bool manualReset, TimeSpan dueTime, int period)
public WaitableTimer(bool manualReset, DateTime dueTime,
   int period, string name)
public WaitableTimer(bool manualReset, TimeSpan dueTime,
   int period, string name)
```

All told, that's seven different constructors, which is on the edge of excessive to me. But it covers the most common cases for me and still allows the create-then-set functionality if I need to use the timer's more advanced features.

I think this is a shining example of the need for optional parameters in C#--a feature that's coming in version 4.0. If optional parameters existed (as they currently do in Visual Basic .NET), those six additional constructors could be cut down to three.

For OpenWaitableTimer, I again chose to go with the convention used by the existing .NET synchronization objects.

```
public static WaitableTimer OpenExisting(string name)
public static WaitableTimer OpenExisting(string name,
   WaitableTimerRights rights)
```

That leaves a method to set the timer options, and a method to cancel the timer. Canceling the timer is trivial:

```
public void Cancel()
```

The System.Threading.Timer class has overloaded methods called Change that change the timer settings. So that's what I decided to call my methods that modify timer settings. There are two "full specification" methods: one for setting absolute due times, and one for setting relative due times.

```
public int Change(
   DateTime dueTime,
   int period,
   WaitableTimerCallback callback,
   object state,
   bool resume)

public int Change(
   TimeSpan dueTime,
   int period,
   WaitableTimerCallback callback,
   object state,
   bool resume)
```

And, since I rarely use the final three parameters, I decided to give myself a couple of shortcuts:

```
public int Change(DateTime dueTime, int period)
public int Change(TimeSpan dueTime, int period)
```

Note here that Change returns an int. The value returned is the value obtained by Marshal.GetLastWin32Error, and the reason it's there is because there's one instance in which SetWaitableTimer can succeed, but GetLastError returns a non-zero value.

If the last parameter to Change is true, then the computer will resume from a low-power state when the timer fires. If the computer supports resume. If the computer does not support resume, the call to SetWaitableTimer succeeds, but GetLastError will return ERROR_NOT_SUPPORTED. To duplicate that functionality, Change returns the value that it gets from Marshal.GetLastWin32Error after calling the API function to set the timer.

There's also the matter of the WaitableTimerCallback delegate. The TimerAPCCallback that's used in the API is rather inconvenient to work with in .NET programs. Since I have to intercept the callback anyway (for reasons I'll explain below), I've taken the liberty of making an easier to use callback delegate:

```
public delegate void WaitableTimerCallback(object state,
   DateTime tickTime);
```

That's the entire WaitableTimer interface. The public interface includes just four methods: the constructor, OpenExisting, Change, and Reset. Of course, all but Cancel have overloads that complicate things a little bit, but the entire thing is a relatively small amount of code.
{mospagebreak title=Object Security}

## Object Security

The existing synchronization objects support object security through classes derived from System.Security.AccessControl.NativeObjectSecurity. These security classes are designed to control access to native objects without direct manipulation of Access Control Lists (ACLs). Since I was unable to find information about deriving from NativeObjectSecurity, I studied documentation for the existing classes, poked around in their code with ILDASM, and then duplicated their interfaces.

The resulting classes, WaitableTimerAccessRule, WaitableTimerAuditRule, and WaitableTimerSecurity are all in the WaitableTimerSecurity.cs source file. For the most part, these classes are simply type-safe wrappers around the base class interfaces.

I won't go into detail about how to use the classes to control object access. For more information, see the MSDN documentation for MutexSecurity.

## Using the WaitableTimer Class

WaitableTimer can do a lot of different things. Let's start by implementing the fictional splash screen program that I used to start this article. With the new WaitableTimer class, that code becomes:

```
using (WaitableTimer splashTimer = new WaitableTimer(true, TimeSpan.FromSeconds(5), 0))
{
   Console.WriteLine("Splash screen");
   // do initialization here
   splashTimer.WaitOne();
}
```

That code, which is almost identical to the "if it were possible" code I showed at the beginning of the article, creates a manual-reset timer one-shot timer will signal after five seconds.

Now, suppose you have multiple threads that access a single global resource, and there are limits to how fast you can access that resource. Say, once per second. How do you prevent the multiple threads from accessing it too fast?

The answer is to create a periodic, auto-reset WaitableTimer that signals once per second. Each time the timer signals, one waiting thread will be released. Here's some simple code that creates the timer, starts some threads, and then waits until each thread has acquired the resource ten times. The threads automatically terminate after the tenth iteration:

