

# Getting a Millisecond-Resolution DateTime under Windows CE

November, 2007

Chris Tacke

OpenNETCF Consulting, LLC

## Introduction

One seemingly strange behavior that Windows CE devices (including Pocket PC, Smartphone and Windows Mobile devices) exhibit is that when you query the device time, you get back either a zero or some constant but invalid value for the milliseconds field. This is true whether you call `GetLocalTime` or `GetSystemTime` in native code or `DateTime.Now` in managed code. Why is it that devices running an OS that has real-time capabilities can't provide us as developers with a clock that has simple millisecond resolution? The answer lies not in the OS – which is quite capable of giving that resolution – but in how the time function is implemented by the device OEM.

Typically Windows CE devices contain a piece of hardware internally called a real time clock or RTC. The RTC is very accurate, but often has a resolution of only 1 second. When the OS was implemented, most OEMs simply return the value that the RTC holds and therefore when you query the time that's all you get.

So the question remains – how can we as developers get a millisecond-resolution time? If you're simply looking to get the time short tasks take, then often calling `GetTickCount` or `Environment.TickCount` (which is the number of milliseconds since the OS started and tied to a separate oscillator in the processor) is sufficient, but what if we want a true time stamp with a millisecond field? The answer is that we simply have to calculate it. In this white paper we'll look at code that does that calculation and to make our code reusable, we'll create a simple class called `DateTime2` that will expose a `Now` property that has millisecond resolution.

## Calculating the millisecond field

Since we know that the `Environment.TickCount` returns the number of milliseconds since startup, we know that we have access to data that can become our millisecond field. By using modulo division on it we can get a "current millisecond" piece of data:

```
int tick = Environment.TickCount % 1000;
```

However the problem with this is that there's only a 1 in 1000 chance that it will be synchronized with the system clock, so it's more than likely that this millisecond tick will roll over to zero at a time when the system clock is not rolling over. All we have to do then is manually synchronize that rollover. To do that we simply need to find what the value of the above tick is when the clock rolls its second value over. So during initialization of our class, we simply poll the clock repeatedly and quickly until we see

the second roll over, then we calculate the offset. Bear in mind that this initialization can take up to 1 second to complete.

```
private static int m_offset = 0;

static DateTime2()
{
    int s = DateTime.Now.Second;
    while (true)
    {
        int s2 = DateTime.Now.Second;

        // wait for a rollover
        if (s != s2)
        {
            m_offset = Environment.TickCount % 1000;
            break;
        }
    }
}
```

So now that we have our offset, we simply use it to calculate our millisecond field every time the Now property is retrieved.

```
public static DateTime Now
{
    get
    {
        // find where we are based on the os tick
        int tick = Environment.TickCount % 1000;

        // calculate our ms shift from our base m_offset
        int ms = (tick >= m_offset) ? (tick - m_offset) : (1000 -
(m_offset - tick));

        // build a new DateTime with our calculated ms
        // we use a new DateTime because some devices fill ms with a
non-zero garbage value
        DateTime now = DateTime.Now;
        return new DateTime(now.Year, now.Month, now.Day, now.Hour,
now.Month, now.Second, ms);
    }
}
```

### **Testing our code**

So now that we have a function that is logically sound, how do we check that it really works? First, toss out any idea that you might test or use this on the emulator. The emulator emulates hardware, including the RTC and the system tick and is therefore completely unreliable for any type of time testing (I actually tested in with this code and saw major creep of the millisecond field across the second rollover period).

To test the function on actual hardware, we use logic that is pretty much the same as our offset initialization code. We simply want to watch the system clock roll over, and when it does, we want to get the millisecond field of our DateTime. It should be at or near zero, and more importantly it should be consistent. So let's look at what a test function might look like.

```
private void Test()
{
    int seconds = 5;
    int s = DateTime2.Now.Second;
    while (seconds > 0)
    {
        DateTime dt = DateTime2.Now;
        int s2 = dt.Second;
        if (s != s2)
        {
            System.Diagnostics.Debug.WriteLine("Test ms=" +
dt.Millisecond);
            s = dt.Second;
            seconds--;
        }
    }
}
```

When I ran this on a Dell Axim x51 I found that the tick repeatedly came out around 998 or so, meaning that my millisecond field would actually roll to zero a couple milliseconds before the actual second field. For many applications this might be fine, but of course we can do better, right? If we can test the inaccuracy, we can always account for it. So let's add a Calibrate function to our DateTime2 class that will run our test code logic for a period of time, calculate the average offset from zero over that period and then adjust our global offset by that average.

```
public static void Calibrate(int seconds)
{
    int s = DateTime2.Now.Second;
    int sum = 0;
    int remaining = seconds;
    while (remaining > 0)
    {
        DateTime dt = DateTime2.Now;
        int s2 = dt.Second;
        if (s != s2)
        {
            System.Diagnostics.Debug.WriteLine("ms=" + dt.Millisecond);
            remaining--;
            // store the offset from zero
            sum += (dt.Millisecond > 500) ? (dt.Millisecond - 1000) :
dt.Millisecond;
            s = dt.Second;
        }
    }

    // adjust the offset by the average deviation from zero (round to
the integer farthest from zero)
    if (sum < 0)
```

```
    {
        m_offset += (int)Math.Floor(sum / (float)seconds);
    }
    else
    {
        m_offset += (int)Math.Ceiling(sum / (float)seconds);
    }
}
```

After I ran Calibrate for 5 seconds on the same device, I found that Test spit out an offset of zero very repeatedly, which is what we want.

### **Conclusion**

So we now understand that most CE devices don't fill the milliseconds field of their system time because it's not readily available, and we've seen that with a little extra code we can derive a reasonably accurate replacement that does. Our new DateTime2 class provides a reasonable utility class that can be used in a lot of applications where you'd like to have a millisecond field.

Keep in mind, however, that what we have here is still based on two separate oscillators on the device and those two will drift apart over time. Your offset might be zero today, but if you run the app for a week I would expect you'll find it to have drifted a few ticks - how much depends on the hardware, its layout and the temperature the device is at. Also keep in mind that managed code is still nondeterministic. Just because you get a millisecond accurate time doesn't mean that you can reliably do tasks with millisecond accuracy.