WINDOWS PROGRAMMING

NT Events are a Terrible Thing to Waste

Click & Retrieve Source CODE!

Read event logs on Windows NT with 32-bit VB 4.0 code and 32-bit API calls.

by L.J. Johnson

If you aren't running a Windows NT server or workstation, don't read any further. If you are not an administrator for an NT server and/or workstation, or if you never have hardware or software problems on NT, this article isn't for you. But if any of this applies to you, you probably already know something about the event logs. Through a defined API, you can use the event logs as a single place to store and retrieve information about security events, system events, or application events. In this column I'll demonstrate how to read the event logs programmatically from the 32-bit version of Visual Basic 4.0.

Windows NT creates and maintains a set of event logs: System, Security, and Application. Physically, the three logs are stored as three files in the \%SystemRoot%\system32\config directory as: SysEvent.Evt, SecEvent.Evt, and AppEvent.Evt (where %SystemRoot% is the Windows NT directory). Examine the HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog key in the 32-bit registry for further information on these logs. If you haven't yet used the Event Viewer applet in NT, bring it up now and browse the three event logs before you read the rest of column—doing so will make it easier to understand the concepts.

You can look at the event logs on your local machine, or on any other NT computer on the network to which you have read rights to that event log. It is not a trace facility, and it does consume resources such as disk space and logging time, but it can be invaluable for tracking subtle problems. For example, consider an application that logs all low-memory conditions. Several low-memory occurrences in the log might indicate that the system needs more memory in order to successfully run this application, or that the application has a bug that causes a memory leak over time.

The System log records events logged by the NT system components. You do not have to enable system logging—it occurs automatically. On my system, the System log contains these records:

- Application Popup: System Popup—Wrong Volume. The wrong volume is in the drive (This refers to my CD-ROM drive. I had changed a CD too early).
- The browser has forced an election on network \Device\Nbf_Elnk32 because a master browser was stopped.
- The Event Log service was started.
- Debug: ShareThisPrinter NetShareDel failed, Printer LP_LaserJet3 sharename LJet3, Error 2114, deleting share.
- The redirector has timed out a request to ntbr61. (I got this RAS message when I lost the connection to the remote server).
- User configuration data for parameter COM1 overriding firmware configuration data.

L.J. Johnson is a technical consultant on the Mary Kay InTouch project at Mary Kay Cosmetics in Dallas, Texas. He has been programming in Visual Basic since 1.0, and is a section leader on the VBPJ *Forum on CompuServe. Reach him on CompuServe at 74777,3047.*

The Security log records security events. You must enable security logging in the User Manager, under the Audit menu item of the Policies menu, for success or failure of such things as:

- Logon and logoff.
- File and object access.
- Use of user rights.
- User and group management.
- Security policy changes.
- Restart, shutdown, and system.
- Process tracking.

Using these settings in conjunction with the settings in File Manager under the Auditing menu item of the Security menu, you can actually look at who reads, writes, executes, deletes, or changes permissions on a single file or group of files.

The Application log is perhaps the most interesting of all the logs. The Security log is designed primarily for system administrators, although logging which users accessed or executed a particular file might be useful for some applications. The System log is a mixed bag.

The Application log, however, is utilized by applications—those things we build for a living. For example, you might build a program to automatically check an entire group of NT servers and workstations daily for certain types of events in the Applications log (or, for that matter, the System or Security log), and warn the network supervisor if any were found. Or you might want to save certain types of event log entries to a text file before clearing (deleting) that log. I won't cover writing to the Application log in this column, but it is possible to do it from VB. That topic might make a good future column.

Reading an event log is more complicated than you might first expect. It requires a number of 32-bit API calls, and a number of workarounds to some of the limitations of VB. Because of the complexity, the entire functionality is encapsulated in an in-process OLE server.

