Newsletter	
Issue Date	
Section	
Main file name	
Accompanying ZIP file name	
Listing file name	
Sidebar file name	
Table file name	
Screen capture file names	
Infographic/illustration file names	
Photos or book scans	
Special instructions for Art dept.	
Editor	
Status	
Spellchecked (set Language to English U.S.)	
EN review	
Character count	
Newsletter blurb	

Overline:

Byline:
by Juval Lowy

Technology Toolbox:
[[Art: Check box for those highlighted below.]]
VB.NET
C#
SQL Server 2000
ASP.NET
XML
VB6

Other:
Xxxxxxx

Note:
Xxxx xxxx xxxx xxx.

Resources:
- xxxxxxx
- xxxxxxx

Head:

TICKING AWAY WITH .NET TIMERS

A timer is an object that calls back into the application at a set interval repeatedly. Timers are often used to update the user interface, with anything from stock quotes to available disk space. You can use a timer to implement a "watch dog", which periodically checks the status of various components or devices in your application. Other usages of timers involve polling communication ports, or checking the status of job queues. In short, every decent size application makes use of timers. In the past, developers were left to their own devices when it came to implementing timers. Developers usually created a worker thread that executed the following logic in pseudo code:

```
while(true)
{
    Tick();
    Sleep(interval);
}
```

However, such solutions had disadvantages: developers had to write the code to start and stop the timer, manage the worker thread, change the interval, and hook the timer to the callback function. Furthermore, even if developers took advantage of timers available by the operating system, they coupled their application to that mechanism, and switching to a different implementation was not trivial.

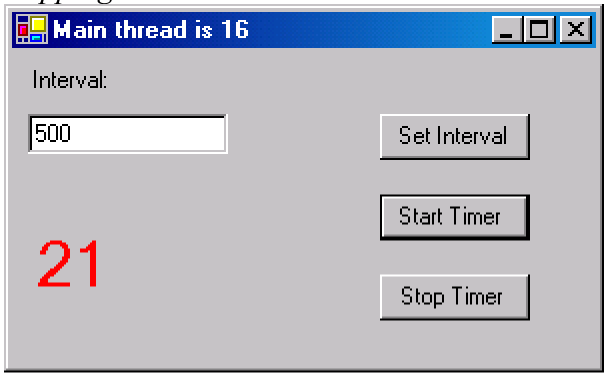
.NET comes out of the box with not one, but three complimentary timer mechanisms you can use in your application. This article discusses and contrasts .NET timers support, and recommends where and when to use each.

Before I dive into the specific of the timers, I would like to list down the generic requirements a timer mechanism should provide:

- The application should be able to start and stop the timer repeatedly.
- The application should be able to change the timer interval, even while the timer is active.
- The timer interface should encapsulate as mush as possible the underlying operating system mechanism used to implement it, to reduce coupling between the application and the actual timer mechanism.
- The application should be able to use the same callback method to service multiple timers, and it should be able to distinguish between the different timers.

With these requirements in mind, let's examine the three available timer mechanisms, by using them to implement timer support for the same application - a simple Windows Forms application (see Figure 1). The application displays a counter, and provides an OnTick method, called by the timer, to increment the counter. The application allows setting the timer interval, and starting and stopping the timer. The OnTick() method also traces to the output Window the thread ID of the thread used for the callback.

Figure 1: The Timer Demo Applications. To compare between the different timers, this article implements the same test client using the different timers. The test client uses a timer to increment a counter, and it allows setting the timer interval, and starting or stopping the timer.



System.Timers.Timer

The System.Timers namespace contains the definition of the Timer class.
Add a Timer member to your class, and make sure to qualify its namespace:

```
protected System.Timers.Timer m_Timer;
```

You need to qualify the namespace because this is a Windows Forms application, and the Windows.Forms namespace has a different Timer class, discussed next.

