Timer surprises, and how to avoid them

By Luc Pattyn | 2 Feb 2007

.NET1.1.NET2.0VS.NET2003VS2005C#WindowsDevIntermediate

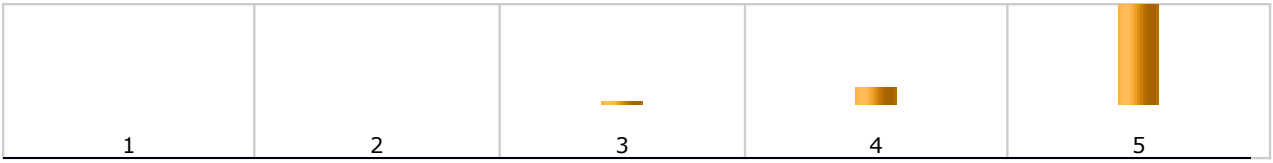
A simple app demonstrates unexpected behavior of .NET timers and Thread.Sleep(); native multimedia timers come to the rescue.

http://www.codeproject.com/Articles/17474/Timer-surprises-and-how-to-avoid-them#

- More by this author

Article	Browse Code	Stats	Revisions

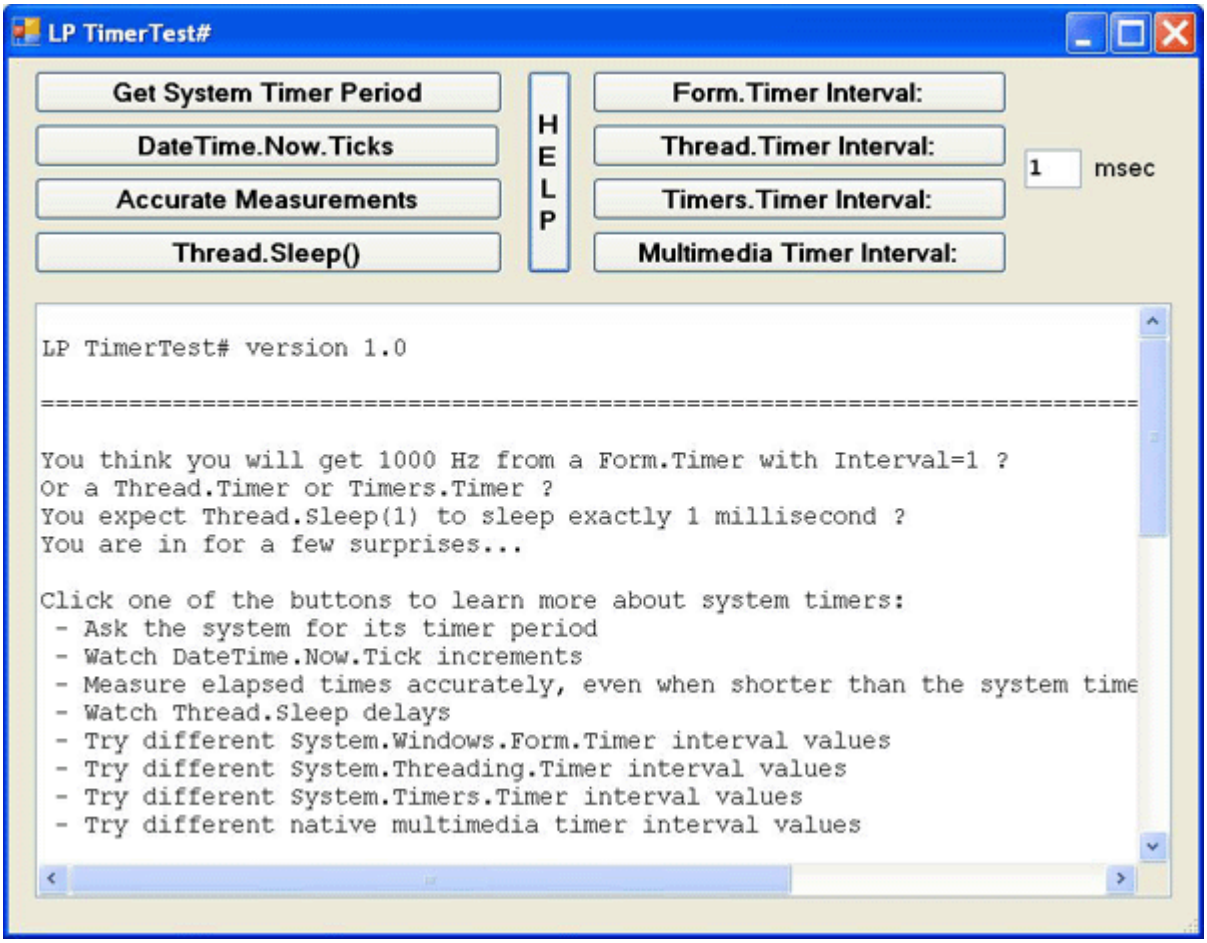
4.81 (67 votes)



