Timer Comparison Tests – Version 1.6

Over the years, I have used various functions to measure time intervals: **Timer, GetSystemTime, GetTickCount**.

Each of these can give times to millisecond precision though I normally round to 2 d.p. (centiseconds).
This is because each function is based on the system clock which is normally updated 64 times per second – approximately every 0.0156 seconds

When I started my series of speed comparison tests, I initially used the GetSystemTime function.
However, some occasional inconsistencies led me to revert to the very simple Timer function.

Recently I was alerted to the **timeGetTime** API by Utter Access member **ADezii** with these comments taken from the Access 2000 Developers Handbook pg 1135-1136:
If you're interested in measuring elapsed times in your Access Application, you're much better off using the timeGetTime() API Function instead of the Timer() VBA Function. There are 4 major reasons for this decision:
***1.** timeGetTime() is more accurate. The Timer() Function measure time in 'seconds' since Midnight in a single-precision floating-point value, and is not terribly accurate. timeGetTime() returns the number of 'milliseconds' that have elapsed since Windows has started and is very accurate.*
***2.** timeGetTime() runs longer without 'rolling over'. Timer() rolls over every 24 hours. timeGetTime() keeps on ticking for up to 49 days before it resets the returned tick count to 0.*
***3.** Calling timeGetTime() is significantly faster than calling Timer().*
***4.** Calling timeGetTime() is no more complex than calling Timer(), once you've included the proper API declaration*

Part of this comment is no longer accurate in that the **Timer** function can measure to milliseconds.
However, as I had never used the **timeGetTime** API, I decided to compare the results obtained using each of the methods using two simple tests:

- Looping through a simple square root calculation repeatedly (20000000 times)
- Measuring the time interval after a specified time delay setup using the **Sleep API** (1.575 s)

I also added two more items to the timer comparison tests - **Stopwatch class** (again based on the system timer) and a **High Resolution Timer** (which has a resolution of 1 microsecond or less).
Many thanks to **ADezii** for this additional code

A quick summary of the 6 methods used in these tests:

- **Timer VBA** – number of seconds since midnight but to millisecond resolution
  https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/timer-function
- **GetSystemTime API –** current system date and time expressed in Coordinated Universal Time (UTC)
  https://docs.microsoft.com/en-us/windows/desktop/api/sysinfoapi/nf-sysinfoapi-getsystemtime
- **GetTickCount API –** number of milliseconds that have elapsed since the system was started (up to 49.7 days) https://docs.microsoft.com/en-us/windows/desktop/api/sysinfoapi/nf-sysinfoapi-gettickcount
- **timeGetTime API –** same calculation as **GetTickCount** but using a different API
  https://docs.microsoft.com/en-us/windows/desktop/api/timeapi/nf-timeapi-timegettime
- **Stopwatch class API -** a set of methods and properties to accurately measure elapsed time.
  https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=netframework-4.7.2
- **High Resolution Timer API –** able to measure to less than one microsecond resolution
  https://docs.microsoft.com/en-us/windows/desktop/winmsg/about-timers

Other timer methods also exist that have not been used here. For example:

- **Multimedia Timer** - used to schedule periodic timer events for multimedia applications
  https://docs.microsoft.com/en-us/windows/desktop/multimedia/multimedia-timers
- **timeGetSystemTime API** – time elapsed in milliseconds since the system was started so very similar to **timeGetTime API**
  https://docs.microsoft.com/en-us/windows/desktop/api/timeapi/nf-timeapi-timegetsystemtime

Obviously, as with any timer tests, other factors such as background windows processes, and overall CPU load will lead to some natural variation. To minimise the effects of those, I avoided running any other applications at the same time and ran each test 10 times. Furthermore, the test order was randomised each time ... just in case. The average times were calculated along with the minimum/maximum times and standard deviation for each test.
As most of the methods are based on the system clock, I expected the results to be similar in each case.
However, it seemed reasonable that certain functions would be more efficient to process