I should say something here about error handling: the VB 4.0 manuals give two ways to pass error information from the OLE server back to the calling application—using the Err object and raising an error in the calling app, or passing the error information back from each exposed method or property. Because I did not like either of these options, I used a third. The two main public methods, OpenAnyEventLog and ReadEventEntries, return False if any errors occur while you're opening or reading

VB4

```
Public Function ReadEventEntries() As Long
 Dim pbyteBuffer() As Byte
 'missing declarations

 'missing code to check EventType has been
 ' set and set the starting date and time bias

 ' Make sure that the event log has been opened
 If mlngEventLogHwd = &HFFFFFFFF Then
   ' Has never been set -- see Initialize event
   LastEventErrorNumber = (ERR_LOG_NOT_OPENED + _
     vbObjectError)
   LastEventErrorSource = EVENT_SOURCENAME & _
     "ReadEventEntries"
   LastEventErrorDescription = "The Event Log” & _
     “has not been opened yet.  Use” & _
     OpenAnyEventLog property."
   ReadEventEntries = False
   mlngCount = 0
   Exit Function
 ElseIf mlngEventLogHwd <> 0 Then
   plngEventLogHwd = mlngEventLogHwd
 Else
   ' Error info set when set the Server Name
   ReadEventEntries = False
   mlngCount = 0
   Exit Function
 End If

 ' See how many event records are in this log
 If API_NumberOfEventLogRecords(plngEventLogHwd) = _
   True Then
   plngNumRecords = CountEventRecords
 End If

 If plngNumRecords <= 0 Then
   ' No records -- set number of records to zero
   ' and exit the function
   ReadEventEntries = False
   mlngCount = 0
   Exit Function
 Else
   ' Redimension the type array to
   ' total number of records
   ReDim colEventRecord(1 To plngNumRecords) _
     As EventRecord
 End If

 If mlngEventReadLogForward = True Then
   plngReadFlags = EVENTLOG_SEQUENTIAL_READ Or _
      EVENTLOG_FORWARDS_READ
 Else
   plngReadFlags = EVENTLOG_SEQUENTIAL_READ Or _
      EVENTLOG_BACKWARDS_READ
 End If

 'missing code to initialize variables

 ' Loop thru the record numbers
 Do While plngNumUnfilteredRecords < plngNumRecords

   plngNumBytesToRead = 1024 * (plngNumRecords - _
     plngNumUnfilteredRecords)
   If plngNumBytesToRead > 16384 Then
     plngNumBytesToRead = 16384
   ElseIf plngNumBytesToRead < 4096 Then
     plngNumBytesToRead = 4096
   End If

TryAgain:
   ReDim pbyteBuffer (0 To plngNumBytesToRead - 1)

   ' Read the event log (multiple records) and
   ' check for errors
   plngRtn = ReadEventLog(plngEventLogHwd, _
     plngReadFlags, plngReadRecordOffset, _
     pbyteBuffer(0), plngNumBytesToRead, _
     plngNumBytesRead, plngMinNumBytesNeeded)

   If plngRtn = False Then
     If Err.LastDllError = ERROR_HANDLE_EOF Then
       ' End of the records
       Exit Do
     ElseIf Err.LastDllError = _
       ERROR_INSUFFICIENT_BUFFER Then
       ' OK, 16K wasn't big enough -- set to
       ' the value returned by the function
       ' for the number of bytes needed
       plngNumBytesToRead = plngMinNumBytesNeeded
       GoTo TryAgain
     Else
       'missing code to set the substitute error
       'properties as above
       ReadEventEntries = False
       mlngCount = 0
       Exit Function
     End If
   End If

   ReDim Preserve pbyteBuffer(0 To _
     plngNumBytesRead - 1)
   pstrMultiRecBuffer = Space$(plngNumBytesRead)
   CopyMem ByVal pstrMultiRecBuffer, _
     pbyteBuffer(0), plngNumBytesRead
   Erase pbyteBuffer

   ' Loop for each record in the buffer
   Do While Len(pstrMultiRecBuffer) > 0

     If plngNumUnfilteredRecords > _
       plngNumRecords Then
       Exit Do
     End If

     plngNumUnfilteredRecords = _
       plngNumUnfilteredRecords + 1

     ' Get the length of the next record and
     ' set the buffer to that length
     plngRecLen = CVL(Mid$(pstrMultiRecBuffer, 1, 4))
     pstrBuffer = Mid$(pstrMultiRecBuffer, _
       1, plngRecLen)
     pstrMultiRecBuffer = Mid$(pstrMultiRecBuffer, _
       plngRecLen + 1)

     ' Get the raw info from the event log
     plngRecNum = CVL(Mid$(pstrBuffer, 9, 4))
     pvntAddDate = CVL(Mid$(pstrBuffer, 13, 4))
     pvntTimeGenerated = DateAdd("s", _
       CLng(pvntAddDate) - plngBiasTimeSecs, _
       pvntStartDate)
     plngEventID = CVL(Mid$(pstrBuffer, 21, 2))
     plngEventIdForFile = CVL(Mid$(pstrBuffer, _
       21, 4))
     pintEventType = CVI(Mid$(pstrBuffer, 25, 2))
     'missing code to extract rest of items
     'from the buffer

     ' Get the EventSourceName and
     ' EventComputerName strings
     pstrTmp = Mid$(pstrBuffer, 57)
     On Error Resume Next
     plngPos = InStr(pstrTmp, Chr$(0))
```