/wEPDwUKMjExO

/wEWCAL2j9O8A

- Download executable - 9.65 KB
- Download source - 13.3 KB



Introduction

The .NET Framework offers different timer classes; each of them can deliver periodic events, and the period is user selectable. Since their period is expressed as a number of milliseconds, one could expect such events could reach a frequency up to 1000 Hz. But this is not at all true, the maximum frequency is far less, and it depends on your specific system.

The TimerTest application provides a simple means to test the behavior of the different .NET timers on your specific system. It also includes some operations based on native code (performance counters and multimedia timers), showing more accurate time measurements and higher timer frequencies can be achieved.

The problem

Windows, first and for all, has a "system timer" that is the primary means for keeping track of time, for switching tasks, and for invoking single-shot or periodic timer events upon the user's request. Historically, the system timer has been based on simple hardware, and a system timer frequency of 50 to 60 Hz seemed appropriate.

Server applications probably do not mind at all; typical desktop applications, when implemented correctly, can also work fine with such a clock; some applications, both multimedia and others, however, may need a finer notion of time.

Nonetheless, the .NET timer classes seem to rely on the good old system timer resolution, and hence are unable to provide short intervals such as 1 or 5 milliseconds, or high frequencies such as 500 or 1000 Hz.

The test program

The program is fairly simple. There is only one form, it holds a listbox showing all output, and a collection of buttons, each calling one of the timer tests.

Get System Timer Period

This calls the Win32 function `GetSystemTimeAdjustment()` and returns the period of the system timer. I have tried this on a couple of PCs and got values of 10 and 15 - 16 milliseconds; I also recall this function returned 55 to 60 msec on an old Pentium II machine running Windows 98 (which also could run .NET!). So the message is twofold: the number may vary, and it is well above 1 msec, so this is not the way to achieve the best timing resolution.

DatTime.Now.Ticks

This is a short loop calling `DateTime.Now.Ticks` and comparing it to the previous result. Most of the time, the difference is zero, suggesting an infinitely fast machine. And when the difference is not zero, it equals a large number, such as 100000 or 156250, which corresponds to 10 msec or 15.625 msec since it counts "ticks" (a tick always is 100 nsec). So the surprise here is that the `DateTime.Now` struct, while holding a millisecond field, only gets updated 60 to 100 times per second, so formatting `DateTime.Now.ToString("HH:mm:ss.fff")` to obtain milliseconds will show at most 60 to 100 different combinations for the fractional digits.

Accurate Measurements

We now use a Win32 `PerformanceCounter` to get more precise time measurements. A performance counter is based on different hardware: a separate 64-bit counter is counting a fixed frequency, and its value can be read at all times. The overall approach expressed in pseudo-code:

Collapse | ~~Copy Code~~

```
long freq=GetPerformanceCounterFrequency();
long start=GetPerformanceCounterValue();
...code to be timed
long stop=GetPerformanceCounterValue();
double secondsElapsed=((double)(stop-start))/freq;
```

The test times a sequence of half-second delays (`Thread.Sleep(500)`) and the results prove that a much higher resolution is obtained. Most systems seem to have a `PerformanceCounterFrequency` of either 3,579,545 Hz or 10,000,000 Hz; the former is a popular frequency for real-time clock devices, the latter corresponds to the tick frequency. I have never encountered a frequency below 1 MHz, so I would conclude this method of measuring elapsed time is adequate down to a few tens of microseconds.

The test also contains an "empty test", where the code to be timed is empty, resulting in around 10 microseconds of measurement overhead.

Thread.Sleep()

This test performs a number of `Thread.Sleep()` calls, with a period of 25, 10, 5, or 1 milliseconds; and it uses the above performance counter to measure the elapsed time in each case. The surprise now is the sleep time is not what we asked for, it is longer, and how much longer heavily depends on the system:

- on one system, I observed all delays corresponded to the requested time rounded up to the next multiple of the system timer period (so 25 became 30 msec), and 10, 5, and 1 all resulted in about 10 milliseconds.
- on another system, I observed the requested 25, 10, 5, and 1 msec resulted in an average of 25.5, 10.8, 5.9, and 2.0 msec, which is not too bad.