For these tests, the main requirement is certainly **NOT** to determine which gives the smallest time.
Here the aim is to achieve **consistency** so that **repeated tests** should provide a small **standard deviation**

1. Test Results

These are the average results for test A – calculation loop (old desktop PC with 32-bit Access & 4GB RAM):

Average Results

Workstation	Test	Test Type	Run Count	Loop Count	Min Time	Max Time	Std Dev	Average Time
COLIN-PC	A	GetSystemTime	20	20000000	1.781	1.797	0.004	1.788
COLIN-PC	A	GetTickCount	20	20000000	1.766	1.797	0.009	1.786
COLIN-PC	A	High Resolution Timer	20	20000000	1.763	1.783	0.005	1.770
COLIN-PC	A	Stopwatch Class	20	20000000	1.742	1.767	0.004	1.756
COLIN-PC	A	TimeGetTime	20	20000000	1.773	1.789	0.004	1.785
COLIN-PC	A	Timer	20	20000000	1.891	1.898	0.003	1.893

Record: 1 of 6 No Filter Search

As expected, the average times for each method were mostly similar except for the **Timer** method which gave noticeably larger values than all other methods.
The **Timer** method also had the least variation and **GetTickCount** the most, but the variation was small for each of the methods

For comparison, I repeated the tests on a laptop with 64-bit Access & 8GB RAM:

Average Results

Workstation	Test	Test Type	Run Count	Loop Count	Min Time	Max Time	Std Dev	Average Time
COLIN-LAPTOP	A	GetSystemTime	20	20000000	1.141	1.391	0.076	1.252
COLIN-LAPTOP	A	GetTickCount	20	20000000	1.078	1.360	0.087	1.219
COLIN-LAPTOP	A	High Resolution Timer	20	20000000	1.023	1.309	0.090	1.176
COLIN-LAPTOP	A	Stopwatch Class	20	20000000	1.080	1.469	0.091	1.174
COLIN-LAPTOP	A	TimeGetTime	20	20000000	1.125	1.328	0.061	1.196
COLIN-LAPTOP	A	Timer	20	20000000	1.047	1.281	0.056	1.131

Record: 1 of 6 No Filter Search

As you would expect, each of the times are faster. The variation was again fairly small for each method.
Once again, the **Timer** method had least variation but, on this workstation, its average time was fastest!
Perhaps surprisingly, the **High Resolution Timer** and **Stopwatch** methods had the largest variation

Finally, I used a Windows tablet with 2GB RAM. Clearly, with that specification its only just adequate for running Access and struggles with any complex processing.

Average Results

Workstation	Test	Test Type	Run Count	Loop Count	Min Time	Max Time	Std Dev	Average Time
COLIN-TABLET	A	GetSystemTime	20	20000000	5.328	6.039	0.163	5.578
COLIN-TABLET	A	GetTickCount	20	20000000	5.531	5.797	0.079	5.647
COLIN-TABLET	A	High Resolution Timer	20	20000000	5.369	5.750	0.110	5.579
COLIN-TABLET	A	Stopwatch Class	20	20000000	5.288	5.820	0.129	5.590
COLIN-TABLET	A	TimeGetTime	20	20000000	5.479	5.758	0.080	5.572
COLIN-TABLET	A	Timer	20	20000000	5.461	5.844	0.099	5.665

Record: 1 of 6 No Filter Search

The times were inevitably a LOT slower but each method gave similar average times
In this case, **TimeGetTime** and **GetTickCount** were most consistent whereas **GetSystemTime** and **Stopwatch** had the largest variation

Overall, there was little to distinguish any of the methods on any of the workstations tested

The second set of tests were done with a **specified time delay** of 1.575s. For these tests, we would expect each value to be slightly larger than the time delay to allow for processing the timer functions. There should also be less variation between the different PCs

These are the average results for test B – on the desktop PC with 4GB RAM:

