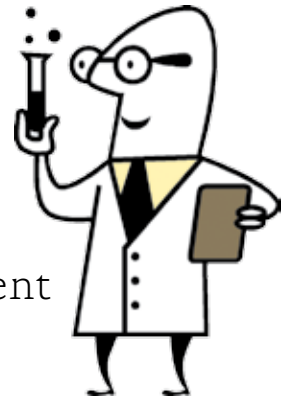


Calculating server uptime



THE MICROSOFT SCRIPTING GUYS ANSWER

How can I use the System event log to find server uptime?

U P IS UP and down is down. Seems rather obvious – except, that is, when you’re talking about server uptime. To know uptime, you need to know downtime. Nearly every network administrator is concerned about server uptime. (Except, that is, when he is worried about server downtime.) Most administrators have uptime goals and need to provide uptime reports to upper management.

What’s the big deal? It seems like you could use the Win32_OperatingSystem WMI class, which has two properties that should make such an operation pretty easy: LastBootUpTime and LocalDateTime. All you need to do, you think, is subtract the LastBootUpTime from the LocalDateTime, and then all is right with the world and you can even go catch a quick nine holes before dinner.

So you fire up Windows PowerShell to query the Win32_OperatingSystem WMI class and select the properties, as shown here:

```
PS C:\> $wmi = Get-WmiObject -Class Win32_OperatingSystem
PS C:\> $wmi.LocalDateTime - $wmi.LastBootUpTime
```

But when you run these commands, you are greeted not with the friendly uptime of your server, but by the rather mournful error message you see in **Figure 1**.

The error message is perhaps a bit misleading: “Bad numeric constant.” Huh? You know what a number is and you know what a constant is, but what does this have to do with time?

When confronted with strange error messages, it’s best to look directly at the data the script is trying to parse. Moreover, with Windows PowerShell,

it generally makes sense to see what type of data is being used.

To examine the data the script is using, you can simply print it to the screen. Here’s what you get:

```
PS C:\> $wmi.LocalDateTime
20080905184214.290000-240
```

The number seems a bit strange. What kind of a date is this? To find out, use the GetType method. The good thing about GetType is that it is nearly always available. All you need to do is call it. And here’s the source of the problem – the LocalDateTime value is being reported as a string, not as a System.DateTime value:

```
PS C:\> $wmi.LocalDateTime.GetType()

IsPublic IsSerial Name BaseType
-----
True     True     String
System.Object
```

If you need to subtract one time from another time, make sure you’re dealing with time values and not strings. This is easy to do using the ConvertToDateTime method, which Windows PowerShell adds to all WMI classes:

```
PS C:\> $wmi = Get-WmiObject -Class Win32_OperatingSystem
PS C:\> $wmi.ConvertToDateTime($wmi.LocalDateTime) -
$wmi.ConvertToDateTime($wmi.LastBootUpTime)
```

When you subtract one time value from another, you are left with an instance of a System.TimeSpan object. This means you can choose how to display your uptime information without having to perform a lot of arithmetic. You need only choose which property to display (and hopefully you count your uptime in TotalDays and not in TotalMilliseconds). The default display of the System.TimeSpan object is shown here:

```
Days           : 0
Hours          : 0
Minutes       : 40
Seconds       : 55
Milliseconds   : 914
Ticks         : 24559148010
TotalDays     : 0.0284249398263889
TotalHours    : 0.68219855833333
TotalMinutes  : 40.93191335
TotalSeconds  : 2455.914801
TotalMilliseconds : 2455914.801
```

The problem with this method is that it only tells you how long the server has been up since the last restart. It does not calculate downtime. This is where up is down – to calculate uptime, first you need to know the downtime.

So how do you figure out how long the server has been down? To do this, you need to know when the server starts and when it shuts down. You can

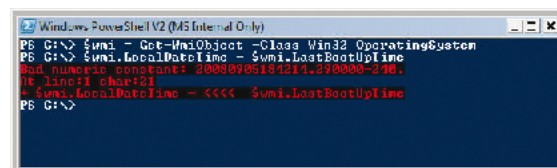


Figure 1 An error is returned when trying to subtract WMI UTC time values

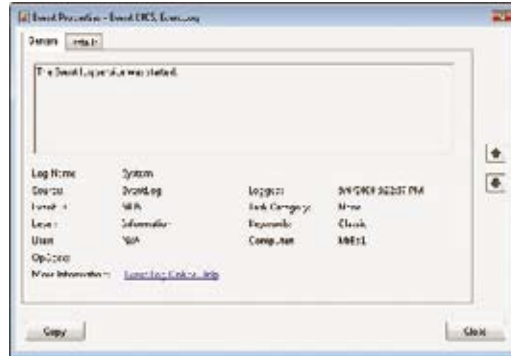


Figure 2 The event log service starts shortly after computer startup

get this information from the System event log. One of the first processes that starts on your server or workstation is the event log, and one of the last things that stop when a server is shut down is the event log. Each of these start/stop events generates an eventID – 6005 when the event log starts and 6006 when the event log stops. **Figure 2** shows an example of an event log starting.

