Visual Studio Magazine Online
Classic VB Corner

Mining the Registry for Structures

ONLINE ONLY

Need to extract binary data from the registry? Here's a quick primer on reading and interpreting structures stored there.

April 20, 2009 · by Karl E. Peterson

Classic VB has always offered a couple of really crude wrappers for reading and writing to the Windows registry. They were hobbled from the beginning by being restricted to a single subkey under the CurrentUser (HKCU) hive. So most folks have found or written wrappers to read and write string and dword keys. But many of the published ones don't offer nice, easy, binary operations, so I thought I'd share the method I use.

I was playing around with reproducing a little utility that determines -- estimates, really -- how long Windows has been running. One of the methods to do this is to read the "ShutdownTime" value stored at *HKLM\System\CurrentControlSet\Control\Windows*. Well, right away, that rules out using native VB methods, as it's in the LocalMachine hive. Worse still, from this perspective, it's stored as a FILETIME structure in binary format. Definitely no native support for that.

The FILETIME structure is simply a convenient way to pack a 64-bit value into manageable 32-bit chunks, and represents the number of nanoseconds since Jan. 1, 1601 (UTC, of course). You can represent FILETIME in ClassicVB as such:

```
Public Type FILETIME
   dwLowDateTime As Long
   dwHighDateTime As Long
End Type
```

So the problem comes down to being able to grab some arbitrary number of bytes from a binary registry entry, and then stuff them into a predefined structure. The answer was to write a generic registry reading routine that would return a dynamic byte array. First the code, then the clues:

```
Public Function RegGetBinaryValue(ByVal RootKey As Long, _
   ByVal Key As String, ByVal ValueName As String, _
   TheData() As Byte) As Boolean

   Dim nRet As Long
   Dim hKey As Long
   Dim nType As Long
   Dim nBytes As Long
   Dim DWord As Long

   ' Open key
```

```
   nRet = apiRegOpenKeyEx(RootKey, Key, 0&, KEY_READ, hKey)
   If nRet = ERROR_SUCCESS Then
      ' If NULL, the default value will be read.
      If ValueName = "*" Then ValueName = vbNullString

      ' Determine how large the buffer needs to be
      nRet = apiRegQueryValueEx(hKey, ValueName, 0&, nType, _
                                ByVal 0&, nBytes)
      If nRet = ERROR_SUCCESS Then
         If (nType = REG_BINARY) Then
            ' Resize buffer and request data at this key, ...
            ReDim TheData(0 To nBytes - 1) As Byte
            nRet = apiRegQueryValueEx(hKey, ValueName, 0&, _
                                      nType, TheData(0), nBytes)
            If nRet = ERROR_SUCCESS Then
               ' ... and return success.
               RegGetBinaryValue = True
            End If
         End If
      End If
      Call apiRegCloseKey(hKey)
   End If
End Function
```

The registry functions are very well-named. So much so, in fact, that I ended up recycling some of them as function names in my standard registry wrappers, so I needed to use aliases in all my declares. You can see the standard I settled on was preceding the API function names with the "api" prefix, while my own function names took the more conversational names.

The RegGetBinaryValue function accepts root key (typically HKCU or HKLM), key and value name parameters to point to what data is to be retrieved. Special handling is given when the value name of "*" is used -- that returns the default value for the key. The function also accepts a pointer to a dynamic array, which is resized to fit and filled with the available data from the requested location.

Calling RegGetBinaryValue can be done in this manner:

```
Public Function LastShutdown() As Date
   Dim b() As Byte
   Dim ft As FILETIME
   Const HKEY_LOCAL_MACHINE = &H80000002
   Const Key As String = _
      "System\CurrentControlSet\Control\Windows"
   Const Value As String = "ShutdownTime"

   If RegGetBinaryValue(HKEY_LOCAL_MACHINE, Key, Value, b) Then
      If UBound(b) = 7 Then
         Call CopyMemory(ft, b(0), 8&)
```

```
         LastShutdown = FileTimeToDouble(ft, True)
      End If
   End If
End Function
```

In this case, I'm expecting a specific structure to be stored in the binary value, one composed of exactly eight bytes. So I pass an empty dynamic array to RegGetBinaryValue, and then test the size of the array upon successful return. If I was handed eight bytes, a quick call to CopyMemory slings them over into a FILETIME structure. That's all there is to it.

What's that FileTimeToDouble call in there, you say? That's a little routine I wrote to convert API date/time values into more VB-friendly values. It exercises a couple more APIs to return a value our code can understand intrinsically:

```
Private Function FileTimeToDouble(ftUTC As FILETIME, _
   Optional ByVal Localize As Boolean = False) As Double
   Dim ft As FILETIME
   Dim st As SYSTEMTIME
   Dim d As Double
   Dim t As Double

   ' Convert to local filetime, if necessary.
   If Localize Then
      Call FileTimeToLocalFileTime(ftUTC, ft)
   Else
      ft = ftUTC
   End If

   ' Convert to system time structure.
   Call FileTimeToSystemTime(ft, st)

   ' Convert to VB-style date (double).
   FileTimeToDouble = DateSerial(st.wYear, st.wMonth, st.wDay) + _
                      TimeSerial(st.wHour, st.wMinute, st.wSecond)
End Function
```

Windows typically stores all time values in UTC, and then converts them to the local time zone upon demand. So the first task when converting FILETIME structures is often to pass them to FileTimeToLocalFileTime which does just as its name suggests, including accounting for daylight-saving time. The next step is to break the encoded low and high values of the FILETIME structure into a much more readable SYSTEMTIME structure using the FileTimeToSystemTime API.

SYSTEMTIME structures neatly map into exactly the values we need to provide DateSerial and TimeSerial, such that we end up with the native data format used by ClassicVB for dates and times. Yes, a Date variable is simply a Double in drag.

The Uptime sample on my site contains this code, and much more which I'll be touching on in forthcoming columns. Next, we'll look at why this registry value may not be the best indicator of system uptime, and what's a better way to estimate that elusive value.

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.

1105 Redmond Media Group
Copyright 1996-2009 1105 Media, Inc. See our Privacy Policy.