In the form constructor, create a new Timer object after the call to InitializeComponent() (see Listing 1). Now you need to associate the timer with an application-provided callback method. The System.Timers.Timer class has an event member called Elapsed, which expects a delegate target of type ElapsedEventHandler, defined as:

```
public delegate void ElapsedEventHandler(object sender,ElapsedEventArgs e);
```

Add a method to the form class called OnTick with the ElapsedEventHandler signature:

```
void OnTick(object sender,ElapsedEventArgs e){…}
```

and set up the OnTick() method as the Elapsed target:

```
m_Timer.Elapsed += new ElapsedEventHandler(OnTick);
```

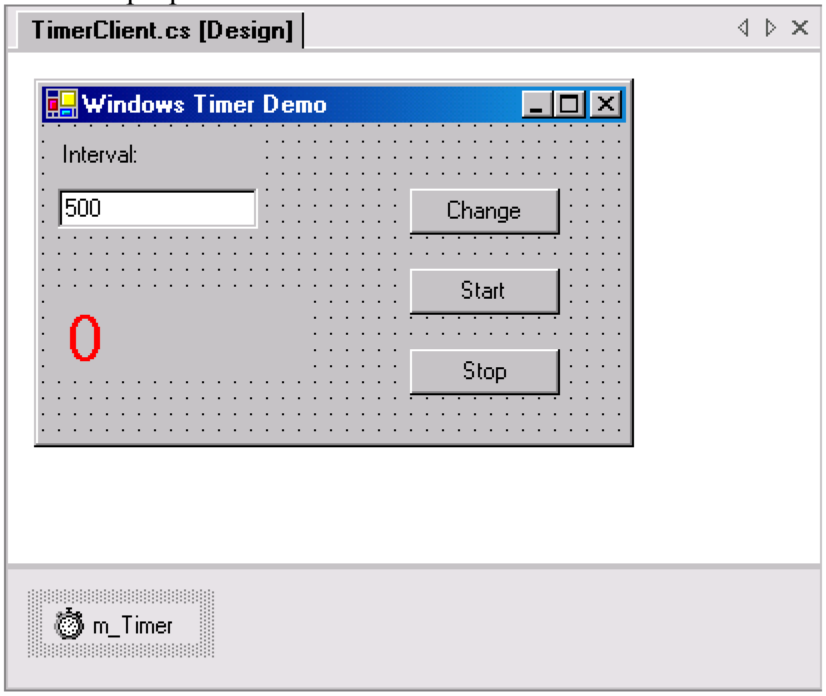
The ElapsedEventArgs class provides the time the method was called as well. Note that you can use the sender argument to distinguish between different timers, if you choose to use the same callback method to handle ticks from multiple timers.

To set the timer period, set the Interval property of the Timer class, and to start or stop the timer notifications, simply set the Enabled property to true or false respectively. Finally, when the application shuts down, call the Close() method of the timer, to dispose of the system resources it used.
As you can see, taking advantage of the System.Timers.Timer class is straightforward and it complies with the general requirements I set.
When you run the application, you will see that different threads are used to call the OnTick method. .NET uses threads from the Windows thread pool to invoke the callback. However, the pool is not endless, and the processing of the OnTick() method should be of short duration, to release the thread back to the pool as soon as possible. If the processing time is longer than the timer interval, then a new thread will be dispatched to call OnTick(). Eventually, you will exhaust the pool, and no new ticks will happen until one of the thread returns to the pool. Pool size may be the product of the hardware and operating system configuration, and on my machine, the pool has 25 threads. Another point worth paying attention to is object synchronization. Because the OnTick() callback is called on a different thread, make sure you synchronize access to the object members it accesses, to prevent state corruption.

System.Windows.Forms.Timer

The System.Windows.Forms namespace contains another definition of a Timer class. Even though this timer is actually based on the WM_TIMER Windows message, the timer exposes the standard way of callback using delegates, and completely encapsulates the underlying message pump loop. Visual Studio.NET has built-in Designer support for the Windows.Forms timer. Simply drag and drop from the toolbox a timer control to the client form. The Designer then displays the timer icon underneath the form (see Figure 2).

Figure 2: The Designer view of Windows.Forms.Timer. This timer has full Designer support. It displays the timers the form uses underneath it, and you can view and modify the timer properties.



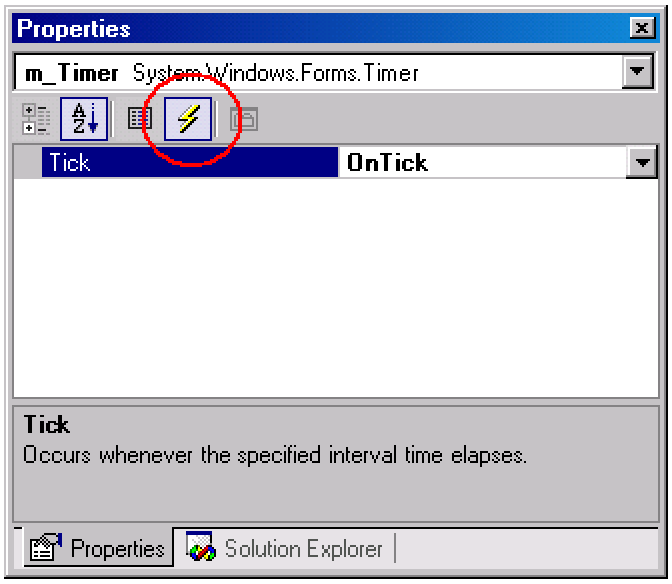
The Designer adds this member to the form class:

```
private System.Windows.Forms.Timer m_Timer;
```