By gathering the 6005 and 6006 events from the System Log, sorting them, and subtracting the starts from the stops, you can determine the amount of time the server was down between restarts. If you then subtract that amount from the number of minutes during the period of time in question, you can calculate the percentage of server uptime. This is the approach taken in the CalculateSystemUpTimeFromEventLog.ps1 script, which is shown in **Figure 3**.

The script begins by using the Param statement to define a couple of command-line parameters whose values you can change when you run the script from the command line. The first, \$NumberOfDays, lets you specify a different number of days to use in the uptime report. (Note that I have supplied a default value of 30 days in the script so you can run the script without having to supply a value for the parameter. Of course, you can change this if necessary.)

The second, [switch]\$debug, is a switched parameter that lets you obtain certain debugging information from the script if you include it on the command line when running the script. This information can help you feel more confident in the results you get from the script. There could be times when the 6006 event log service-stopping message is not present, perhaps as a result of a catastrophic failure of the server that rendered it unable to write to the event log, causing the script to subtract an uptime value from another uptime value and skew the results.

Figure 3 CalculateSystemUpTimeFromEventLog 3

```

#-----
# CalculateSystemUpTimeFromEventLog.ps1
# ed wilson, msft, 9/6/2008
#
# Creates a system.TimeSpan object to subtract date values
# Uses a .NET Framework class, system.collections.sortedlist to sort the events from eventlog.
#
#-----
#Requires -version 2.0
Param($NumberOfDays = 30, [switch]$debug)

if($debug) { $DebugPreference = " continue" }

[timespan]$uptime = New-TimeSpan -start 0 -end 0
$currentTime = get-Date
$startUpID = 6005
$shutDownID = 6006
$minutesInPeriod = (24*60)*$NumberOfDays
$startingDate = (Get-Date -Hour 00 -Minute 00 -Second 00).adddays(-$numberOfDays)

Write-debug "$uptime $uptime" ; start-sleep -s 1
write-debug "$currentTime $currentTime" ; start-sleep -s 1
write-debug "$startingDate $startingDate" ; start-sleep -s 1

$events = Get-EventLog -LogName system |
Where-Object { $_.eventID -eq $startUpID -OR $_.eventID -eq $shutDownID `
-and $_.TimeGenerated -ge $startingDate }

write-debug "$events $($events)" ; start-sleep -s 1

$sortedList = New-object system.collections.sortedlist

ForEach($event in $events)
{
    $sortedList.Add( $event.timeGenerated, $event.eventID )
} #end foreach event
$uptime = $currentTime - $sortedList.keys[$($sortedList.Keys.Count-1)]
Write-Debug "Current uptime $uptime"

For($item = $sortedList.Count-2 ; $item -ge 0 ; $item -- )
{
    Write-Debug "$item `t `t $($sortedList.GetByIndex($item)) `t `
    $($sortedList.Keys[$item])"
    if($sortedList.GetByIndex($item) -eq $startUpID)
    {
        $uptime += ($sortedList.Keys[$item+1] - $sortedList.Keys[$item])
        Write-Debug "adding uptime. `t uptime is now: $uptime"
    } #end if
} #end for item

"Total up time on %env:computername since $startingDate is " + "{0:n2}" -f `
$uptime.TotalMinutes + " minutes."
$UpTimeMinutes = $uptime.TotalMinutes
$percentDowntime = "{0:n2}" -f (100 - ($UpTimeMinutes/$minutesInPeriod)*100)
$percentUptime = 100 - $percentDowntime

"$percentDowntime% downtime and $percentUptime% uptime."

```

After the \$debug variable is supplied from the command line, it is present on the Variable: drive. In that case, the value of the \$debugPreference variable is set to continue, which means the script will continue to run and any value supplied to Write-Debug will be visible. Note that, by default, the value of \$debugPreference is silentlycontinue, so if you don't set the value of \$debugPreference to continue, the script will run, but any value supplied to Write-Debug will be silent (that is, it will not be visible).