```
WaitableTimer resourceTimer = new WaitableTimer(false, TimeSpan.FromSeconds(1), 1000);
try
{
   // start threads
   Thread[] threads = new Thread[numThreads];
   for (int i = 0; i < numThreads; ++i)
   {
      Thread t = new Thread(() =>
         {
            Console.WriteLine("Thread {0} started",
               Thread.CurrentThread.ManagedThreadId);
            for (int x = 0; x < 10; ++x)
            {
               resourceTimer.WaitOne();
               Console.WriteLine("Thread {0}: {1}",
                 Thread.CurrentThread.ManagedThreadId,
                 x);
               // Simulate doing some work
               Thread.Sleep(1200);
            }
            Console.WriteLine("Thread {0} exit",
             Thread.CurrentThread.ManagedThreadId);
         });
      threads[i] = t;
      t.Start();
   }
   Console.WriteLine(
      "Threads started.  Waiting for all to exit.");
   for (int i = 0; i < numThreads; ++i)
   {
      threads[i].Join();
   }
}
finally
{
   resourceTimer.Close();
}
```

The call to <tt>Thread.Sleep</tt> in the thread's processing loop is just taking up time to simulate accessing the resource and processing whatever data it obtained. The idea here is that if the resource can only be accessed once per second, it makes little sense to have multiple threads accessing it unless it takes more than a second to process an item.

If you change the sleep to just 10 milliseconds, you'll likely see that one or two threads will monopolize the resource until they've done their ten iterations, and then another thread will monopolize it. That's just the nature of synchronization options and the Windows scheduling algorithm: there's no guarantee that threads will be released from a wait in anything like a first come, first served basis.

The nice thing about waitable timers is that they can be used across processes. Suppose we had the same resource as described above, but rather than multiple threads in a single process wanting to access it, we have multiple processes that want to access it. We still want to rate limit it to one access per second.

In this case, we need to create a named, periodic, auto-reset timer. The code to support multiple processes is actually easier, because it doesn't have to create or shut down any threads:

```
WaitableTimer resourceTimer = new WaitableTimer(false, TimeSpan.FromSeconds(1), 1000, "$ResourceTimer$");
try
{
   for (int x = 0; x < 10; ++x)
   {
      resourceTimer.WaitOne();
      Console.WriteLine("{0}", x);
      // Simulate doing some work
      Thread.Sleep(1200);
   }
}
finally
{
   resourceTimer.Close();
}
```

The big difference here is that the constructor passes a timer name, "$ResourceTimer" to the constructor. The Windows API CreateWaitableTimer function checks to see if a timer with that name exists. If the name does exist, a reference to the existing object is returned. Otherwise, a new Windows object is created and the WaitableTimer constructor sets its parameters.

If you run multiple instances of a program that includes the code above, you'll see that they all share the timer.

For the final example, I want show how you can "wake up" your laptop at a specified time. Remember that the last parameter to the SetWaitableTimer API function is a flag that indicates whether the computer should resume from a low-power state when the timers due time arrives.

```
static void Main(string[] args)
{
   using (WaitableTimer wt = new WaitableTimer(true))
   {
      int rslt = wt.Change(TimeSpan.FromMinutes(1), 0, WakeupProc, null, true);
      if (rslt == WaitableTimer.ErrorNotSupported)
      {
         Console.WriteLine("Computer does not support resume.");
      }
      // wait for event to occur
      Console.WriteLine("Waiting for timer to expire");
      wt.WaitOne();

      // Sleep for 5 seconds, which should be
      // enough time for the WakeupProc to do its thing.
      Thread.Sleep(5000);