You can use the Properties view to change the name of the timer and its initial interval.

To add a tick event handler, switch the Timer Properties view to Events (see Figure 3),

Figure 3: The events Properties view of Windows.Forms.Timer. Use this view to hook up the Tick event to a callback method on the form.



Tick OnTick

Tick
Occurs whenever the specified interval time elapses.

and input OnTick for the tick event handler. The Designer then adds the OnTick() method to the class:

```
void OnTick(object source,EventArgs e){…}
```

Note that you can use the source argument to distinguish between different timers, if you choose to use the same callback method to handle ticks from multiple timers.

The Windows.Forms.Timer class has Enabled and Interval properties, used exactly the same way as the System.Timers.Timer class.
Listing 2 shows the formb definition when using Windows.Forms.Timer. Note how similar to Listing 1 the resulting code it, even though the mechanism used is drastically different. Also worth mentioning is that all the callbacks are dispatched on the main UI thread, and there is no need to manage concurrency.

System.Threading.Timer

The third timer class is defined in the System.Threading namespace. This timer is similar to the System.Timers.Timer class: it too uses the thread pool and the application using it is subject to the same design limitations. The main differences are that is provides fine grained and advanced control - you can set its due time (when should it start ticking) and you can pass any generic information to the callback tick method. Because the timer derives from MarshalByRefObject, it is the only timer mechanism that can be accessed remotely from outside the application domain or across machines. This makes it particularly useful in remoting and leasing scenarios, if you set the lease to automatically renew on call. I will discuss remoting in future articles.

To use the System.Threading.Timer class, add this member to the form:

```
protected System.Threading.Timer m_Timer;
```

This timer requires a delegate of type TimerCallback, defined as:

```
Public delegate void TimerCallback(object state);
```

You need to manually define and hookup the callback method to the timer object.
Add the OnTick() method to your class:

```
void OnTick(object state){}
```

Create a TimerCallback delegate object, warping the OnTick() method:

```
TimerCallback callBack = new TimerCallback(OnTick);
```

and provide it as a constructor parameter to the timer object:

```
m_Timer = new System.Threading.Timer(callBack,null,0,period);
```

The other parameters the timer constructor accepts are due time, period (the timer interval), and a generic object, used to pass additional information to the callback:

```
public   Timer(TimerCallback   callback,object   state,int   dueTime,int
period);
```

The state object is typically the object that created the timer, so that you could use the same callback method to handle ticks from multiple senders, but you can of course pass in any argument you like.

To change the timer period, simply call the Change() method, which accepts the new due and interval.
On the other hand, this timer does not provide easy to use Enable property. It starts ticking immediately after the constructor (actually after the due time has elapsed), and to stop it, you must call its Dispose() method. If you want to re-start it, you need to create a new object.
Listing 3 demonstrates using this timer.

Conclusion

The three timer types provide different degrees of flexibility versus ease of use. The Windows.Forms.Timer class is by far the easiest to use of them all, it poses no concurrency issues, and it has full designer support. However, it only works with Windows Forms applications, and it processes the tick event on the main thread. That may be a problem if the processing is long, because during the processing, the user interface will not be responsive.

I recommend using the System.Timers.Timer class in all other cases. It's methods are easy to use, and it automatically manages the ticking threads for you. In any case, the additional features of the System.Threading timer class are still available for you if you require them.

