Microsecond Resolution Time Services for Windows

Arno Lentfer, June 2012

Last update: Version 2.60, February 2017

1. Abstract

Various methods for obtaining high resolution time stamping on Windows have been described. The most promising implementations have been proposed by W. Nathaniel Mills: "When microseconds matter" (2002) and Johan Nilsson: "Implement a Continuously Updating, High-Resolution Time Provider for Windows" (2004).

Suggested auxiliary initial reading: Keith Wansbrough: "Obtaining Accurate Timestamps under Windows XP" (2003), msdn: "Guidelines for Providing Multimedia Timer Support", and Chuck Walbourn: "Game Timing and Multicore Processors" (2005).

A substantial amount of time and effort has been spent on the attempt to get a proper high resolution time service implemented for Windows. However, the performance of these implementations is still not satisfactory. The complexity arises from the variety of Windows versions running on an even greater variety of hardware platforms.

Proper implementation of an accurate time service for Windows will be discussed and diagnosed within the Windows Timestamp Project. Test code will be released to prove functionality on a broader range of hardware platforms. Besides the timestamp functionality, high resolution (microsecond) timer functions are also discussed.

2. Resources

Time resources on Windows are mostly interrupt controlled entities. Therefore, they show a certain granularity. Typical interrupt periods are 10 ms to 20 ms. The interrupt period can also be set to be 1 ms or even a little below 1 ms by using API calls to *NTSetTimerResolution* or *timeBeginPeriod*. However, for several reasons they can and shall never be set to anything near the 1 µs regime. The best resolution to observe by means of Windows time services is therefore in the 1 ms regime.

The best resource for retrieving the system time is the *GetSystemTimeAsFileTime* API. It is a fast access API that is able to hold sufficiently accurate (100 ns units) values in its arguments. The alternative API is *GetSystemTime*, which is 20 times slower, has double the structure size, and does not provide a well-suited data format.

An interrupt independent system resource is used to extend the accuracy into the microsecond regime i.e., the performance counter. The performance counter API provides the asynchronous calls *QueryPerformanceCounter* and *QueryPerformanceFrequency*. A virtual counter delivers a performance counter value, which increases by a performance counter frequency. The frequency is typically a few MHz and can therefore open the microsecond regime. The counter parameters are typically backed by a physical counter, but they are not necessarily independent of the version of the operating system. A hardware platform can deliver different performance frequencies when running Windows 7 or Windows Vista, for example.

The Sleep() API and the WaitableTimer API are further timing resources in the context of this project. Their functionality and their habit also need to be looked at.

2.1. GetSystemTimeAsFileTime API

The GetSystemTimeAsFileTime API provides access to the system time in file time format. It is stated as

```
void WINAPI GetSystemTimeAsFileTime(OUT LPFILETIME lpSystemTimeAsFileTime);
```

with its argument of type

```
typedef struct _FILETIME {
    DWORD dwLowDateTime;
    DWORD dwHighDateTime;
} FILETIME;
```

A 64-bit FILETIME structure receives the system time as FILETIME in 100 ns units, which have been expired since Jan 1, 1601. After some 400 years about 1.28×10^{10} seconds or 1.28×10^{17} 100 ns slices have been accumulated. The 64-bit value can hold almost 2×10^{19} 100 ns time slices. The remaining time before this scheme wraps would be about 58,000 years from now. The call to GetSystemTimeAsFileTime typically requires 10 ns to 15 ns.

In order to investigate the real accuracy of the system time provided by this API, the granularity that comes along with the time values needs to be discussed. In other words: How often is the system time updated? A first estimate is provided by the hidden API call:

```
NTSTATUS NtQueryTimerResolution(
  OUT PULONGMinimumResolution,
  OUT PULONGMaximumResolution,
  OUT PULONGActualResolution);
```

NtQueryTimerResolution is exported by the native Windows NT library NTDLL.DLL. The *ActualResolution* reported by this call represents the update period of the system time in 100 ns units, which obviously do not necessarily match the interrupt period. The value depends on the hardware platform. Common hardware platforms report 156,250 or 100,144 for *ActualResolution*; older platforms may report even larger numbers. This is one of the heartbeats controlling the system. The *MinimumResolution* and the *ActualResolution* are relevant for the multimedia timer configuration. Two common hardware platform configurations are discussed here to highlight the details to be dealt with:

Platform configuration A

- MinimumResolution: 156,250
- MaximumResolution: 10,000
- ActualResolution: 156,250

Platform configuration B

- MinimumResolution: 100,144
- MaximumResolution: 10,032
- ActualResolution: 100,144

Platform A simply has 64 timer interrupts per second (64 x 156,250 x 100 ns = 1 s) but when looking at platform B the difficulties become more obvious: 99.856 interrupts per second? Answer: The full second interrupt is not available on all platforms.

However, the system time may be updated at these interrupt events. An API call to

```
BOOL WINAPI GetSystemTimeAdjustment(
  OUT PDWORD lpTimeAdjustment,
  OUT PDWORD lpTimeIncrement,
  OUT PBOOL lpTimeAdjustmentDisabled);
```

will disclose the time adjustment and time increment values. The actual purpose of this call is to query the status of the system time correction, which is active when *TimeAdjustmentDisabled* is FALSE. When *TimeAdjustmentDisabled* is TRUE, no adjustment takes place and *TimeAdjustemt* and *TimeIncrement* are equal and do report exactly what was read as *ActualResolution* before. For a platform A type system the call will report that the system time has incrementally increased by 156,250 100 ns units every 156,250 100 ns units. Within this description, this is considered the *granularity* of the system time.

Knowing the system time granularity raises doubts about its accuracy. Certainly, the *TimeIncrement* will be applied, thus changes of the system time will always be one *TimeIncrement*, but does the interrupt period or any multiple of it always match the time increment?

Even when the standard setting of *ActualResolution* corresponds to the *MinimumResolution*, the *ActualResolution* may have a setting different from *MinimumResolution* (see table below). In fact, it may be configured to values in the range from *MinimumResolution* to *MaximumResolution*. The *ActualResolution* determines the interrupt period of the system. That is the period after which the timer generates an interrupt to let the system react. The *ActualResolution* can be set by using the API call

```
NTSTATUS NtSetTimerResolution(
  IN ULONG RequestedResolution,
  IN BOOLEAN Set,
  OUT PULONG ActualResolution);
```

or via the multimedia timer interface

```
MMRESULT timeBeginPeriod(UINT uPeriod);
```

with the value of *uPeriod* derived from the range allowed by

```
MMRESULT timeGetDevCaps(LPTIMECAPS ptc, UINT cbtc );
```

which fills the structure

```
typedef struct {
    UINT wPeriodMin;
    UINT wPeriodMax;
} TIMECAPS;
```