Average Results

Workstation	Test	Test Type	Run Count	Delay Time	Min Time	Max Time	Std Dev	Average Time
COLIN-PC	B	GetSystemTime	20	1.575	1.578	1.602	0.008	1.584
COLIN-PC	B	GetTickCount	20	1.575	1.562	1.610	0.012	1.584
COLIN-PC	B	High Resolution Timer	20	1.575	1.575	1.590	0.005	1.579
COLIN-PC	B	Stopwatch Class	20	1.575	1.575	1.603	0.010	1.583
COLIN-PC	B	TimeGetTime	20	1.575	1.575	1.594	0.008	1.582
COLIN-PC	B	Timer	20	1.575	1.570	1.602	0.009	1.581

Record: 1 of 6 No Filter Search

The **High Resolution Timer** was the most consistent and had smaller times than the other methods suggesting it may be the fastest to process time values
The **GetTickCount** method had the largest variation
All the other functions were similar both in terms of variation and average times obtained.
There were a couple of 'impossible' values less than 1.575 seconds for both **GetTickCount** & **Timer** methods.

Here are the average results using the laptop with 8GB RAM:

Average Results

Workstation	Test	Test Type	Run Count	Delay Time	Min Time	Max Time	Std Dev	Average Time
COLIN-LAPTOP	B	GetSystemTime	20	1.575	1.578	1.594	0.006	1.592
COLIN-LAPTOP	B	GetTickCount	20	1.575	1.578	1.609	0.009	1.590
COLIN-LAPTOP	B	High Resolution Timer	20	1.575	1.576	1.590	0.004	1.583
COLIN-LAPTOP	B	Stopwatch Class	20	1.575	1.578	1.596	0.007	1.590
COLIN-LAPTOP	B	TimeGetTime	20	1.575	1.578	1.596	0.007	1.590
COLIN-LAPTOP	B	Timer	20	1.575	1.578	1.602	0.008	1.590

Record: 1 of 6 No Filter Search

Similar results once again with the **High Resolution Timer** being most consistent and with the fastest times.
Once again, the **GetTickCount** method had the largest variation
The other methods were broadly comparable both in terms of variation and average times obtained.

The results using the 2GB tablet were

Average Results

Workstation	Test	Test Type	Run Count	Delay Time	Min Time	Max Time	Std Dev	Average Time
COLIN-TABLET	B	GetSystemTime	20	1.575	1.570	1.578	0.003	1.577
COLIN-TABLET	B	GetTickCount	20	1.575	1.562	1.579	0.007	1.574
COLIN-TABLET	B	High Resolution Timer	20	1.575	1.575	1.576	0.000	1.575
COLIN-TABLET	B	Stopwatch Class	20	1.575	1.575	1.577	0.001	1.576
COLIN-TABLET	B	TimeGetTime	20	1.575	1.575	1.577	0.000	1.576
COLIN-TABLET	B	Timer	20	1.575	1.570	1.578	0.004	1.575

Record: 1 of 6 No Filter Search

Once again, **GetTickCount** produced the largest variation.
In this test **Stopwatch class, timeGetTime** and **the High Resolution Timer** were all extremely consistent.
Three of the methods had at least one 'impossible' result less than the time delay of 1.575 s

2. *Conclusions*

Overall, I would suggest that all methods are reasonably reliable with minimal variation.
Two of the simplest methods (**Timer** and **timeGetTime**) were just as consistent and at times better than the other approaches

Stopwatch class works well but requires additional code compared to the **Timer** or **TimeGetTime** methods
GetTickCount is satisfactory but perhaps not as reliable as other methods
GetSystemTime uses a combination of the **Timer** function & **SystemTime** API. As it is no better than other methods, using a combined approach such as this, is probably not the best solution.

The **high resolution timer** operates with a level of precision far greater than is needed for speed comparison tests. However, the standard deviation is far smaller than using the other methods which seems to make it more reliable in my view. The second test using a specified time delay also seems to indicate the test itself runs faster so is likely to be closer to the actual time taken as distinct from that measured.

Even so, for most of the tests, the variation between methods wasn't significant enough to make any of the approaches stand out as a clear 'winner'.