Listing 1: Using the System.Timers.Timer class. The Timer class provides the Enable property to start and stop the timer, and the Interval property to set the timer period. The timer requires ElapsedEventHandler compliant callback method signature.

```
public class TimerClientFrom : Form
{
	private Button m_StartButton;
	private Button m_StopButton;
	private Label m_IntervalLabel;
	private TextBox m_IntervalBox;
	private Button m_IntervalButton;
	private Label m_Counter;
	protected System.Timers.Timer m_Timer;

	public TimerClientFrom()
	{
		InitializeComponent();

		m_Timer = new System.Timers.Timer();
		m_Timer.Elapsed += new ElapsedEventHandler(OnTick);
	}
	protected void OnTick(object source,ElapsedEventArgs e)
	{
		int currentCounter = Convert.ToInt32(m_Counter.Text);
		currentCounter++;
		m_Counter.Text = currentCounter.ToString();

		Trace.WriteLine("Timer thread is " +
		Thread.CurrentThread.GetHashCode().ToString());
	}

	private void InitializeComponent()
	{
		//Designer generated code, including this line:
		Closed += new EventHandler(OnClose);
	}

	static void Main()
	{
		Application.Run(new TimerClientFrom());
```

```
      }

      private void OnStart(object sender,EventArgs e)
      {
            m_Timer.Enabled = true;
      }

      private void OnStop(object sender,EventArgs e)
      {
            m_Timer.Enabled = false;
      }

      private void OnSetInterval(object sender,EventArgs e)
      {
            m_Timer.Interval = Convert.ToInt32(m_IntervalBox.Text);
      }

      private void OnClose(object sender,EventArgs e)
      {
            m_Timer.Close();
      }
}
```

Listing 2: Using Windows.Forms.Timer. The Timer is a control with full Designer support. It provides the Enable property to start and stop the timer, and the Interval property to set the timer period. The timer requires generic EventHandler compliant callback method signature.

```
public class TimerClientFrom : Form
{
      private Button m_StartButton;
      private Button m_StopButton;
      private Label m_IntervalLabel;
      private TextBox m_IntervalBox;
      private Button m_IntervalButton;
      private System.Windows.Forms.Timer m_Timer;
      private System.ComponentModel.IContainer components;
      private Label m_Counter;

      public TimerClientFrom()
      {
            InitializeComponent();
      }
      protected void OnTick(object source,EventArgs e)
      {
            int currentCounter = Convert.ToInt32(m_Counter.Text);
            currentCounter++;
            m_Counter.Text = currentCounter.ToString();

            Trace.WriteLine("Timer thread is " +
            Thread.CurrentThread.GetHashCode().ToString());
      }
      private void InitializeComponent()
      {
            //Designer generated code, including this line:
            m_Timer.Tick += new EventHandler(OnTick);
      }

      static void Main()
      {
            Application.Run(new TimerClientFrom());
      }

      private void OnStart(object sender,EventArgs e)
      {
            m_Timer.Enabled = true;
      }

      private void OnStop(object sender,EventArgs e)
      {
            m_Timer.Enabled = false;
      }

      private void OnSetInterval(object sender,EventArgs e)
      {
            m_Timer.Interval = Convert.ToInt32(m_IntervalBox.Text);
      }
}
```

Listing 3: Using System.Threading.Timer class. This timer is for advanced use. You cannot stop and start a timer object, but it does offer advanced features such as remoting and due period.

```
public class TimerClientFrom : Form
{
      protected System.Threading.Timer m_Timer;

      private Button m_StartButton;
      private Button m_StopButton;
      private TextBox m_IntervalTextBox;
      private Label m_Counter;
      private Label m_IntervalLabel;
      private Button m_Change;

      public TimerClientFrom()
      {
            InitializeComponent();
      }


      private void InitializeComponent() {…}


      static void Main()
      {
            Application.Run(new TimerClientFrom());
      }

      private void OnChange(object sender,EventArgs e)
      {
            if(m_Timer == null)
                  return;
            int period = Convert.ToInt32(m_IntervalTextBox.Text);
            m_Timer.Change(0,period);
      }

      private void OnStart(object sender,EventArgs e)
      {
            TimerCallback callBack = new TimerCallback(OnTick);
            int period = Convert.ToInt32(m_IntervalTextBox.Text);
            m_Timer = new
            System.Threading.Timer(callBack,null,0,period);
      }

      private void OnStop(object sender,EventArgs e)
      {
            m_Timer.Dispose();
            m_Timer = null;
      }
      protected void OnTick(object state)
```

```
        {
                int counter = Convert.ToInt32(m_Counter.Text);
                counter++;
                m_Counter.Text = counter.ToString();
                Trace.WriteLine("Timer thread is " +
                Thread.CurrentThread.GetHashCode().ToString());
        }
}
```