CONTINUED ON PAGE 106.

LISTING 1 ***Read All About It.*** *The ReadEventEntries method is the main interface method for the server. It reads all the entries in the log you selected through the OpenAnyEventLog property, and filters the records if requested. It also fills in a private type array for each log item in the filtered or unfiltered list. Access the various pieces of each log item, such as EventID and Event Type, through properties and an index value in the calling program.*

the event logs. The same information that *could* be returned through the Err properties is instead returned through the LastEventErrorNumber, the LastEventErrorSource, and the LastEventErrorDescription public properties.

OPEN SESAME

The OpenAnyEventLog method takes a single parameter, ServerName, and returns either True or False. Before calling this method, you must set the TypeEventLog. If you do not set the

CONTINUED FROM PAGE 105.

```
           If plngPos > 0 Then
              pstrEventSourceName = Mid$(pstrTmp, 1, _
                 plngPos - 1)
           End If

           If plngPos > 0 Then
              pstrTmp = Mid$(pstrTmp, plngPos + 1)
              plngPos = InStr(pstrTmp, Chr$(0))
              If plngPos > 0 Then
                 pstrEventComputerName = Mid$(pstrTmp, _
                    1, plngPos - 1)
              End If
           End If

           ' Get the message string
           If pintNumStrings > 0 Then
              pstrStrings = Mid$(pstrBuffer, _
                 plngStringOffset + 1, _
                 (plngDataOffset - plngStringOffset))
           End If
           pstrMsgString = _
              ResourceString(pstrEventSourceName, _
              pintNumStrings, pstrStrings, _
              plngEventIdForFile)

           Select Case CLng(pintEventType)
              Case EVENTLOG_SUCCESS
                 pstrEventType = "Success"
              'missing code to set string for other
              'types of EventType return value
           End Select

           ' For no filter, all records match
           If mlngFilterType = Filter_Type_None Then
              plngIsMatch = True

           ' Have a filter -- this record may or
           ' may not match
           Else
              plngIsMatch = False
              Select Case mlngFilterType
                 Case Filter_Type_TimeBefore
                    If CDate(pvntEventTimeWritten) < _
                       CDate(mvntFilter) Then
                       plngNumFilteredRecords = _
                             plngNumFilteredRecords + 1
                       plngIsMatch = True
                    End If
                 Case Filter_Type_EventID
                    If CLng(plngEventID) = _
                       CLng(mvntFilter) Then
                       plngNumFilteredRecords = _
                             plngNumFilteredRecords + 1
                       plngIsMatch = True
                    End If
                 'missing code to handle other
                 'types of filters
              End Select
           End If

           ' Only add to array if this is a matching
           ‘ record (with no filter, all records match)
           If plngIsMatch = True Then
              If mlngFilterType = Filter_Type_None Then
                 plngRecCount = plngNumUnfilteredRecords
              Else
                 plngRecCount = plngNumFilteredRecords
              End If

              colEventRecord(plngRecCount)._
                 EventRecordNum = plngRecNum
              colEventRecord(plngRecCount)._
                 EventTimeWritten = pvntEventTimeWritten
              'missing code to fill rest of type array

           If plngSidLength = 0 Then
              colEventRecord(plngRecCount)._
                 EventUserName = "N/A"
           Else
              pstrTmp = Mid$(pstrBuffer, _
                 plngSidOffset, plngSidLength)
              pstrSid = ""
              For pintInnerLoop = 1 To plngSidLength
                 pstrSid = pstrSid & _
                    Format$(Hex$(Asc(Mid$(pstrTmp, _
                    pintInnerLoop, 1))), "00") & " "
              Next pintInnerLoop
              colEventRecord(plngRecCount)._
                 EventUserName = pstrSid
           End If

           If plngDataLength > 0 Then
              pstrTmp = Mid$(pstrBuffer, _
                 plngDataOffset + 1, _
                 plngDataLength)
              pstrData = ""
              ' Convert to hex only if less than 256
              ' bytes or if the user wants it that way
              If mlngEventDataReturnHex = False Then
                 If Len(pstrTmp) > 256 Then
                    pstrData = pstrTmp
                 Else
                    For pintInnerLoop = 1 To _
                          plngDataLength
                          pstrData = pstrData & _
                                   Format$(Hex$(Asc_
                                   (Mid$(pstrTmp, _
                                   pintInnerLoop, _
                                   1))), "00") & " "
                    Next pintInnerLoop
                 End If
              Else
                 For pintInnerLoop = 1 To _
                    plngDataLength
                    pstrData = pstrData & _
                          Format$(Hex$(Asc(Mid$_
                          (pstrTmp, pintInnerLoop, _
                          1))), "00") & " "
                 Next pintInnerLoop
              End If
              colEventRecord(plngRecCount)._
                 EventData = pstrData
           Else
              colEventRecord(plngRecCount)._
                 EventData = "N/A"
           End If
        End If
     Loop
  Loop

  ' Re-adjust the number of items in array if it
  ‘ was filtered, and set the number of records
  ‘ in the module-level variable
  If mlngFilterType <> Filter_Type_None Then
     ReDim Preserve colEventRecord(1 To _
        plngNumFilteredRecords)
     mlngCount = plngNumFilteredRecords
  End If

  On Error GoTo 0
End Function
```