As a result, I suggest using either **Timer** or **TimeGetTime** unless you really need more precision than milliseconds

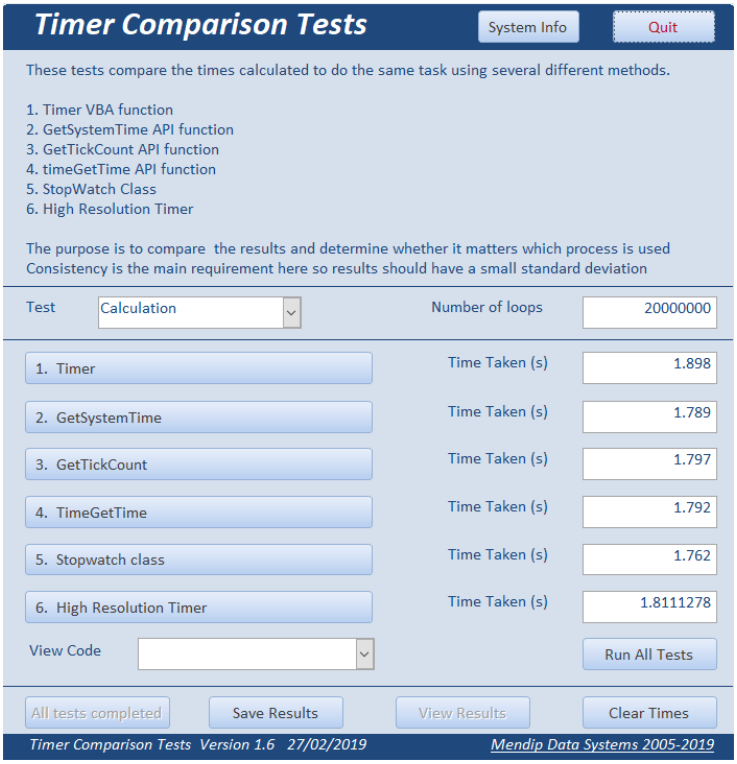
Bearing in mind that the **Timer** function is based on the time elapsed since midnight whereas **timeGetTime** runs for 49 days before resetting, **timeGetTime** should be used if the timing tests are likely to cross midnight or last longer than 24 hours.

However, for smaller time intervals on a reasonably powerful PC, I don't think there is much advantage in one method compared to the other. In any case, the code based on the **Timer** function allows for a 'round midnight' error

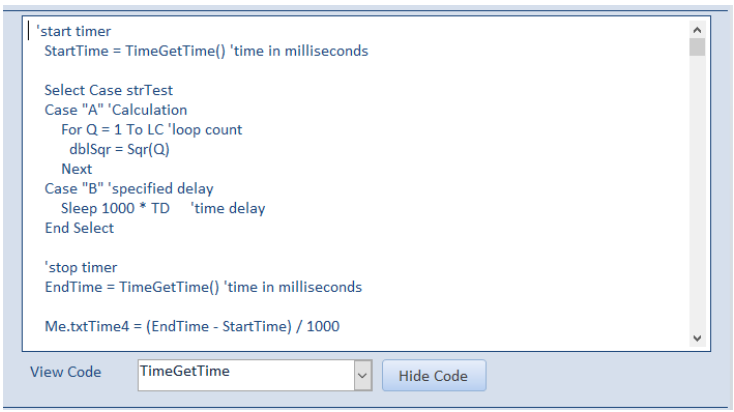
NOTE: there are other methods that I haven't yet tested successfully including the **multimedia timer**.