When the script runs, the resulting output lists each occurrence of the 6005 and 6006 event log entries (as you can see in **Figure 4**) and shows the uptime calculation. Using this information, you can confirm the accuracy of the results.

The next step is to create an instance of the System.TimeSpan object. You could use the New-Object cmdlet to create a default timespan object you would use to perform date-difference calculations:

```
PS C:\> [timespan]$ts = New-Object system.  
timespan
```

But Windows PowerShell actually has a New-TimeSpan cmdlet for creating a timespan object, so it makes sense to use it. Using this cmdlet makes the script easier to read, and the object that is created is equivalent to the timespan object created with New-Object.

Now, you can initialise a few variables, starting with \$currentTime, which is used to hold the current time and date value. You get this information from the Get-Date cmdlet:

```
$currentTime = get-Date
```

Next, initialise the two variables that will hold the startup and shutdown eventID numbers. You don't really need to do this, but the code will be easier to read and easier to troubleshoot if you avoid embedding the two as string literal values.

The next step is to create a variable called \$minutesInPeriod to hold the result of the calculation used to figure

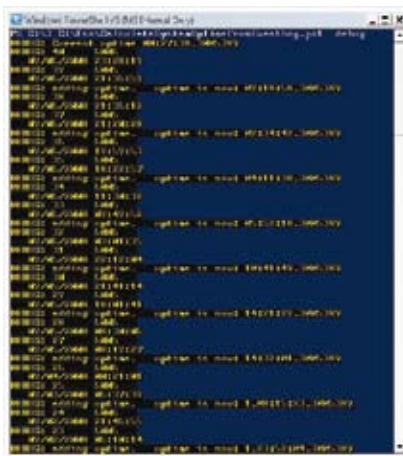


Figure 4 Debug mode displays a tracing of each time value that is added to the uptime calculation

out the number of minutes during the period of time in question:

```
$minutesInPeriod = (24*60)*$NumberOfDays
```

Finally, you need to create the \$startingDate variable, which will hold a System.DateTime object that represents the starting time of the reporting period. The date will be midnight of the beginning date for the period:

```
$startingDate = (Get-Date -Hour 00 -Minute 00  
-Second 00).adddays(-$NumberOfDays)
```

After the variables are created, you retrieve the events from the event log and store the results of the query in the \$events variable. Use the Get-EventLog cmdlet to query the event log, specifying "system" as the log name. In Windows PowerShell 2.0, you could use a -source parameter to reduce the amount of information that needs to be sorted out in the Where-Object cmdlet. But in Windows PowerShell 1.0, you don't have that choice and must therefore sort through all the unfiltered events returned by the query. So pipeline the events to the Where-Object cmdlet to filter out the appropriate event log entries. As you examine the Where-Object filter, you'll see why the Scripting Guys had you create the variables to hold the parameters.

The command reads much better than it would if you used the string lit-

erals. The get eventIDs are equal either to \$startUpID or to \$shutDownID. You should also make sure the timeGenerated property of the event log entry is greater than or equal to the \$startingDate, like so:

```
$events = Get-EventLog -LogName system |  
Where-Object { $_.eventID -eq $startUpID  
-OR $_.eventID -eq $shutDownID -and  
$_TimeGenerated -ge $startingDate }
```

Keep in mind that this command will only run locally. In Windows PowerShell 2.0, you could use the -computerName parameter to make the command work remotely.

The next step is to create a sorted list object. Why? Because when you walk through the collection of events, you are not guaranteed in which order the event log entries are reported. Even if you pipeline the objects to the Sort-Object cmdlet and store the results back into a variable, when you iterate through the objects and store the results into a hash table, you cannot be certain that the list will maintain the results of the sort procedure.

In order to sidestep these frustrating and difficult-to-debug problems, you create an instance of the System.Collections.SortedList object, using the default constructor for the object. The default constructor tells the sorted list to sort dates chronologically. Store the empty sorted list object in the \$sortedList variable:

```
$sortedList = New-object system.collections.  
sortedlist
```

After you create the sorted list object, you need to populate it. To do this, use the ForEach statement and walk through the collection of event log entries stored in the \$entries variable. As you walk through the collection, the \$event variable keeps track of your position in the collection. You can use the add method to add two properties to the System.Collections.SortedList object. The sorted list allows you to add a key and a value property (similar to a Dictionary object except that it also lets you index into the collection just like an array does). Add the time-

generated property as the key and the eventID as the value property:

```
ForEach($event in $events)
{
    $sortedList.Add( $event.timeGenerated,
    $event.eventID )
} #end foreach event
```

Next, you calculate the current uptime of the server. To do this, you use the most recent event log entry in the sorted list. Note that this will always be a 6005 instance because if the most recent entry were 6006, the server would still be down. Since the index is zero based, the most recent entry will be the count -1.