event log type first, or if you pass an incorrect event log type, the LastEventErrorNumber, LastEventErrorSource, and LastEventErrorDescription properties are set.

The OpenAnyEventLog method encapsulates the OpenEventLog API call, which takes only two parameters: the server name (in UNC format, although the server name itself will work if you are already attached to that server) and the type of log. That is why you use a Property Let to set the type of log, instead of a simple global variable as a property. With Property Let, you can check for valid types. Using a blank string for the server name opens the log on the local computer. The OpenAnyEventLog, if successful, also sets the private variable mlngEventLogHwd that is used by the other methods and properties in this class.

After opening the log, you have two choices concerning how to view the returned event log entries. The first choice is whether to filter the returned values. If you want to filter the values, set the EventFilter property before calling the ReadEventEntries method. If you do not need to filter the returned values, just call ReadEventEntries. Note that filtering the events will reduce the time for the OLE call, because the server's internal type array is filled with only the filtered records, not all the records for that log.

The second choice is whether you want to read the log backwards (latest events first) or forwards. By default the Event Viewer applet displays the log backwards, which is also the default for the ReadLogForward method (that is, FALSE = 0).

The EventFilter property has an interesting job—the filter itself can be a date/time stamp, a long, an integer, or a text string. In order to be able to check for valid parameters, the property also accepts a filter type parameter. VB does not allow constants from an OLE object to be exposed (like other OLE objects in the Object Viewer).

Here's where I get on my soapbox: how does the programmer using the OLE object know about these constants, or about the properties that he or she must set before calling a particular method? The only answer is a complete help file that you include with the object. Arguably, the most important part of an OLE server is the quality of the included help file. Generally, the programmer cannot see your code. He or she can see only the public interface. You should include examples to make it easy to cut and paste from the help file to the calling application. Think of your OLE server as a third-party tool you purchased by mail-order—it would be difficult or impossible to use that tool effectively without documentation or online help.

READ IT AGAIN, SAM

The heart of the program is the ReadEventEntries method (see Listing 1), which sets up an array (collection) of all the matching event records. At the beginning of the function, set the start date—event-log times are formatted in seconds past midnight on 01/01/1970—and the time bias, which is the difference in seconds between Coordinated Universal Time and local time. Actually, the API call GetTimeZoneInformation returns the time in minutes, but the private function GetTimeBias converts it to seconds, the format the event log requires.

Next, make sure the event log has been opened and use the private variable mlngEventLogHwd to ensure you have a valid event log handle. The NumberOfEventLogRecords property encapsulates the GetNumberOfEventLogRecords API call. Note that at this point the number of records is the number of unfiltered records.