Typical values are 1 ms for *wPeriodMin* and 1,000,000 ms for *wPeriodMax*. The 1,000 s period for *wPeriodMax* is somewhat meaningless within the context of this description. However, the possibility of setting the timer resolution to 1 ms requires a more detailed investigation. When the multimedia timer interface is used to set the multimedia timer to

wPeriodMin, the *ActualResolution* received by a call to *NtQueryTimerResolution* will show a new value. For the two platform configurations discussed, the examples are as follows:

Platform configuration	A	B
MinimumResolution	156,250	100,144
MaximumResolution	10,000	10,032
ActualResolution	156,250	100,144

ActualResolution varies according to the varying multimedia timer periods *uPeriod* applied by the timeBeginPeriod() API:

Platform configuration	A	B	*uPeriod*
ActualResolution	9,766	10,032	1 ms
ActualResolution	19,532	20,064	2 ms
ActualResolution	19,532	30,096	3 ms
ActualResolution	39,063	39,952	4 ms
ActualResolution	39,063	49,984	5 ms
ActualResolution	39,063	60,016	6 ms
ActualResolution	39,063	70,048	7 ms
ActualResolution	156,250	80,080	8 ms
ActualResolution	156,250	89,936	9 ms
ActualResolution	156,250	100,144	10 ms
ActualResolution	156,250	100,144	11 ms
ActualResolution	156,250	100,144	12 ms
…	…	…	…
ActualResolution	156,250	100,144	100 ms

This list shows the supported interrupt periods for platforms of type A and B in 100 ns units. Platform A only supports four different interrupt heartbeat frequencies, while platform B has a better approximation to the desired period. The specific numbers are relevant for the procedures described here and thus need a detailed interpretation.

Note: *TimeIncrement* provided by *GetSystemTimeAdjustment* and *ActualResolution* provided by *NtQueryTimerResolution* are not necessarily identical. Platform A operates with an ACPI PM timer and platform B operates with a PIT timer. More modern platforms do not show "unsupported" values of *uPeriod*.

2.1.1. *ActualResolution* on Platform Type A

The timer intervals are given with 100 ns accuracy in the last digit. Since the true *ActualResolution* cannot be expressed correctly, rather than reporting the true *ActualResolution* of 0.9765625 ms, the call to *NtQueryTimerResolution* reports the rounded value of 0.9766 ms. The other values are also rounded (shall be 1.953125 ms and 3.90625 ms respectively).

A quick test using the Sleep(dwMilliseconds) API confirms this assumption:

Sleep(1) = 1.9531 ms = 2 x 0.9765625 ms
Sleep(2) = 2.9295 ms = 3 x 0.9765625 ms
Sleep(3) = 3.9062 ms = 4 x 0.9765625 ms

The Sleep() will only return when n x *ActualResolution* exceeds the desired duration. The required accuracy for the interval specification would have to extend to 0.5 ns, in other words show the 100 ps digit. The number would be 156,250,000 for the *MinimumResolution* and 9,765,625 for the *MaximumResolution* (in 100 ps or 10^{-10} s units).

Note: Sleep(1) measurements (10000, with 100 ahead) result in a mean delay of 1953.163824 µs. This is 2.0000397 times the interrupt time slice (should have been 1953.125 µs, so the measurement was off by 0.04 µs).

2.1.2. *ActualResolution* on Platform Type B

An interrupt timer period of 1.0032 ms will accumulate 10.032 ms after 10 interrupts and change the system time by 10.0144 ms. A time change of 10.0144 ms after 10.032 ms means that the time is behind by 176 µs. At the 57th of such periods, the deviation has accumulated to 1.0032 ms, which is exactly one timer interrupt period and the time will be updated after just 9 interrupts (9.0288 ms). This way the time is updated by 10.0144 ms 56 times after 10.032 ms and one time after 9.0288 ms, which is a total elapsed time of 570.8208 ms with an adjustment of 57*10.0144 ms = 570.8208 ms. This corresponds to a total number of interrupts of 569 (57*100,144 = 569*10,032). As a result, the time will lose 176 µs for each of the 56 consecutive system time updates and then gain 9.856 ms in the 57th interrupt interval.

2.1.3. Changes of System File Time

The system time changes according to the described mechanisms after a certain period of time. Additional time changes do happen if time corrections are caused by periodic time changes, which are continuously applied to the system time over a longer period of time to adjust to an external time reference. The occurrence and the parameters of this adjustment can be gathered by a call to *GetSystemTimeAdjustment*. Sudden time changes, for example, introduced by using the clock GUI or *SetSystemTime(…)*, are not announced or predictable; they happen spontaneously.

Changes of the system time will have no influence on the expiration of Sleep periods or waitable timer periods. The actual change will be taken over by the routines here. Nevertheless, system time changes are discontinuities in time, whether they are sudden or spread over a longer period of time. What is an accurate time stamp supposed to deliver when the system inserts several hundred seconds at an interval of 1.0000032 s? The system will assume that the seconds are that long (elongated) for the time being. This can be accomplished by the temporary adaptation of the performance counter frequency to the applied granular time correction.

2.1.4. Windows 7/8/8.1/10 and Server 2008 R2/2012/2012 R2/2016

Time services on windows have undergone changes with any new version of Windows. Considerable changes are to be reported beyond VISTA and Server 2008. The synchronous progress in hardware and software development requires the software to stay compatible with a whole variety of hardware platforms. On the other hand new hardware enables the software to conquer better performance. Today's hardware provides the High Precision Event Timer (HPET) and an invariant Time Stamp Counter (TSC). The variety of timers is described in "Guidelines For Providing Multimedia Timer Support". The "IA-PC HPET Specification" is now more than 10 years old and some of the goals have not yet been reached (e.g. aperiodic interrupts). While QueryPerformanceCounter benefited using the HPET/TSC when compared to ACPI PM timer, these days the HPET is outdated by the invariant TSC for many applications. However, the typical HPET signature (*TimeIncrement* of the function GetSystemTimeAdjustment() and *MinimumResolution* of the function NtQueryTimerResolution() are 156001) disappeared with Windows 8.1. Windows 8.1 goes back to the roots; it goes back to 156250. The TSC frequency is calibrated against HPET periods to finally get proper timekeeping.

An existing invariant TSC influences the behavior of GetSystemTimeAsFileTime() noticeable. The influence to the functions QueryPerformanceCounter() and QueryPerformanceFrequency() is described in sections 2.4.3. and 2.4.4. Windows 8 introduces the function GetSystemTimePreciseAsFileTime() *"with the highest possible level of precision (<1us)."* This seems the counterpart to the linux gettimeofday() function.

2.1.4.1. Resolution, Granularity, and Accuracy of System Time

Since Windows 7, the operating system runs tests on the underlying hardware to see which hardware is best used for timekeeping. When the processors Time Stamp Counter (TSC) is suitable, the operating system uses the TSC for timekeeping. If the TSC cannot be used for timekeeping the operating system reverts to the High Precision Event Timer (HPET). If that does not exist it reverts to the ACPI PM timer. For performance reasons it shall be noted that HPET and ACPI PM timer cause IPC overhead, while the use of the TSC does not. The evolution of TSC shows a variety of capabilities:

- Constant: The TSC does not change with CPU frequency changes, however it does change on C state transitions.
- Invariant: The TSC increments at a constant rate in all ACPI P-, C- and T-states.
- Nonstop: The TSC has the properties of both Constant and Invariant TSC.