3. *Using the test application*

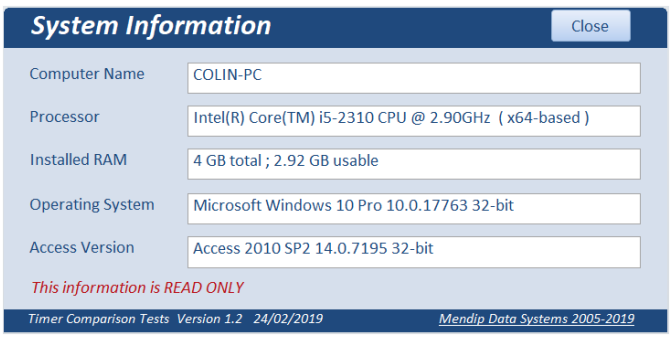
The **main form** allows you to run each test individually or to run all tests in turn.
If you choose the latter the test order will be randomised each time



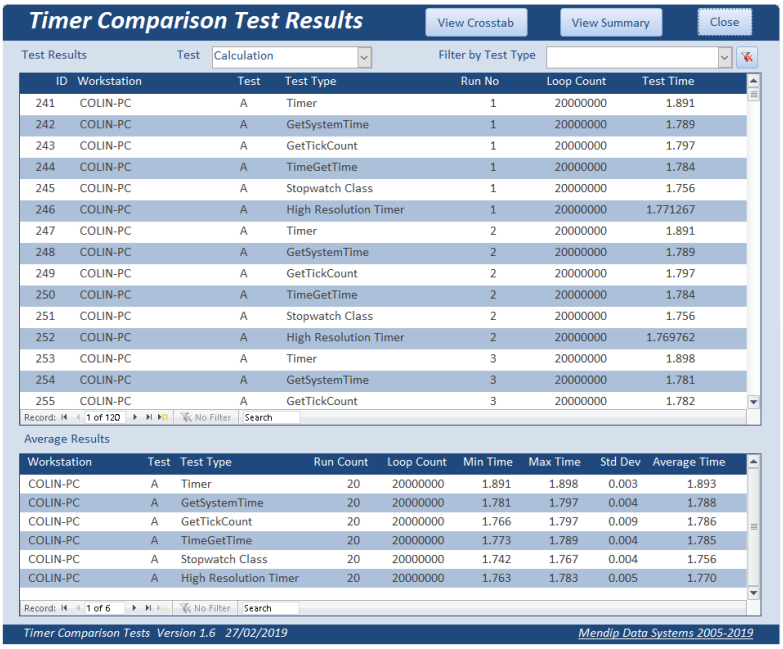
The buttons at the bottom of the form allow you to save or view the results, clear the recorded times or cancel the tests. You can also view the code used for each test by selecting from the combo box:



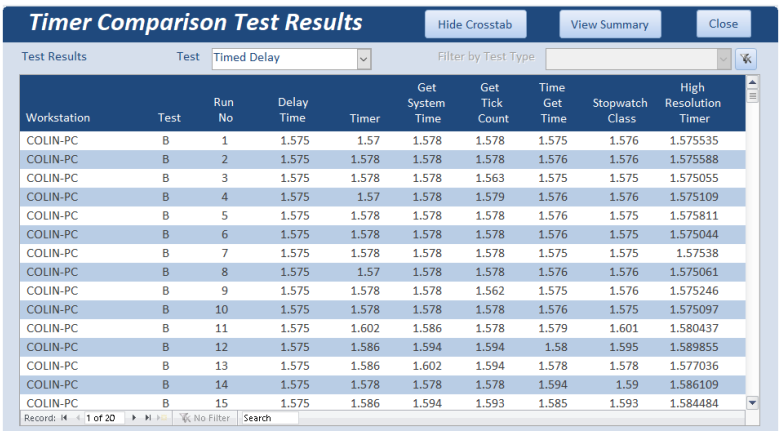
Click the **System Info** button to obtain information about your workstation. This can be useful for benchmarking. The data collection will take a few seconds with data mostly obtained using WMI.