The ReadEventLog call is an interesting exercise in working around some of VB's limitations. The primary problem is the EVENTLOGRECORD structure. Unlike most of the API typedefs, this particular one can change for each record that is read. As a result, declaring a Type variable in VB to match the C typedef is not possible. However, you can read this information with a byte array (new to VB 4.0). The CopyMem API function then converts the byte array to a string:

```
Public Declare Sub CopyMem _
   Lib "kernel32" Alias _
   "RtlMoveMemory" (dst As Any, _
   src As Any, ByVal Size As Long)
```

and the private CVL or CVI functions extract any numeric values from that string to longs or integers.

Another programming choice is whether to read only one record at a time, or read multiple event records at once. To read only one record at a time, set the number of bytes to be read (plngNumBytesToRead) to less than the length of a single record, call ReadEventLog, trap the error ERROR_INSUFFICIENT_BUFFER, and re-call the function with the number of bytes set to plngMinimumBytesNeeded. While this works, it is programmatically cleaner to read multiple records at one time. I tried it both ways, and saw no significant speed differences between the two, for either a large or a small number of total records.

DON'T TAKE TOO BIG A BITE

Note that I arbitrarily set the buffer size for a single read to 16K. Setting it to higher values may reduce the number of cycles in the loop, but may not necessarily reduce the total time the function needs. One big variable (pun intended) is the amount of memory installed on your NT computer. Feel free to experiment with various maximum buffer sizes. Note that some records may be bigger than 16K (thanks to Karl Peterson for sending a log indicating the problem—a utility called DrWatson can generate system dumps and include them in the log entry). In this case, I trap the ERROR_ INSUFFICIENT_BUFFER and reset the size to the plngMinimumBytes needed, as I explained.

After getting the records, "trim" the byte array to the size of the number of bytes actually read (the ReadEventLog API function returns only full records) by using the plngNumBytesRead return value. Byte arrays are hard to work with, so use the CopyMem API function to convert the array into a string.

The main loop parses the individual event records from the multiple-record string. Fortunately, after you convert the first four bytes of each record string into a long integer, you have the record length of that record. The CVL function uses CopyMem to convert the passed string (four bytes) into a long integer. Once the record length of the first record is determined, that string length is isolated into a new string for further processing.

Now that you have the string for just one event record, you can process that string into the individual elements of a type array. The functions CVL and CVI convert substrings to long integers or integers. The date substring uses the VB function DateAdd to add the number of seconds from the base timestamp to the timestamp returned from the event log. The program parses out the EventSourceName and ComputerName by looking for Chr$(0). The Message string, SID, and Data strings are located with the various offset and length values returned by the event record. Both SID and Data values are converted to hex before being put into the type array.

Actually, the inclusion of system dumps in some event logs presents an interesting problem. These dumps are stored as Data values, and tend to be large (12K to more than 16K). If the server translates them to hex, it will take a long time to return to the calling program: string concatenation in VB 4.0 is even slower than in VB 3.0. Because they are already in hex/ASCII format (hex on left), the data will be indecipherable.

As a kludge, you can set the property EventDataReturnHex. The default, if you don't set it, is false. When it's false, it returns the hex values for Data strings that are 256 bytes or fewer but untranslated values for strings of more than 256 bytes. All the values I have seen for Data (except for the dumps) were displayed as hex in the Event Viewer and were 32 bytes or fewer.

One interesting aside is the EventID. Use the EventID as a pointer to the correct string to look up in the resource file. Simply take the correct four bytes from the record and convert them to a long integer using CVL. But that number is not the same number as appears in the Event Viewer applet. In order to get the Event Viewer number, you must take only the two low-order bytes and convert them into a long integer. I guess Microsoft was concerned with the confusion that some of the very large numbers could create among the users of the Event Viewer—so they made it confusing to the programmer instead.

DO WE NEED AN INTERPRETER?

The Message string is the most complicated of all the items returned from the ReadEventLog API call. The message that is returned from the event record is not the message that you see in the Event Viewer. Most messages contain replaceable strings (for example, *%1*, *%2*, etc.).