      Console.Write("Press Enter:");
      Console.ReadLine();
   }
}

static void WakeupProc(object state, DateTime eventTime)
```

```
{
   Console.WriteLine("Current time is {0}", DateTime.Now);
   Console.WriteLine("Event fired at {0}", eventTime);
}
```

If the computer doesn't support resuming, SetWaitableTimer will still succeed, but a call to GetLastError will return ERROR_NOT_SUPPORTED. The return value of the Change method is the last error code. The example above shows how to use that value to determine if the computer supports resuming.

One other thing about the wakeup example is that the main thread calls Thread.Sleep after the timer has fired. Recall that, unlike other timers, the waitable timer calls the completion method on the same thread that called SetWaitableTimer. In order for the APC mechanism to make that call, the thread must be in an alertable wait state. The call to Thread.Sleep puts the thread in such a state, so the WakeupProc can be called.

Another way to achieve that result is to create a second event: a ManualResetEvent, for example, and have the main thread wait on that. The completion method would then signal that event, which would allow the main thread to continue.
{mospagebreak title=About the Code }

About the Code

Most of the code in WaitableTimer.cs is simple API wrapper code: calling API functions with method parameters, returning the results, and handling errors. Only two parts of the code required extra special attention.

All of the constructors end up calling the CreateTimerHandle method, which in turn calls the CreateWaitableTimer API function. For the most part, CreateTimerHandle just shuffles parameters and makes the API call. Except that when the timerSecurity parameter is non-null, the method has to construct a SECURITY_ATTRIBUTES structure and populate it. That ends up requiring unsafe code because I have to allocate some memory and supply a pointer to the lpSecurityDescriptor member.

The other place that required some fiddling around is the callback. If you recall, the waitable timer can call a function when the timer period elapses. The TimerAPCProc delegate is defined as:

```
public delegate void TimerAPCProc(
   IntPtr completionArg,
   UInt32 timerLowValue,
   UInt32 timerHighValue);
```

The first parameter is some user-defined state information that's passed to SetWaitableTimer. The idea is that the API code will hold that reference and pass it to the callback function. Although it's possible to pass a managed reference to unmanaged code and then reconstitute it in a callback, doing so involves creating and pinning a GCHandle, converting that to an IntPtr, and then passing the result to the API. The callback then has to go the other way: from IntPtr to GCHandle, and then to whatever the object's real type is.

The conversion back and forth isn't so difficult, but it's also necessary to keep track of that GCHandle so that it is released once the timer is destroyed. It's all possible, but it's a messy bit of code.

The other problem with TimerAPCProc is that the timer value is expressed as two unsigned integers. Combined, these make up a 64 bit value that represents the date and time (in UTC) that the timer fired. It's terribly inconvenient to use in C#.

To solve both problems, I created an internal class called CallbackContext, which holds the state and the actual callback. The Change method creates a CallbackContext object and passes a reference to that object's TimerTick method (which is a TimerAPCProc delegate) to SetWaitableTimer. When the timer fires the completion callback converts the timer value into a DateTime structure. It then calls the WaitableTimerCallback that the user specified, passing it the user's state information and the constructed DateTime structure.

Doing it this way makes the timer callback much easier to use in C#. It does require slightly more time, but timer events occur rarely enough (no more than once per millisecond, which is pretty darned slow to a 2 GHz processor) that it shouldn't have any noticeable impact on performance.
{mospagebreak title=Parting Thoughts}

## Parting Thoughts

WaitableTimer is a very useful synchronization object to have. Although not used nearly as often as EventHandle and Mutexe--or even Semaphore--when you need a waitable timer, you really need one. As I said, there are ways to simulate most of the waitable timer functionality using a regular timer object and an EventHandle, although doing so carries some very tight restrictions. However, WaitableTimer has one feature--the ability to "wake up" your computer--that doesn't seem to be possible any other way. Combined with its other features, that makes WaitableTimer an essential synchronization object. I wonder why it was never included as part of the .NET Framework.

## Source Code Availability

A Visual Studio 2008 project, including full source for the WaitableTimer class and supporting classes, and some test programs is available for download from my Web site at http://www.mischel.com/pubs/waitabletimer.zip.