To retrieve the time-generated value, you need to look at the key property of

You can't talk about uptime without taking downtime into consideration

the sorted list. To get the index value, use the count property and subtract one from it. Then subtract the time the 6005 event was generated from the date time value stored in the \$currenttime variable you populated earlier. You can print out the results of this calculation only if the script is run in debug mode. This code is shown here:

```
$uptime = $currentTime -
$sortedList.keys[$($sortedList.Keys.Count-1)]
Write-Debug "Current uptime $uptime"
```

It is now time to walk through the sorted list object and calculate the uptime for the server. Because you are using the System.Collections.Sorted list object, you will take advantage of the fact that you can index into the list. To do this, use the for statement, beginning at the count -2 because we used count -1 earlier to figure the current amount of uptime.

We are going to count backward to get the uptime, so the condition specified in the second position of the for statement is when the item is greater or equal to 0. In the third position

Version issues

While testing the CalculateSystemUptimeFromEventLog.ps1 script on his laptop, Contributing Editor Michael Murgolo ran across a most annoying error. I gave the script to my friend Jit, and he ran into the same error. What was that error? Well, here it is:

```
PS C:\> C:\fso\
CalculateSystemUptimeFromEventLog.ps1
Cannot index into a null array.
At C:\fso\CalculateSystemUptimeFromEventLog.
ps1:36 char:43 + $uptime = $currentTime
- $sortedList.keys[$ <<<< ($sortedList.Keys.
Count-1)]
Total up time on LISBON since 09/02/2008
00:00:00 is 0.00 minutes.
100.00% downtime and 0% uptime.
```

The error, "Cannot index into a null array," is an indication that the array was not created properly. So I dug into the code that creates the array:

```
ForEach($event in $events)
{
    $sortedList.Add( $event.timeGenerated,
    $event.eventID )
} #end foreach event
```

In the end, that code was fine. What was causing the error?

Next, I decided to look at the SortedList object. To do this, I wrote a simple script that creates an instance

of the System.Collections.SortedList class and adds some information to it. At this point, I used the keys property to print out the listing of the keys. Here is that code:

```
$aryList = 1,2,3,4,5
$sl = New-Object Collections.SortedList
ForEach($i in $aryList)
{
    $sl.add($i,$i)
}

$sl.keys
```

On my computer, this code works fine. On Jit's computer, it failed. Bummer. But at least it pointed me in the right direction. The problem, as it turns out, is that there is a bug with the System.Collections.SortedList in Windows PowerShell 1.0. And I happen to be running the most recent build of the yet to be released Windows PowerShell 2.0 where that bug was fixed, and thus the code runs fine.

So where does that leave our script? As it turns out, the SortedList class has a method called GetKey, and that method works on both Windows PowerShell 1.0 and Windows PowerShell 2.0. So for the 1.0 version of the

script, we modify the code to use GetKey instead of iterating through the keys collection. In the 2.0 version of the script, we add a tag that requires version 2.0 of Windows PowerShell. If you try to run that script on a Windows PowerShell 1.0 machine, the script will simply exit and you will not get the error.

Michael also pointed out something that's not a bug but is related to a design consideration. He noted that the script will not properly detect uptime if you hibernate or sleep the computer. This is true, as we do not detect or look for these events.

In reality, however, I am not concerned with uptime on my laptop or desktop computer. I am only interested in uptime on a server, and I have yet to meet the server that sleeps or hibernates. It could happen, of course, and it might be an interesting way to conserve electricity in the data center, but it's not something I've encountered yet. Let me know if you hibernate your servers. You can reach me at Scripter@Microsoft.com.

of the for statement you use --, which will decrement the value of \$item by one. You use the Write-Debug cmdlet to print out the value of the index number if the script is run with the -debug switch. You also tab over by using the `t character and print out the timegenerated time value. This section of code is shown here:

```
For($item = $sortedList.Count-2 ; $item -ge 0 ; $item--)
{
    Write-Debug "$item `t `t $($sortedList.
    GetByIndex($item)) `t `t
    $($sortedList.Keys[$item])"
```