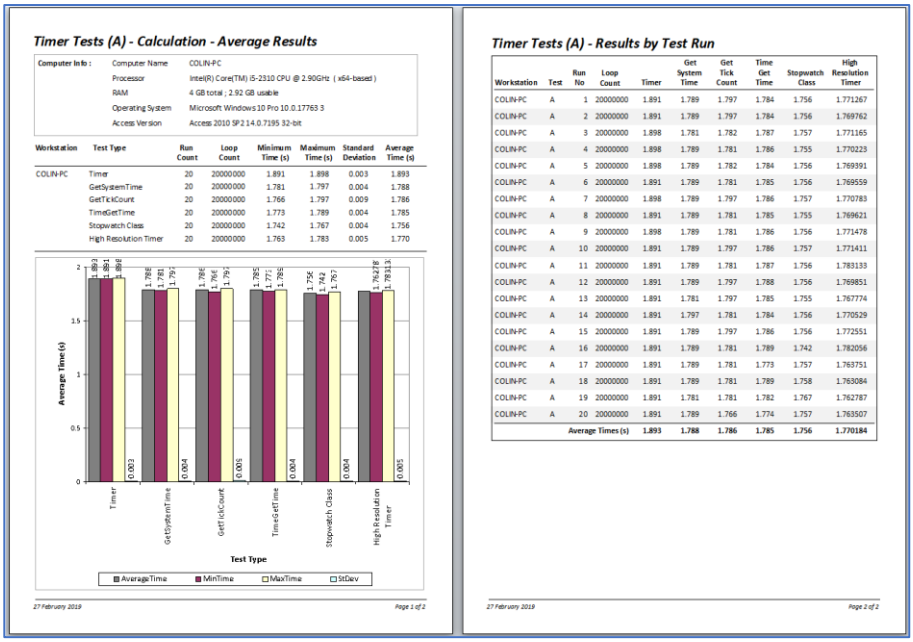
Clicking **View Results** on the main form takes you to the **Results** form



The lower part of the form shows the average results discussed earlier in this document.
The top part shows the individual results for each test run.
You can filter these for an individual test type if you wish.
Click the **View Crosstab** button to view the results for each test run in crosstab format:



Click the **View Summary** button to view summary reports with a chart. For example:



4. Clear existing data

To remove all existing data and start afresh, run 3 queries:

- qryEmptySpeedTests / qryEmptySysInfo / qryClearComputerInfo

5. Test Code

The main code used for each timing test is:

ID	Test	Timer Code	API etc
1	Timer	sngStart = Timer 'start timer Select Case strTest Case "A" 'Calculation For Q = 1 To LC 'loop count dblSqr = Sqr(Q) Next Case "B" 'specified delay Sleep 1000 * TD 'time delay End Select sngEnd = Timer 'stop timer 'check for 'round midnight issues' If sngEnd<sngStart Then sngEnd=sngEnd+86400 Me.txtTime1 = Round((sngEnd - sngStart), 3)	VBA Timer function
2	GetSystemTime	sngStart = GetCurrentSystemTime 'start timer Select Case strTest Case "A" 'Calculation For Q = 1 To LC 'loop count dblSqr = Sqr(Q) Next Case "B" 'specified delay Sleep 1000 * TD 'time delay End Select 'stop timer sngEnd = GetCurrentSystemTime 'check for 'round midnight issues' If sngEnd<sngStart Then sngEnd=sngEnd+86400 Me.txtTime2 = Round((sngEnd - sngStart), 3)	#If VBA7 Then Declare PtrSafe Sub GetSystemTime Lib "kernel32" (lpSystemTime As SYSTEMTIME) #Else Declare Sub GetSystemTime Lib "kernel32" (lpSystemTime As SYSTEMTIME) #End If ========== Function GetCurrentSystemTime() As Double 'get time to milliseconds Dim tSystem As SYSTEMTIME GetSystemTime tSystem GetCurrentSystemTime = (1000 * Int(Timer) + tSystem.wMilliseconds) / 1000 End Function
3	GetTickCount	StartTime = GetTickCount 'start timer - milliseconds Select Case strTest Case "A" 'Calculation For Q = 1 To LC 'loop count dblSqr = Sqr(Q) Next Case "B" 'specified delay Sleep 1000 * TD 'time delay End Select EndTime = GetTickCount 'stop timer -'milliseconds Me.txtTime3 = (EndTime - StartTime) / 1000	#If VBA7 Then Declare PtrSafe Function GetTickCount Lib "kernel32" () As Long #Else Declare Function GetTickCount Lib "kernel32" () As Long #End If