The private function ResouceString encapsulates all the API calls you need to return the filled-in Message string. It gets the resource name using the passed SourceName to look up the full path of that resource in the 32-bit registry (through a private class module, RegistryDB). Any replaceable environmental parameters such as %SystemRoot% in the returned string are replaced with the ExpandEnvironmentalStrings API call. The full path is then used to load that resource (normally an EXE or DLL). You can find the ResourceString function in WP0396.ZIP, in the Magazine Library of the *VBPJ* Forum on CompuServe.

Next, any passed replaceable parameters are parsed into an array, and that array is passed to the function TranslateArray. This function hides a messy solution—the FormatMessage API call will accept only a type variable from VB, not an array, when you are using the FORMAT_MESSAGE_ARGUMENT_ARRAY bit flag. Thanks to Karl E. Peterson and Microsoft for this critical piece of information.

THE EVENT LOG'S FUNCTIONALITY IS ENCAPSULATED IN AN IN-PROCESS OLE SERVER.

The function FormatMsgFromResources encapsulates the FormatMessage API call. With this one API call, you can retrieve the specific message you need from the resource (based on the EventID from the event record), and replace all the replaceable parameters, if any.

Actually, it's a little more complicated than this, because the algorithm I described doesn't work for some of the entries (perhaps 1 percent or so). For these event log entries, when you retrieve the strings from the resource (used to replace the %1, and so on in the message string), those resource strings themselves have placeholders in the format %%1234.

So, if the returned resource string contains any "%%," that string is sent to another private function, ResourceString2. In order to find the resource file to look up the parameter, in you look at the registry again, but with the value ..\..\Parameter-MessageFile instead of ..\..\Event-MessageFile. The number immediately after the %% is used as a pointer to get the exact text string from the Parameter-MessageFile, just as with the original lookup into the Message file, and that string is used to replace the *%%xxxx* placeholder.

Finally, after all the data and strings for a particular event record have been resolved, each item is placed in its own element in a type array. Because VB 4.0 does not allow you to pass a UDT from an OLE server, the calling program will access each of these elements of the type array as a property, and pass an index value to indicate which item in the collection you are requesting. Because you're creating your own collection, more or less, the indexes will run from 1 to the number of event log items. Then, go to the top of the loop to either read the next event record from the already-retrieved byte array, or read another set of records into a new byte array.

CALL IT OUT

The full code for this column includes a help file for the DLL server and an example program. At a minimum, the calling program would:

• Instantiate the object:
```
Dim pEventLog As New EventLogs
```
• Set the filter:
```
pEventLog.EventFilter(FilterText) = _
   Filter Type
```
• Set the log type:
```
pEventLog.TypeEventLog = "Application"
```
• Open the log:
```
If pEventLog.OpenAnyEventLog_
   ("ServerName") = False Then
   ' Get the error information
```
• Read the log entries:
```
If pEventLog.ReadEventEntries = _
   False Then
   ' Get error information
```
• Get a count of the records:
```
plngEventCnt = _
   pEventLog.CountEventRecords
```
• Redim a type array to hold return values.
• Loop through the properties (passing the index) to fill in the type array.
• Set the object to Nothing:
```
Set pEventLog = Nothing
```

You could do several things to improve the server, but I'll leave those as an exercise for you. It would be nice to add sorting as a property of the OLE server. Another would be to optimize the sublookups for those items that have placeholders in the resource string. This could be either a very important optimization or an insignificant point, depending upon the type of log items retrieved.

Finally, if you are doing multiple lookups with different filters, it would be nice to read and store the entire event log internally in the server, even on filtered items, but return only the items that match the filter. Then, another call to the same log with another filter could just reread the stored items and apply the new filter without having to reread the log. However, you would have to at least check to see if the number of items had changed since the last time it was called, and reread the entire log if a new item had been added. You would also have to come up with some method to ensure that you had consecutive index numbers for a filtered string.

You can find the complete code for this project, along with the code for a test program to exercise the in-process server and a help file, in the *VBPJ* Forum on CompuServe. Search for the file WP0396.ZIP.

Author's Note: Because no installation program is included, you must either manually register the DLL with REGSVR32.EXE, or compile the DLL again, which will automatically register the server. In the test program, under the References item in the Tools menu, uncheck the ReadEventLogs reference. After you register the DLL, go to References again and check "Read NT EventLogs."