Details of the TSC capabilities are described in "Intel® 64 and IA-32 Architectures Software Developer's Manual". Chapter 16.12.1 of this documentation releases the key for using the TSC for wall clock timer services:

"The time stamp counter in newer processors may support an enhancement, referred to as invariant TSC. Processor's support for invariant TSC is indicated by CPUID.80000007H:EDX[8].

The invariant TSC will run at a constant rate in all ACPI P-, C--, and T-states. This is the architectural behavior moving forward. On processors with invariant TSC support, the OS

may use the TSC for wall clock timer services (instead of ACPI or HPET timers). TSC reads are much more efficient and do not incur the overhead associated with a ring transition or access to a platform resource."

An invariant TSC enables QueryPerformanceCounter(), QueryPerformanceFrequency(), and GetSystemTimeAsFileTime() to be served by the same hardware. Deviations, as described in 2.4.3 are non existing when the performance counter values and the wall clock are supported by the same counter (TSC).

More information can be obtained in "Intel 64® and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2".

Polling system time changes by repeated call of GetSystemTimeAsFileTime() discloses a new behavior on Windows 8: Examples given in 2.1.1. and 2.1.2. are typical timekeeping schemes for systems running with a ACPI PM timer a PIT timer respectively. System time changes occurred at some regular base. This is not the case on Windows 8; a whole bunch of varying file time increments is observed when polling on file time transition. A truly periodic cycle can only be approximated by a "mean increment". However, this mean increment matches the result given by *ActualResolution*. Despite these little hiccups, resolution, granularity, and accuracy of GetSystemTimeAsFileTime() are comparable to earlier Windows versions.

2.1.4.2. Desktop Applications: GetSystemTimePreciseAsFileTime()

This new Windows 8 API is restricted to desktop applications.

```
VOID WINAPI GetSystemTimePreciseAsFileTime(
            _Out_ LPFILETIME lpSystemTimeAsFileTime);
```

GetSystemTimePreciseAsFileTime() uses the performance counter to achieve the microsecond precision. Depending on the hardware platform and Windows version, a call to QueryPerformanceCounter may be expensive or not (HPET, ACPI PM timer, or TSC, see "MSDN: Acquiring high-resolution time stamps."). Consecutive calls may return the same result. The call time is less than the smallest increment of the system time. The granularity is in the sub-microsecond regime. The function may be used for time measurements but some care has to be taken: Time differences may be ZERO.

The function shall also be used with care when a system time adjustment is active. Current Windows versions treat the performance counter frequency as a constant. The high resolution of GetSystemTimePreciseAsFileTime() is derived from the performance counter value at the time of the call and the performance counter frequency. However, the performance counter frequency should be corrected during system time adjustments to adapt to the modified progress in time. Current Windows versions don't do this. The obtained microsecond part may be severely affected when system time adjustments are active. Seconds may consist of more or less than 1.000.000 microseconds. Microsoft may or not fix this in one of the next updates/versions.

GetSystemTimePreciseAsFileTime() works on all platforms.

As of Windows 10 (Build 10240), the inaccuracy of GetSystemTimePreciseAsFileTime() during system time adjustments persists.

2.1.4.3. Timer Periods with Invariant TSC

Using the processors invariant time stamp counter for timekeeping requires a calibration of timer periods. The TSC is used as a measure for the progress in time. However, the periodic update of the system time is still done by timer hardware because the TSC does not produce periodic events. Such periodic event may be generated by the HPET. Querying the timer resolutions on such a platform as described in section 2.1 will produce a pattern like this:

ActualResolution	*uPeriod*
1,0007	1 ms
2,0001	2 ms
3,0008	3 ms
4,0002	4 ms
5,0009	5 ms
6,0003	6 ms
7,0010	7 ms
8,0005	8 ms
9,0012	9 ms
10,0006	10 ms
11,0000	11 ms
12,0007	12 ms
13,0001	13 ms
14,0008	14 ms
15,0003	15 ms
15,6351	16 ms

The values of *ActualResolution* are accompanied by small offsets which may vary from boot to boot but they stay constant during operation. This clearly indicates that the timer periods are calibrated during boot time. Consequently, system time updates are done a those periods with a mean progress of *ActualResolution.*

The calibration of the performance counter frequency during boot is described in section 2.4. The tiny deviations seen in the list above are a result of the calibration accuracy. Again: The TSC frequency is calibrated against HPET timer periods. This is to be done in a reasonable short time to not extend the boot time too much. The remaining deviations are small but noticeable (e.g. 1.2 µs in 9,0012 for a 9 ms period corresponds to 840ppm!).

2.2. The Sleep API

The Sleep function suspends the execution of the current thread for a specified interval.

```
VOID Sleep(DWORD dwMilliseconds);
```

This would indeed be a very useful function if it were doing what it is supposed to do. Unfortunately, a detailed view discloses some artifacts, some of which are helpful, and others that are not. The Sleep() function is backed up by the system's interrupt services. As described in section 2.1., the interrupt period can be configured to some extent. This

has a direct impact on Sleep(). The call to Sleep() passes the parameter *dwMilliseconds* to the system and expects the function to return after *dwMilliseconds.* In practice, the Sleep() only returns when two conditions are met: Firstly, the requested delay must be expired and secondly an interrupt has occurred (the test to see if the requested delay has expired is only done with an interrupt). A simple Sleep(1) call may therefore have a number of different results. The results also depend on the time at which the call was made with respect to the interrupt period phase.

Say the *ActualResolution* is set to 156,250, the interrupt heartbeat of the system will run at 15.625 ms periods or 64 Hz and a call to Sleep is made with a desired delay of 1 ms. Two scenarios are to be looked at:

- The call was made < 1 ms (ΔT) ahead of the next interrupt. The next interrupt will not confirm that the desired period of time has expired. Only the following interrupt will cause the call to return. The resulting sleep delay will be ΔT + 15.625 ms.
- The call was made >= 1 ms (ΔT) ahead of the next interrupt. The next interrupt will force the call to return. The resulting sleep delay will be ΔT.

The observed delay heavily depends on the time at which the call was made. This matters particularly when the desired delay is shorter than the *ActualResolution*. However, when the *ActualResolution* is set to *MaximumResolution*, the system runs at its maximum interrupt frequency and the deviations are in the order of one interrupt period.

This behavior can be used to synchronize code with the interrupt period in an easy way by simply calling two or more consecutive sleeps. Regardless of what ΔT is, the first will end at the time of an interrupt. Consequently, the following sleep call will start at the interrupt time (or at least so close to it that the system will assume that it happened at the same time). As a result, a ΔT = 0 applies and the sleep will return when N x *ActualResolution* becomes larger than the desired period. Right after the return of a sleep the system has just processed an interrupt. Conditional latency may be on board due to a priority and/or task/process switching delay or due to interrupt handler CPU capture reasons. Typical latencies of a few µs can be observed with very little implementation effort.

