Visual Studio Magazine Online
Classic VB Corner

Measuring Optimizations for Classic VB

We all tend to obsess on optimizations at times, often needlessly. Here's how to figure out if all your extra work is paying off.

May 18, 2010 · by Karl E. Peterson

We've all spent needless time optimizing things that don't matter, often as a point of pride, right? We might need to actually show someone else that code someday, and it just wouldn't do to be wasting cycles. Of course, this can and often does lead to needlessly unreadable code, which is more often than not what that someone will be cursing us out for somewhere down the road.

That said, there obviously are actual bottlenecks that do need to be addressed from time to time. It could be that you've worked up several approaches to a given problem, and you want to know exactly which will perform better. It's also useful to be able to weigh the utility of obscure, barely-readable code by quantifying the actual time savings it provides.

Long ago I wrote up a drop-in ready CStopWatch class that has resided in my Templates folder ever since. CStopWatch works by reprogramming the multimedia timer chip to run at its maximum resolution, typically 1 millisecond. It then effectively divides that uncertainty in half, by waiting until right after a timer tick before returning from a reset.

The multimedia timer API consists of just a handful of functions. We start by calling timeGetDevCaps to determine the minimum and maximum resolution the hardware is capable of. Note the MSDN docs are a bit mixed up between period and resolution, inappropriately using these terms interchangeably. The minimum period is the maximum resolution, and vice-versa, of course.

Although it sounds dangerous, reprogramming the multimedia timer is very straightforward. The only caveat is that your tweaks to this hardware are global, and can therefore affect all running applications. You call timeBeginPeriod to set a new resolution by requesting the shortest period that is acceptable for your needs. Each call to timeBeginPeriod must be paired with a call to timeEndPeriod to restore the previous setting.

Wrapping this functionality in a class helps enforce this clean-up by leveraging the Terminate event. The loss of this automated clean-up methodology in VFred was a huge blow, and CStopWatch serves to highlight once again the utility of deterministic finalization in the Classic VB toolbox.

CStopWatch has only one meaningful property and one heavily used method. You'll want to call the Reset method before any timing run, and then check the Elapsed property when the work is done. Reset loops until the value returned from timeGetTime changes, then stores that new value as the starting time for later measurements using the Elapsed property.

Let's move right into an example, which compares exponentiation against multiplication:

```
Public Sub TimeSquares()
   Dim stp As CStopWatch
   Dim i As Double, n As Double
   Dim d1 As Long, d2 As Long, d3 As Long
   Dim msg As String
   Const Loops As Long = 1000000

   ' Initialize the stopwatch.
   Set stp = New CStopWatch

   ' Time the empty loop.
   stp.Reset
   For i = 1 To Loops
   Next i
   d1 = stp.Elapsed

   ' Time the exponentiation operator.
   stp.Reset
   For i = 1 To Loops
      n = i ^ 2
   Next i
   d2 = stp.Elapsed

   ' Time the multiplication operator.
   stp.Reset
   For i = 1 To Loops
      n = i * i
   Next i
   d3 = stp.Elapsed

   msg = "Empty Loop: " & CStr(d1) & "ms" & vbCrLf & _
         "Exponents: " & CStr(d2 - d1) & "ms" & vbCrLf & _
         "Multiply: " & CStr(d3 - d1) & "ms" & vbCrLf & _
         "Loops: " & CStr(Loops)
   Debug.Print msg
   Clipboard.Clear
   Clipboard.SetText msg
   MsgBox msg, , "Benchmarks"
End Sub
```

This is a bit of a contrived example, as some of the advanced compilation optimizations will virtually wipe out the results of the last loop. But it does serve as a template for testing various ways of doing things.

Obviously, both exponentiation and multiplication happen almost instantly, so we need to do either operation a great deal of times to be able to tell which is actually taking longer. Here, we use a one million iteration loop.

The first step is to time how long the timing mechanism itself -- the empty loop -- takes. It's best to do timing code directly inline, but if the code you want to time resides in separate subroutines, you'll want to include a call to a dummy routine in the empty loop to better simulate the target operation.

Next, we replicate the timing loop code, but inject the method of interest in each subsequent loop. First by assigning the value of an exponentiation, and then by computing the same value using multiplication. Before each loop we reset the stopwatch, and after each loop we store the number of elapsed milliseconds.

Finally, the results are reported in a number of ways. We will want to use Debug.Print only at first, to insure our tests are setup correctly. Any real benchmark needs to be run with code compiled to use the same optimizations as you intend to compile the final product with. That's why we also pop a message box for immediate feedback, and put the results on the clipboard so that they might be pasted into supporting docs (like this!):

```
Empty Loop: 3ms
Exponents: 483ms
Multiply: 13ms
Loops: 1000000
```

Surprised? That happens a lot when you actually time different algorithms. You can download the complete Stopwatch sample from my Web site. I hope CStopWatch.cls earns a place in your Templates folder!

About the Author

Karl E. Peterson wrote Q&A, Programming Techniques, and various other columns for VBPJ and VSM from 1995 onward, until Classic VB columns were dropped entirely in favor of other languages. Similarly, Karl was a Microsoft BASIC MVP from 1994 through 2005, until such community contributions were no longer deemed valuable. He is the author of VisualStudioMagazine.com's new Classic VB Corner column. You can contact him through his Web site if you'd like to suggest future topics for this column.