Visual Studio Magazine Online
Classic VB Corner

Determining Process Times

ONLINE ONLY

Here's the least complicated method for determining how long a given process has been running or how much CPU time it's actually consumed. This technique is used here to establish how long Windows has been running.

May 5, 2009 · by Karl E. Peterson

In my last column, I showed how to grab a random structure from binary data stored in the registry. Actually, it wasn't all that random of a choice. I was looking for a way to determine how long Windows had been running, and it turned out that a timestamp was stored with each shutdown. Or more properly, with each *proper* shutdown, crashes not included. Retrieving the structure was still a useful exercise, though, as we'll see later.

To truly determine "uptime" for the system, it would be more consistent to find a metric of when the system came up, rather than when it went down. Windows provides a GetProcessTimes API, which will tell us when any given process was created, as well as how much CPU time it's consumed in user and kernel modes. Jeff Atwood published a nice piece on the difference between user and kernel modes about a year ago, if you're wondering about that.

To use GetProcessTimes, you simply need a handle to the process of interest, and it must have PROCESS_QUERY_INFORMATION access rights. Using Process Explorer, a few good process candidates pop right out. It would appear that smss.exe, csrss.exe and winlogon.exe are the first three processes to fire up. I chose to focus on winlogon.exe, as that's the one that will actually facilitate the log-on process, and the system can't truly be considered "up" unless it's willing to let a user log on, right?

So we know the name of the executable we want to query, and the trick now is getting a handle to that process. In order to obtain a handle, we need to first determine the process ID. The CreateToolhelp32Snapshot API will take a snapshot of all running processes, and we can then just loop through the list looking for a match. The following routine will accept either a filename or a known process ID, and find the other by looping through the snapshot. Both values are then stored in module-level variables for use in other routines:

```
Private Sub ProcessFind(Optional ByVal ExeFile As String, _
                        Optional ByVal PID As Long)
   Dim hSnap As Long
   Dim ProcEntry As PROCESSENTRY32
   Dim Found As Boolean

   ' Clear cached values.
   m_PID = 0
   m_ExeFile = ""
   ' Start enumeration (9x/2000+)
   hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0&)
   If hSnap Then
      With ProcEntry
         .dwSize = Len(ProcEntry)
         ' Iterate through the processes
         If Process32First(hSnap, ProcEntry) Then
            Do
               If PID Then
                  Found = (PID = .th32ProcessID)
               ElseIf Len(ExeFile) Then
                  Found = (StrComp(TrimNull(.szExeFile), _
                           ExeFile, vbTextCompare) = 0)
               End If
               If Found Then
                  m_ExeFile = TrimNull(.szExeFile)
                  m_PID = .th32ProcessID
                  Exit Do
               End If
            Loop While Process32Next(hSnap, ProcEntry)
         End If
      End With
   End If
End Sub
```

Now that we have the process ID we're after, calling OpenProcess will give us the needed handle:

```
Private Sub ProcessOpen()
   Const opFlags As Long = PROCESS_QUERY_INFORMATION
   If (m_hProcess = 0) And (m_PID <> 0) Then
      m_hProcess = OpenProcess(opFlags, False, m_PID)
   End If
End Sub
```

I tend to stuff code like this, which operates on a given object, into class modules. Here again, you'll see I'm using module-level variables to store properties of the object (process). The test to see whether I already have a process handle is a bit of a shortcut in that I tend to hold the handle open as long as I'm working with it, or for the life of the class. At this point, our process class offers cached ExeName, ProcessID and hProcess properties.

As I mentioned above, GetProcessTimes is plural -- it retrieves four different times related to the specified process. The following property of my class will report on any one of them:

```
Public Enum ProcessTimes
   [_ProcessTimesMin] = 0
   piCreationTime = 0
   piExitTime = 1
   piKernelTime = 2
   piUserTime = 3
   [_ProcessTimesMax] = 3
End Enum
```

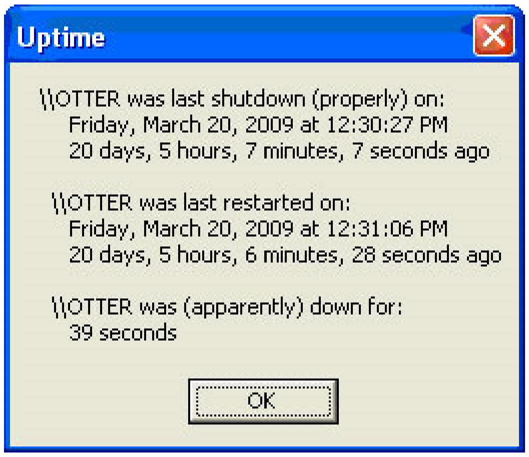
```
Public Property Get ProcessTime(ByVal WhichTime As ProcessTimes) _
                                                           As Date
   Dim pt(0 To 3) As FILETIME
   If (WhichTime >= [_ProcessTimesMin]) And _
      (WhichTime <= [_ProcessTimesMax]) Then
      Call ProcessOpen
      If m_hProcess Then
         Call GetProcessTimes(m_hProcess, _
                              pt(0), pt(1), pt(2), pt(3))
         If pt(WhichTime).dwHighDateTime <> 0 And _
            pt(WhichTime).dwLowDateTime <> 0 Then
            ProcessTime = FileTimeToDouble(pt(WhichTime), True)
         End If
         Call ProcessClose
      End If
   End If
End Property
```

The final clean-up is to close the open process handle. The process class has an option to hold the process open, which is slightly more efficient if you need to query multiple properties of the process. The force parameter to ProcessClose is used in the class Terminate event, to ensure no leaked handles.

```
Private Sub ProcessClose(Optional ByVal Force As Boolean)
   If m_hProcess Then
      If (m_HoldOpen = False) Or (Force = True) Then
         Call CloseHandle(m_hProcess)
         m_hProcess = 0
      End If
   End If
End Sub
```

So what use is that value for last shutdown we dug out of the registry? Well, it can be interesting to look at the difference between the process time of winlogon and the time of last shutdown, which roughly equates to "downtime," thus doubling the information our new little uptime utility provides. Remember, though, the last shutdown timestamp is only valid if the system was shutdown properly.

You can download the fully functional utility with all the code from this and my last column, and a bonus version compiled to run at the console using (redirectable) standard output, from my Web site.



```
\\OTTER was last shutdown (properly) on:
   Friday, March 20, 2009 at 12:30:27 PM
   20 days, 5 hours, 7 minutes, 7 seconds ago

\\OTTER was last restarted on:
   Friday, March 20, 2009 at 12:31:06 PM
   20 days, 5 hours, 6 minutes, 28 seconds ago

\\OTTER was (apparently) down for:
   39 seconds
```

Figure 1. ClassicVB authored Uptime utility output. Note the uncertainty over "apparent" downtime, as we're relying on a proper shutdown for this bonus information.

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.