A special case is the call Sleep(0). It looks meaningless, but it is a very powerful tool since it relinquishes the reminder of the thread's time slice. That means that other threads of equal priority level will take over when ready to run. When a number of threads are running at the same priority level and all of them are very responsive, all of them will make frequent calls to Sleep(0) whenever they can afford it. As a result, a task switch can be forced to happen in just a few µs.

2.3. The WaitableTimer API

Another important mechanism for performing timed operations is provided by the waitable timer interface:

```
HANDLE WINAPI CreateWaitableTimer(
  IN LPSECURITY_ATTRIBUTES lpTimerAttributes,
  IN BOOL bManualReset,
  IN LPCTSTR lpTimerName);
```

The returned handle is used to setup a timer function:

```
BOOL WINAPI SetWaitableTimer(
  IN HANDLE hTimer,
  IN const LARGE_INTEGER* pDueTime,
  IN LONG lPeriod,
  IN PTIMERAPCROUTINE pfnCompletionRoutine,
  IN LPVOID lpArgToCompletionRoutine,
  IN BOOL fResume);
```

This tool can be used in a variety of ways. Below are just a few things that need to be mentioned within the scope of this description:

- The LARGE_INTEGER structure *DueTime* specifies when the timer is to be set signaled for the first time. This is basically a file time, but formatted as LARGE_INTEGER to allow signed values. The sign is used by the system to allow input of absolute times (positive) or relative times (negative). The system time only changes in steps of *TimeIncrement* and the *DueTime* is only compared when an interrupt occurs. This effectively means that the timer can only reach a signaled state for the first time when a system time transition occurs.
- The *Period* parameter specifies whether the timer will be a single shot timer or a periodic timer. With *Period* = 0, the timer will only get signaled once when the system time has reached the *DueTime*. With *Period* > 0, the period specifies a timer period in ms, resulting in a timer heartbeat of *Period* ms. Similar to the Sleep, the periodic waitable timer will be set signaled when *Period* expires. But this is only tested when an interrupt occurs. Real cyclic periods can only be observed if *Period* is a multiple of *ActualResolution* (the interrupt period) or when the overshoot remains constant. An example for the first case can be easily described for a platform configuration of type A. A timer *Period* of 1,000 ms hosts exactly 1,024 interrupt intervals of 0.9765625 ms. Such a periodic timer will be truly cyclic.
  If, on such a platform, *Period* is setup to 995 ms, the timer will expire after 1,019 interrupt periods, resulting in a delay of 995.1171875 ms. However, the waitable timer uses the system file time and those overshoots will show deviations when a *Period* hits a system time transition. In other words: A non-truly cyclic timer setup will suffer from a beat frequency with the system file time increment frequency. The detailed discussion of this behavior falls outside the scope of this description. Evidently, a truly cyclic timer interval can also be set up when the beat frequency stays in phase with the system file time update. A typical scenario can be described for a platform configuration B type system:

  Assuming the *ActualResolution* is set to *MaximumResolution* (10,032 100 ns units); the *TimeIncrement* (100,144 100 ns units) is not a multiple of the *ActualResolution*. In order to set up a truly cyclic timer, the least common multiple of 100,144 and 10,032 has to be found. The value of 5,708,208 suits this need here; it hosts 57 periods at *MinimumResolution* or 569 periods at *ActualResolution*. The first truly cyclic timer period is therefore 570.8208 ms. It will be setup by a *Period* value of 570 and will expire after 569 interrupt periods. At the time of expiration the system will have done 57 system time updates. More truly cyclic timer setups can be created at any multiple of 5,708,208 for this type of platform. (Example: *Period*= 1,141, the timer will expire after 1,138 interrupt periods or 1141.6416 ms and the system time will have progressed by 114 x 10.0144 ms which is 1141.6416 ms too.)

- An optional asynchronous *CompletionRoutine* (APC) with an optional pointer to arguments *ArgToCompletionRoutine* can be passed to the timer. However, the calling thread needs to be in the *alertable* state to allow execution of the APC. The only advantage of the scheme with a completion routine is that this routine is automatically supplied with the systems FILETIME at which the timer was signaled. Calling the APC unfortunately results in a considerable extension of the observed cyclic interval. When the system file time is needed, it can be queried as described in 2.1). The extra time required to do this is a tiny fraction (1/2,000) of the time added to the timer period by calling the APC.

The expired (signaled) timer can be handled by means of an asynchronous procedure call (APC) or by means of a call to *WaitForSingleObject*, for example. According to the last point above, the former is useless when high accuracy is required. The latter suits the needs of the mechanisms described here much better. The API needs the handle of the object to wait for and allows specifying a timeout *dwMilliseconds*, which can be optionally set to INIFINTE.

```
DWORD WINAPI WaitForSingleObject(
  IN HANDLE hHandle,
  IN DWORD dwMilliseconds);
```

Waitable timers synchronize to the rhythm of the system interrupt period (*ActualResolution*). This has to be kept in mind because it has severe implications to the system's overall performance. All of the tasks waiting for a Sleep() or an object to reach signaled state will continue after the interrupt has occurred. The system's load tends to reach peaks at interrupts.

2.4. The QueryPerformanceCounter and QueryPerformanceFrequency API

This API is backed by a virtual counter running at a "fixed" frequency started at boot time. The following two basic calls are used to explore the microsecond regime: QueryPerformanceCounter() and QueryPerformanceFrequency(). The counter values are derived from some hardware counter, which is platform dependent. However, the Windows version also influences the results by handling the counter in a version specific manner. Windows 7, in particular, has introduced a new way of supplying performance counter values.

2.4.1. QueryPerformanceCounter

The call to

```
BOOL QueryPerformanceCounter(OUT LARGE_INTEGER *lpPerformanceCount);
```

will update the content of the LARGE_INTEGER structure *PerformanceCount* with a count value. The count value is initialized to zero at boot time.

2.4.2. QueryPerformanceFrequency

The call to

```
BOOL QueryPerformanceFrequency(OUT LARGE_INTEGER *lpFrequency);
```

will update the content of the LARGE_INTEGER structure *PerformanceFrequency* with a frequency value. The frequency is treated by the system as a constant. From Windows

7/Server 2008 R2 onwards the result of QueryPerformanceCounter() may be calibrated at boot time and may therefore return varying results. This depends on the underlying hardware (see 2.1.4.1.), But QueryPerformanceCounter() never reports any changes of the frequency during operation; its result remains constant. The following chapter describes deviations on systems on which the underlying hardware neither provides an invariant TSC nor provides a HPET for time services.