4	TimeGetTime	StartTime = TimeGetTime()'start timer -milliseconds Select Case strTest Case "A" 'Calculation For Q = 1 To LC 'loop count dblSqr = Sqr(Q) Next Case "B" 'specified delay Sleep 1000 * TD 'time delay End Select EndTime = TimeGetTime() 'stop timer - milliseconds Me.txtTime4 = (EndTime - StartTime) / 1000	#If VBA7 Then Public Declare PtrSafe Function TimeGetTime Lib "winmm.dll" Alias "timeGetTime" () As Long #Else Public Declare Function TimeGetTime Lib "winmm.dll" Alias "timeGetTime" () As Long #End If
5	Stopwatch class	'Create a New Instance of the Class Dim stpw As New StopWatch 'start timer If Not stpw.IsRunning Then stpw.StartTimer Select Case strTest Case "A" 'Calculation For Q = 1 To LC 'loop count dblSqr = Sqr(Q) Next Case "B" 'specified delay Sleep 1000 * TD 'time delay End Select stpw.StopTimer 'stop timer Me.txtTime5 = stpw.GetSecondsElapsed	Code in Stopwatch class module
6	High Resolution Timer	'start timer getFrequency curPerSec getTime curStartTime Select Case strTest Case "A" For Q = 1 To LC dblSqr = Sqr(Q) Next Case "B" Sleep 1000 * TD End Select 'stop timer getTime curEndTime DoCmd.Hourglass False 'calculate elapsed time Me.txtTime6 = (curEndTime - curStartTime) / curPerSec	#If VBA7 Then Declare PtrSafe Function getFrequency Lib "kernel32" Alias "QueryPerformanceFrequency" (ByRef Frequency As Currency) As Long Declare PtrSafe Function getTime Lib "kernel32" Alias "QueryPerformanceCounter" (ByRef Counter As Currency) As Long #Else Declare Function getFrequency Lib "kernel32" Alias "QueryPerformanceFrequency" (ByRef Frequency As Currency) As Long Declare Function getTime Lib "kernel32" Alias "QueryPerformanceCounter" (ByRef Counter As Currency) As Long #End If

6. *Some Useful links*

timeGetTime
https://docs.microsoft.com/en-us/windows/desktop/api/timeapi/nf-timeapi-timegettime
https://bytes.com/topic/access/insights/618175-timegettime-vs-timer

StopWatchClass
https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?redirectedfrom=MSDN&view=netframework-4.7.2

General
The following quote is based on an article at:
https://stackoverflow.com/questions/18346879/timer-accuracy-c-clock-vs-winapis-qpc-or-timegettime

Timer(), GetTickCount and timeGetTime() *are derived from a calibrated hardware clock. Resolution is not great, they are driven by the clock tick interrupt which ticks by default 64 times per second or once every 15.625 msec. You can use* ***timeBeginPeriod()*** *to drive that down to 1.0 msec. Accuracy is very good, the clock is calibrated from a NTP server, you can usually count on it not being off more than a second over a month.*

The high resolution timer *is based on the* ***Query Performance Counter API*** *and has a much higher resolution, always better than one microsecond and as little as half a nanosecond on some machines. It however has poor accuracy, the clock source is a frequency picked up from the chipset somewhere. It is not calibrated and has typical electronic tolerances. Use it only to time short intervals.*

Latency *is the most important factor when you deal with timing. You have no use for a highly accurate timing source if you can't read it fast enough. That's always an issue when you run code in user mode on a protected mode operating system which always has code that runs with higher priority than your code. Device drivers are trouble-makers, video and audio drivers in particular. Your code is also subjected to being swapped out of RAM, requiring a page-fault to get loaded back. On a heavily loaded machine, not being able to run your code for hundreds of milliseconds is not unusual. You'll need to factor this failure mode into your design*

Colin Riddington *Mendip Data Systems* *26/02/2019*