Unfortunately, I see no major reason for such a different behavior, and I can not predict what it will be without running a test application like the one described here.

Form.Timer Interval

This test will launch a `System.Form.Timer` and wait for it to finish a number of events. The timer period is user selectable through the `TextBox`, the value must be in the range [1, 1000] msec. The number of events measured is such that the total test time is approximately 5 seconds. Again, in pseudo-code:

Collapse | ~~Copy Code~~

```
timer.Interval=int.Parse(TextBox.Text);
int maxCount=5000/timer.Interval;
timer.Start();

private void timerEventHandler(...) {
    count++;
    if (count>=maxCount) {
        ... stop the test
    }
}
```

The result for all systems seems to be that the requested timer period always gets rounded up to the next system timer period multiple, so the shortest achievable interval equals 10 to 16 milliseconds. Please note that the event handler is extremely short, except for its last invocation when the timer gets stopped and the results get reported.

Thread.Timer Interval

Similar code, but using a `System.Thread.Timer`, same results. Remark: the timer's event handler runs on a different thread, so the log method needed the InvokeRequired/Invoke pattern to keep the listbox happy.

Timers.Timer Interval

Similar code, but using a `System.Timers.Timer`, same results, and same remark.

Multimedia Timer Interval

Similar code, but this time based on the Win32 functions **`timeSetEvent()`** and `timeKillEvent()` from ***WinMM.dll***.

The prototypes and delegates involved are:

Collapse | ~~Copy Code~~

```
[DllImport("WinMM.dll", SetLastError=true)]
private static extern uint timeSetEvent(int msDelay, int msResolution,
        TimerEventHandler handler, ref int userCtx, int eventType);

[DllImport("WinMM.dll", SetLastError=true)]
static extern uint timeKillEvent(uint timerEventId);

public delegate void TimerEventHandler(uint id, uint msg,
       ref int userCtx,    int rsv1, int rsv2);
```

With this timer, the results are excellent: asking a 1 millisecond interval really generates a 1 millisecond interval. So this is the timer to use when small intervals are important. Be careful though, the threading and `Invoke` remark applies here too. Of course, if the event handler were to consume more time (e.g., by updating the user interface through `Invoke` and delegates), the results could easily become much worse.

Conclusions, points of interest

Timers do not always behave as one would expect:

- the standard timer classes typically generate too large an interval; they will not reach their theoretical maximum frequency of 1000 Hz.
- the `Thread.Sleep()` method may or may not largely exceed the delay you asked for.
- on some systems, `Thread.Sleep(1)` will not even come close.

If better behavior is required for both .NET Framework versions 1.1 and 2.0, one can use:

- A Win32 performance counter for measuring elapsed times in the millisecond and submillisecond range.
- A Win32 multimedia timer for generating timer events up to 1000 Hz (make sure to keep the handler short!).

The test application uses the following basic techniques:

- `InvokeRequired` and `Invoke` to have timer event handlers interact with the form's controls.
- PInvoke to call native Win32 functions.
- Since the **`timeSetEvent`**`()` function needs a callback, the application shows how to have Windows call a user-supplied C# method.

History

- LP TimerTest# 1.0 (first release).

License

About the Author

~~**Luc Pattyn**~~

Software Developer (Senior)
Perceler
Belgium

Member

I am an engineer with a background in electronics, software and mathematics.

I develop technical software, both for embedded systems and for desktop equipment. This includes operating systems, communication software, local networks, image processing, machine control, automation, etc.

I have been using all kinds of microcontrollers and microprocessors (Intel 4004/8080/8051/80386/Pentium, Motorola 680x/680x0/ColdFire/PowerPC, Microchip PIC, Altera NIOS, and many more), lots of programming languages (all relevant assemblers, Fortran, Basic, C, Java, C#, and many more), and different operating systems (both proprietary and commercial).

For desktop applications and general development tools I have been using both UNIX systems and Mac/MacOS for many years, but I have switched to x86-based PCs with Windows, Visual Studio and the .NET Framework several years ago.

I specialize in:
- cross-platform development (making software that runs on diverse hardware/OS combinations)
- instruction set simulation
- improving software performance, i.e. making sure the software runs the job at hand in as short a time as possible on the given hardware. This entails algorithm selection, implementation design, accurate measurements, code optimisation, and sometimes implementing virtual machines, applying SIMD technology (such as MMX/SSE), and more.