2.4.3. Performance of the Performance Counter

The range in time that can be held by the LARGE_INTEGER structure *PerformanceCount* depends on the update rate or the *Frequency* at which the count will incrementally increase. Depending on the hardware platform, the counter may be an Intel 8245 at 1,193,000 Hz or an ACPI Power Management Timer chip with an update frequency of 3,579,545 Hz or even another source. A number of platforms do not have these timers at all; they mimic the timer by providing the CPU clock. As a result of the latter, the frequency can get into the GHz range. *PerformanceCount.QuadPart* (signed) will change sign after 2^{63} increments. At a frequency of say 1GHz ($10^9\ s^{-1}$) such a system can run for about 290 years without reaching the sign bit. Even for multi-GHz platforms, there does not seem to be a serious limit.
However, apart from the system's treatment, the frequency cannot be considered as constant. Firstly, the frequency generating hardware will deviate from the specified value by an offset and secondly the frequency may vary (i.e., due to thermal drift). The impact of these deviations is not negligible. Oscillators do have tolerances in the range of a few ppm and would consequently introduce errors of a few µs/s in the measured time period. Within this description, the performance counter will be used to predict time intervals over a few seconds at accuracies better than 1µs. If an accuracy of 0.1 µs shall be reached after 10s, the frequency needs to be known to 0.01 ppm, which corresponds to 0.035 Hz at a nominal frequency of 3,579,545 Hz. Obviously, that value is not provided by the system and needs to be calibrated. A first estimate of the true frequency can be gathered by querying two counter values at a certain (known) time apart from each other. The code snippet uses the API call

```
DWORD timeGetTime(VOID);
```

and could look like this:

```
DWORD ms_begin,ms_end;
LARGE_INTEGER count_begin, count_end;
double ticks_per_second;
ms_begin = timeGetTime();
QueryPerformanceCounter(&count_begin);
Sleep(1000);
ms_end = timeGetTime();
QueryPerformanceCounter(&count_end);
ticks_per_second = (double)(count_end-count_begin)/(ms_end-ms_begin);
```

However, due to artifacts described in 2.2 timeGetTime() is accompanied by an inaccuracy of up to 2 ms; thus a Sleep(1,000) would give an accuracy for *ticks_per_second* of 0.002 (2,000ppm) at most. An accuracy of 2 ppm would be achievable when the Sleep extends to 1,000,000 ms or 1,000 s. In order to obtain 0.01 ppm the Sleep would have to cover more than 55 hours. This is obviously a hopeless approach. It also averages temporary changes of the frequency and it will not forgive frequency changes due to thermal drifts.

The thermal drift of the performance counter frequency can be severe:

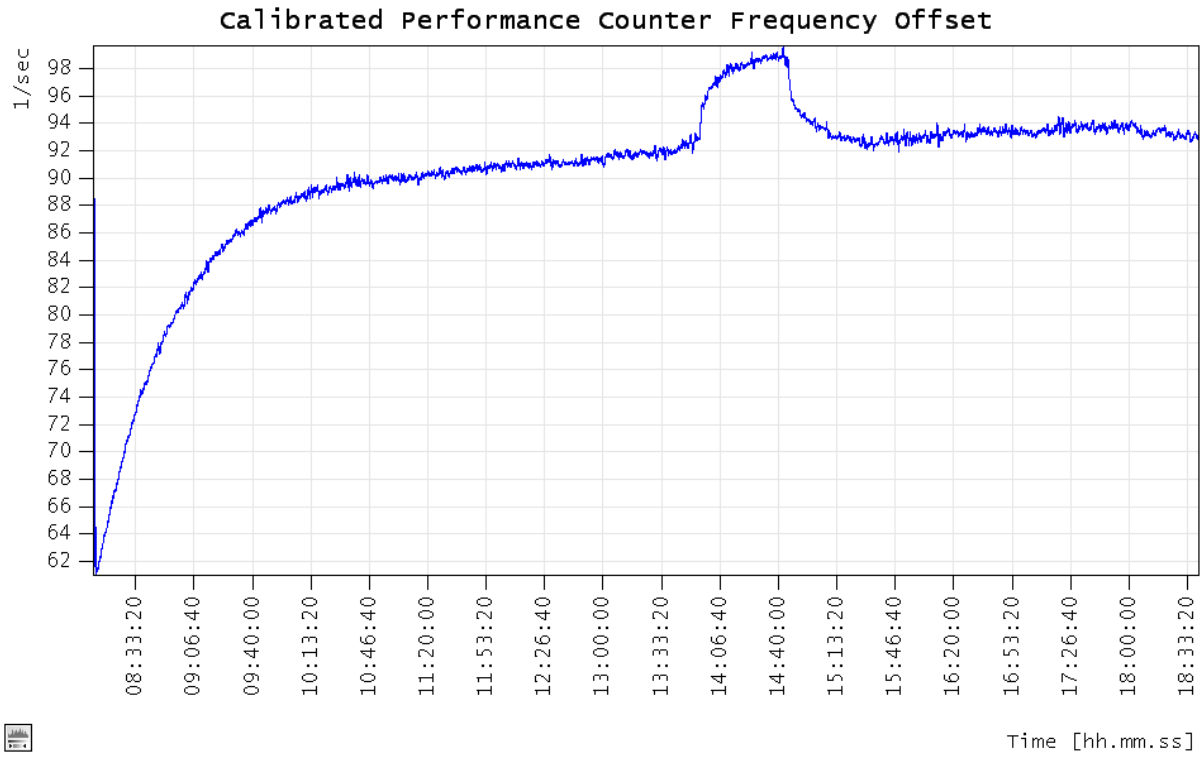

Fig. 2.4.3.1: Calibrated Performance Counter Offset on ACPI PM timer hardware.

This graph shows an older system with heavy thermal drift. At boot time (~8:00), the measured performance counter frequency is off by about 60 Hz. The system reports the performance counter frequency as 3,579,545 Hz. In fact, it is already at 3,579,605 Hz when it is "cold". After many hours of doing nothing, the system seems to reach a thermal equilibration. At ~14:00 (six hours after boot), the system was heavily loaded for about 45 minutes and consequently warmed up. The load has increased the main board temperature by 5 deg. (centigrade scale) only, but the influence to the measured performance counter frequency is quite considerable. It rose to an offset of almost 100 Hz or a true performance counter frequency of 3,579,645 Hz. A 100 Hz offset at a base frequency of 3,579,605 Hz is a deviation of about 28 ppm or an error in time of 28µs/s.

The calibration procedure used for the time stamp mechanism described here uses a repeated averaging period evaluation and reaches an accuracy of better than 0.05 ppm after about 100 s. Thermal drifts can be captured reasonably well and can be applied without much delay. (Note: The declaration of *ticks_per_second* as a 64-bit float in the code snippet above enables the *ticks_per_second* to hold a number with an accuracy of 15 digits. A value of 3,579,545.12 Hz shows the 0.01 ppm accuracy in the last digit.)

The use of QueryPerformanceCounter on multi-processor platforms implies that the call is made on the same processor all the time. The *SetThreadAffinityMask* API and its associated calls are used to ensure this. This rule only applies to systems using non invariant TSC hardware. The system analyzed in this chapter operates time services based on ACPI PM hardware.