If the eventID value is equal to 6005, which is the startup eventID value, you calculate the amount of uptime by subtracting the start time from the previous downtime value. Store this value in the \$uptime variable. If you are in debug mode, you can use the Write-Debug cmdlet to print these values to the screen:

```
if($sortedList.GetByIndex($item) -eq $startUpID)
{
    $uptime += ($sortedList.Keys[$item+1]
    - $sortedList.Keys[$item])
    Write-Debug "adding uptime. `t uptime is
    now: $uptime"
} #end if
} #end for item
```

Last, you need to generate the report. Pick up the computer name from the system environment variable computer. You use the current time stored in the \$startingdate value, and you display the total minutes of uptime for the period. You use the format specifier {0:n2} to print the number to two digits. Next, calculate the percentage of downtime by dividing the number of uptime minutes by the number of minutes in the time period covered by the report. Use the same format specifier to print the value to two decimal places. Just for fun, you can also calculate the percent of uptime and then print out both values, like so:

```
"Total up time on $env:computername since
$startingDate is " + "{0:n2}" -f `
    $uptime.TotalMinutes + " minutes."
$UpTimeMinutes = $Uptime.TotalMinutes
$percentDowntime = "{0:n2}" -f (100 -
($UpTimeMinutes/$minutesInPeriod)*100)
$percentUpTime = 100 - $percentDowntime
"$percentDowntime% downtime and
$percentUpTime% uptime."
```

Dr Scripto's scripting perplexer

The monthly challenge that tests not only your puzzle-solving ability but also your scripting skills.

December 2008: PowerShell commands

The list below contains 21 Windows PowerShell commands. The square contains the same commands, but they are hidden. Your job is to find the commands, which may be hidden horizontally, vertically, or diagonally (forward or backward).

- | | | |
|------------------|-------------|-----------------|
| EXPORT-CSV | FORMAT-LIST | FORMAT-TABLE |
| GET-ACL | GET-ALIAS | GET-CHILDITEM |
| GET-LOCATION | INVOKE-ITEM | MEASURE-OBJECT |
| NEW-ITEMPROPERTY | OUT-HOST | OUT-NULL |
| REMOVE-PSSNAPIN | SET-ACL | SET-TRACESOURCE |
| SPLIT-PATH | START-SLEEP | STOP-SERVICE |
| SUSPEND-SERVICE | WRITE-DEBUG | WRITE-WARNING |

S	R	E	M	O	V	E	P	S	S	N	A	P	I	N	A
U	T	M	E	T	I	D	L	I	H	C	T	E	G	E	T
S	G	O	F	O	R	M	A	T	T	A	B	L	E	W	C
P	R	E	P	I	S	A	I	L	A	T	E	G	X	I	E
E	X	P	T	S	O	H	T	U	O	U	N	U	P	T	J
N	W	H	I	L	E	A	N	D	B	I	L	B	O	E	B
D	R	A	T	M	O	R	M	A	N	V	C	E	R	M	O
S	O	L	H	A	R	C	V	R	E	D	A	D	T	P	E
E	G	E	T	A	C	L	A	I	R	I	T	E	C	R	R
V	E	D	A	H	K	W	E	T	C	F	E	T	S	O	U
R	A	N	P	I	E	R	A	L	I	E	S	I	V	P	S
I	S	E	T	T	R	A	C	E	S	O	U	R	C	E	A
C	U	T	I	O	U	T	N	U	L	L	N	W	I	R	E
E	A	R	L	M	E	T	I	E	K	O	V	N	I	T	M
S	W	A	P	F	O	R	M	A	T	L	I	S	T	Y	A
R	E	C	S	N	O	P	E	L	S	T	R	A	T	S	

You can find the answer to this puzzle at technetmagazine.com/puzzle.

Let's return to the original question: When is up down? You now see that you can't talk about uptime without taking downtime into consideration. If you thought this was fun, check

out the other "Hey, Scripting Guy" columns at <http://technet.microsoft.com/magazine/cc135880> or go to the Script Center at www.microsoft.com/technet/scriptcenter. ■

ED WILSON is a senior consultant at Microsoft and a well-known scripting expert. He is a Microsoft Certified Trainer who delivers a popular Windows PowerShell workshop to Microsoft Premier customers worldwide. He has written eight books including several on Windows scripting, and has contributed to almost a dozen other books. Ed holds more than 20 industry certifications.

CRAIG LIEBENDORFER is a wordsmith and longtime Microsoft web editor. Craig still can't believe there's a job that pays him to work with words every day. One of his favourite things is irreverent humour, so he should fit right in here. He considers his magnificent daughter to be his greatest accomplishment in life.