2.4.4. Is the CPU Time Stamp Counter an Alternative?

The RDTSC specifies a call to query the time stamp counter of the CPU. The advent of multi processor platforms or muti-core processors highly recommends not using RDTSC calls. Newer processors also support adaptive CPU frequency adjustments. This is just another reason to not use RDTSC calls for the purpose discussed here. Microsoft strongly discourages using the TSC for high-resolution timing ("Game Timing and Multicore Processors"). However, the introduction of invariant Time Stamp Counters has changed the situation. Starting with Windows 7/Server 2008 R2, Windows has a clear preference: Look for invariant TSCs, see whether they can be synchronized on different cores and use them for wall clock and performance counter whenever possible ("MSDN: Acquiring high-resolution time stamps.").

2.5. Discussion of Resources

Some of the resources discussed show platform-specific behavior. They may deliver results depending on the hardware and/or on Windows version. The precision time functions developed within the windows timestamp project mainly rely on four function suites provided by the operating system:

- GetSystemTimeAsFileTime
- QueryPerformanceCounter with QueryPerformanceFrequency
- Sleep
- The WaitableTimer function together with WaitForSingleObject

The complexity of the system time update with respect to the interrupt settings was explained and is understood. A complex automatic diagnosis of the system has to establish proper settings in order to obtain the desired accuracy. Particularly, the continuous calibration of the performance counter frequency described in 2.4.3 is of utmost importance to obtain a high accuracy. In addition, the proper interrupt period setting to obtain truly cyclic timer behavior (e.g., as described for example in 2.1) is very important. Another set of APIs is used to establish functionality:

- Pipes
- Events
- Shared Memory
- Mutexes (mutual exclusions)

The description of these functions falls outside the scope of this description.

3. Goals

The Windows timestamp project provides the tools to enable access to time at microsecond resolution and accuracy. Furthermore, it provides timer functions at the same resolution and accuracy. The high accuracy and microsecond resolution are archived by synchronizing the system time with the performance counter. In fact, the performance counter is phase locked to the system time. A diagnosis determines the system's specific parameters and establishes a "truly cyclic" timer interval for updating the phase of the performance counter value. The drift of the performance counter is permanently evaluated and taken into account while the system is running.

The code runs in a real-time priority process providing time information. An auxiliary IO process builds the interface to an optional graphical user interface. Nonblocking IO enables proper performance testing and debugging.

3.1. Time Support

Any time providing mechanism needs time for its internals. Thus, the following question arises with respect to time: Is the time requested at the time the call is made or shall the time be reported at the time in which the call returns? This may sound strange, but considering the level of resolution and accuracy aimed for here, it matters. Example:

- Something just happened and you want to assign a timestamp to it. In this case, you would want the time at the time you are asking for it.
- You want to do something at a specific time. In this case, you would want the time at the time that you are getting the answer.

Two time functions are implemented to fulfill these two needs:

3.1.1. GetTimeStamp

The function *GetTimeStamp*, declared as

```
void GetTimeStamp(TimeStamp_TYPE * TimeStamp);
```

fills the argument pointed to by *TimeStamp* with numbers according to the *TimeStamp* structure definition:

```
typedef struct {
    long long Time;
    long long ScheduledTime;
    double RefinedPerformanceCounterFrequency;
    long Accuracy;
    int State;
TimeStamp_TYPE;
```

The 64-bit value *Time* represents the number of elapsed 100-nanosecond intervals elapsed since January 1, 1601. *ScheduledDueTime* reports the system file time at which the next reference time is scheduled for an attempt to update the phase. This value should primarily be used to verify the operation of the precision time mechanism. If *ScheduledDueTime* is noticeable behind the current system file time, the scheduled update of the time reference must have failed for a number of consecutive attempts. Finally the 32-bit value of *Accuracy* gives an estimate of the assumed accuracy (rms) of the time stamp in 1 ns units (error in ns/s).

RefinedPerformanceCounterFrequency returns the calibrated frequency of results from QueryPerformanceCounter(). GetTimeStamp() may be called at any time, thus it provides information about the state of the calibration. States can be the following:

- TIME_STAMP_OFFLINE (1): Time calibration service is offline.
- TIME_STAMP_AWAITING_CALIBRATION (2): Calibration service just started but time service not yet calibrated.
- TIME_STAMP_CALIBRATED (3): Time service calibrated.
- TIME_STAMP_LICENSE_EXPIRED (4): License expired during runtime.

The call to *GetTimeStamp* is fast and it reports the time at the time it is called. As of version 2.01, this call is done in 10 to 20 ns on current platforms.

3.1.2. Time

A simple function is stated as

```
long long Time(void);
```

The function is as fast as *GetTimeStamp* and it returns the time at the time the call returns. With the need for a few thousand CPU cycles, the call will require very few µs with the current hardware. The *Time()* can be used to compare times or to wait until a certain time is observed. The 64-bit return value represents the number of elapsed 100-nanosecond intervals since January 1, 1601.

3.2. Timer Support

A set of timer functions:

- Creating a timed named event:

```
HANDLE CreateTimedEvent(
  BOOL bManualReset,
  LPCTSTR lpTimerName);
```

bManualReset [in]

If this parameter is TRUE, the function creates a manual reset event object, which requires the use of the ResetEvent function to set the event state to non-signaled. If this parameter is FALSE, the function creates an auto reset event object, and the system automatically resets the event state to non-signaled after a single waiting thread has been released.

lpTimerName [in, optional]

The name of the event object. The name is limited to MAX_PATH characters. Name comparison is case sensitive. If *lpTimerName* matches the name of an existing named event object, this function will fail. If *lpTimerName* is NULL, the event object is created without a name. If *lpTimerName* matches the name of another kind of object in the same namespace (such as an existing semaphore, mutex, waitable timer, job, or file-mapping object), the function fails and the GetLastError function returns ERROR_INVALID_HANDLE. This occurs because these objects share the same namespace. The name can have a "Global\" or "Local\" prefix to explicitly create the object in the global or session namespace. The remainder of the name can contain any character except the backslash character (\). For more information,

see Kernel Object Namespaces. Fast user switching is implemented using Terminal Services sessions. Kernel object names must follow the guidelines outlined for Terminal Services so that applications can support multiple users. The object can be created in a private namespace. For more information, see Object Namespaces.

Return value

If the function succeeds, the return value is a handle to the event object. If the named event object existed before the function call, the function returns NULL and GetLastError returns ERROR_ALREADY_EXISTS. If the function fails, the return value is NULL. To get extended error information, call GetLastError.

- Setting the timed event with a DueTime in 100 ns units and an optional Period in 100 ns units:

```
int SetTimedEvent(
  HANDLE hTimerEvent,
  long long TimerDueTime,
  long long TimerPeriod);
```

hTimerEvent [in]

A handle to a named timed event. The *CreateTimedEvent()* function returns this value.

TimerDueTime [in]

The time after which the state of the timer is to be set to signal, in 100 nanosecond intervals. Positive values indicate absolute time. Be sure to use a UTC-based absolute time since the system uses UTC-based time internally. Negative values indicate relative time.

TimerPeriod [in]

The period of the timer in 100ns intervals. If *TimerPeriod* is zero, the timer is signaled once. If *TimerPeriod* is greater than zero, the timer is periodic. A periodic timer automatically reactivates each time the period elapses, until the timer is canceled using the CancelTimedEvent function or reset using SetTimedEvent. If *TimerPeriod* is less than zero, the function fails.

Return value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information, call GetLastError.

- Canceling the timed event:

```
int CancelTimedEvent(HANDLE hTimerEvent);
```

hTimerEvent [in]

A handle to a named timed event. The *CreateTimedEvent()* function returns this value.

Return value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information, call GetLastError.

- Opening a timed event:

```
HANDLE OpenTimedEvent(LPCTSTR lpTimerName);
```

lpTimerName [in]

The timed event name used when the timed event was created.

Return value

If the function succeeds, the return value is the handle to the named timed event. If the function fails, the return value is NULL. To get extended error information, call GetLastError.

- Deleting the timed event:

```
int DeleteTimedEvent(HANDLE hTimerEvent);
```

hTimerEvent [in]

A handle to a named timed event. The *CreateTimedEvent()* function returns this value.

Return value

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information, call GetLastError.

These timer functions are based on timed events. The handle returned by *CreateTimedEvent()* is in fact a handle to a named event of which signaled state is supervised by a time service routine. Standard wait functions like *WaitForSingleObject* or *WaitForMultipleObjects* can be used to wait for the high resolution timer events.

4. Implementation

Only two hardware platforms were described here to highlight some of the problems to bear in mind when implementing reliable time services for Windows. Many more configurations need to be diagnosed to ensure platform independent functionality to a large extent. However, a flexible and automatic evaluation of hardware specific behavior may result in hardware independence.

The implementation of all the above into a time service is done by careful separation into different processes and threads. The time critical parts are hosted by a process running at real-time priority class. Some of the threads inside this process are even running at time-critical priority level. In the case of a multi-processor or multi-core system, certain threads are assigned to a specific CPU/core. This is the *Kernel* and hosts the time service routines. For testing and debugging the *Kernel* process has some IO capabilities shared with the IO process. A later version may not need this additional functionality. The high priority class requires the *Kernel* process to run with administrator privileges.

A second process hosts all kinds of less time-critical service threads. It shares some IO services with the *Kernel* process by means of piped IO between these two processes. Furthermore, it provides pipe services to the graphical user interface (GUI).

The third process is a graphical user interface (GUI), which runs optionally and helps in the current stage of the development to get an insight into what is going on.

The GUI and the IO process are development tools only. The only process that needs to run to access the time functions discussed here is the *Kernel* process.

4.1. The Real-time Priority Class Process: Kernel

The Kernel is the heart of the time service described here. It provides the important link between the system file time and the performance counter value. The idea in this context is to provide data triplets of system file time, performance counter, and performance counter frequency. Knowing the performance counter value at a certain system file time allows the extrapolation of the system file time to the actual time by applying the performance counter value and the performance counter frequency. As discussed, the performance counter frequency is of insufficient accuracy; a refined performance counter frequency is supplied in format double (64-bit float). There is also some internal information which allows a refinement of the performance counter value itself (as a result of some self calibration). Thus, it is also represented in double (64-bit float) format.

A typical result of such a data triplet could be:

- int64 FileTime: 129,737,733,817,343,750 (some value captured on Feb. 15, 2012)
- double PerformanceCounter: 20,918,865,472.23
- double PerformanceCounterFrequency: 3,579,515.24

This information is sufficient to establish time services. Querying the current performance counter value gives the difference to the value calculated to match the last captured file time. This difference is divided by the performance counter frequency and the result is the elapsed time since the last file time capture. This data triplet is, besides other parameters, written to a mutex protected shared memory section. Other processes/threads have access to this data triplet. As of version 2.02 the mutex scheme has been replaced by a read-copy-update (RCU) scheme to improve the performance.

As described in sections 2.1 and 2.3, the important part is to get the file time updated correctly. It proves best to gather the data triplet exactly when the file time transits or just transited. Difficulties archiving this have been described for platform examples A and B. At startup, a complex diagnosis of the interrupt timing structure and file time update/transition structure is performed. This results in a timing scheme for updating the data triplet. The desired update period is in the range of 1 to 10 seconds. As discussed, the period duration influences the accuracy. Algorithms are looking for patterns and beat frequencies in the file time update and interrupt timing structure. As a result, a periodic timer is set up to run the data triplet generation and the calibration in parallel. Once exact ΔT file time periods occur, the true performance counter frequency can be measured and averaged over a number of consecutive measurements. A running average over the last n captures is maintained at all times to provide information about the true (calibrated) performance counter frequency. When the accuracy of the average reaches a certain quality, the phase locking of file time change and performance counter is considered as established and timestamp requests are accompanied by information about their accuracy.

Running all of this at utmost priority ensures that there is very little overhead after an interrupt. Remember: Many processes/threads are waiting for interrupts. Therefore, systems have a workload peak at the occurrence of an interrupt. Even running at such priority settings, it is unavoidable to be influenced by the load of other processes. However, the accuracy of this scheme easily stays below a few microseconds, even with heavy load on the system.

The routines Time() and GetTimeStamp() are applying the extrapolation scheme described here. Both calls are done in far less than a few 10 ns, even on older systems.

The functionality of the timer routines listed in 3.2 is handled in this real-time process as well. Timed events are registered in a timer event queue. They are monitored with respect to their due time/period. When there is less than one interrupt period left before the due time expires, the timer service polls the timed event queue for the precise time to set the event. This may happen for a number of timed events, even within the same interrupt period. However, it should be noted that the time service thread is running at a high priority level and the signaled event may not be accessible to other processes/threads when there is just one CPU. A single CPU/core system simply cannot cope with multiple timed events setup to signal within the same interrupt period.

4.2. Less critical services: The IO-Process

In order to implement the kernel as small as possible, much of the functionality is performed by a second process. The IO, in particular, matters. The IO process establishes pipe services to release the kernel from blocking IO. All IO done by the kernel is queued into the IO processes pipe service. These operations are non-blocking. A complex fprintf() can be queued in just a few microseconds. This allows extensive output for diagnosis. Furthermore, output is logged into a file.

4.3. The Optional GUI-Process

The current GUI is mainly created for developing the time service. Meanwhile, it has become a valuable tool for diagnosing platforms. It runs optionally.

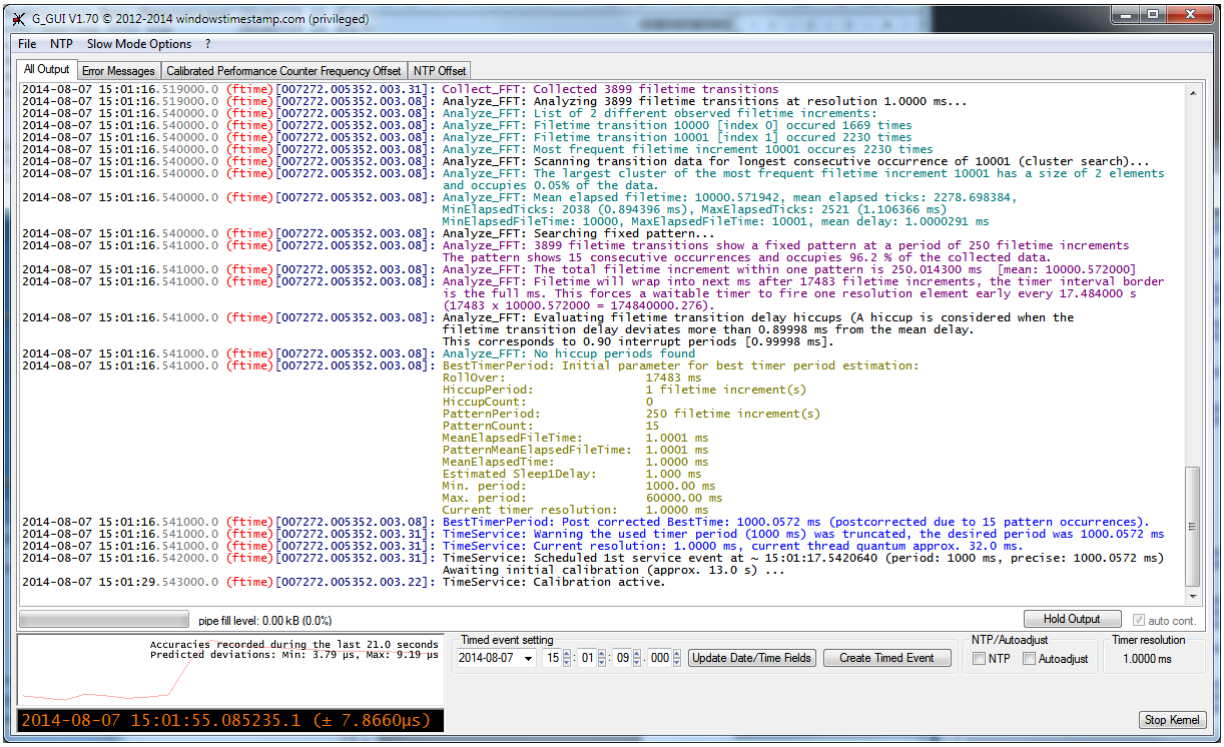

Fig. 4.3.1: The Graphical User Interface (Version 1.70).

The output is split into four tabs: the *all output* tab, the *error messages* tab, the *Calibrated Performance Counter Frequency Offset* tab, and the *NTP Offset* tab. The text output within the first two tabs is produced using the queued qfprintf(...) function. This function makes its message time stamped and shows also some other parameters of the output piping thread:

The output line format is:

yyyy-mm-dd hh:ii:ss.µµµµµµ.n (s/a)[PID.THID.Processor,Priority]: *Message*

- yyyy-mm-dd hh:ii:ss: Year-Month-Day Hour:Minute:Second
- µµµµµµ: 6 digits microseconds
- n: The 100ns units field
- s/a: Shows "ftime" while calibration is not ready, otherwise accuracy in microseconds/second
- PID: Identifier of the process
- THID: Identifier of the thread
- Processor: The number of the core or processor the thread runs on (0...n)
- Priority: A combination of thread priority level and process priority class (1...31)
- *Message*: The message

As already mentioned, the GUI runs optionally and any number of GUI can be started and ended at anytime. Ending a GUI will neither end the kernel process nor end the IO process. In order to terminate the whole group of processes, the *Kernel* process has to be stopped. The *Stop Kernel* button (lower right corner) stops the kernel. By doing so, queued messages that are supposed to be processed are stuck. A few message windows will pop up to show the contents of the unprocessed parts of the queues of all involved processes. These popup windows are not error messages; they just report what was happening while the *Kernel* was stopped.

The plot at the left lower corner shows the history of the accuracy in µs/s during the last 600 seconds. The GUI produces this information by means of GetTimeStamp() imported from the time service DLL.

It also provides a tiny test of the timer functionality: A single shot timer can be setup. The due time setting here is absolute, thus the time has to be in the future. Hint: Use the *Update Date/Time Fields* button to get the actual time into the fields and than incrementally change the minute field by 1. Press the *Create Timed Event* button quickly before the due time expires. Progress of the timed event approaching its due time is shown next to the button, which has now converted into a *Stop Timed Event Button* to allow cancellation of the timed event. A message window will popup when the timed event has signaled. It shows the precise time at which the signaled state was detected and how much it deviates from the requested due time.

The output can be stopped for the *all output* tab (*Hold Output* Button). All output will be queued and the button converts into a *Continue Output* button until the *Continue Output* button is pressed. An optional *auto cont.* check box lets the GUI continue automatically when the queue buffer reaches a critical stage. The *auto cont.* check box can only be checked when the output is hold.

The *Calibrated Performance Counter Frequency Offset* tab shows the offset of the calibrated performance counter frequency. The graph shown in 2.4.3 was created within this tab. The graphs context menu (right mouse button) allows saving the graph or

clearing the graph's data. Clearing the data will not stop further recording; creation of the graph will continue. Version 1.2 introduced the *NTP Offset* tab, the NTO/autoadjust status lines, and the NTP/autoadjust check boxes. Details about these items are given *in "Part II: Adjustment of System Time"*.

4.4. The Libraries

The functions described above are accessible to other processes/threads through a static library (LIB) or a dynamic link library (DLL).

5. Results

Microsecond resolution time stamps are possible on Windows systems. Resolution in the microsecond regime can be observed at accuracies of a few microseconds without distracting the system too much. Timer functions at the same resolution and accuracy are implemented and tested. Handling many timed events created by those timer functions set up to fire within the same millisecond is tricky but possible. The evaluation at the startup of the services may sometimes take a few seconds and needs all the CPU time. Doing this at utmost priority will freeze single core/processor systems for a moment.

Fine granularity time services are established with the tools described above. System time adjustment following an NTP time server relies on fine granularity of time keeping. The accuracy obtained when synchronizing to an NTP server is determined by the accuracy of the system time. Granularities of 15.625 ms are way to poor to achieve reasonable NTP synchronization. The time keeping has improved with newer windows versions. The function GetSystemTimePreciseAsFileTime() was described in 2.1.4.2. It is proposed to have very fine granularity. Unfortunately GetSystemTimePreciseAsFileTime() shows a misbehavior when a Windows system time adjustment is active. This is counterproductive when the goal is precise synchronization of the system time to an NTP server.

The following second part "Adjustment of System Time" deals with high accuracy system synchronization.

...

Note: Don't miss more details described on the "News" page. A pdf version of the **News History** can be downloaded here.