Timing Pitfalls and Solutions

Jan Wassenberg

June 10, 2007

Abstract

The seemingly simple task of retrieving a high-resolution timestamp on PCs is fraught with peril. Various hardware and software defects can induce discontinuities in measured time, causing some applications to fail. After giving a brief overview of time sources in the PC architecture, this report explores their pitfalls and presents a high-resolution timing implementation that avoids falling into them.

1 Introduction

High-resolution timing is important for several applications. Profilers[1] depend on particularly fine-grained measurements so they can examine the performance of small sections of code. Interactive games require precise timestamps to enable smooth updates. Finally, multimedia applications also benefit from accurate timing to synchronize their audio and video streams.

The ideal timer would be a high-resolution, low jitter, strictly increasing source of UTC timestamps. Of these properties, monotonicity is the most important, since many applications understandably cannot cope with their time reference running backwards. Conversely, exact synchronization of existing hardware counters to the time of day appears infeasible and even undesirable – this issue is discussed in Appendix A. In the following, emphasis will be placed on acquiring precise relative timestamps.

How close can we come to reaching this goal on today's systems? Unix BSD derivatives provide the `gettimeofday` routine, which delivers a high-resolution rendition of the time that can easily be converted to UTC. The

[1] Programs to find bottlenecks in other programs by analyzing their execution time.

returned time values are not monotonic nor mostly continuous, but this capability is provided by the POSIX `clock_gettime(CLOCK_MONOTONIC)` facility[2], which meets all our needs. On Windows, however, the situation is much worse. Due to documented bugs and the lack of a universal patch, the `QueryPerformanceCounter` high-resolution timer API cannot safely be used. On certain systems, only a timer with mere millisecond resolution is available, and even this is encumbered with the issue of falling behind over time. The need for further work to improve upon this miserable state of affairs is thus apparent.

The remainder of this report is structured as follows: Section 2 introduces the various hardware timers in the PC architecture. Section 3 lists the Windows timing APIs that access them. Section 4 goes on to explore their respective pitfalls. Section 5 presents possible workarounds that solve them. Section 6 is concerned with notes pertaining to the implementation of these techniques. Section 7 then concludes the discussion with a summary of the results.

2 PC Timing Hardware

Due to incremental additions to the PC architecture, there are several hardware modules that can be used to obtain timestamps. For each of them (in order of adoption), we will see a brief overview of their functionality and disadvantages.

2.1 PIT

The 8254 Programmable Interval Timer was included in the original PC. At its heart are three 16-bit counters running at 1.19 MHz[3]. They are accessible at fixed addresses in I/O-space, but must be latched and then read as two single-byte values, which takes 3 μs – the slowest of all the timers. Additionally, the narrow counter registers roll over too often to measure anything longer than 55 ms intervals. Instead, the PIT's utility lies in its ability to generate periodic interrupts.

[2]Although `CLOCK_MONOTONIC` is specified as an optional feature, support for it is included in glibc 2.3.3, so it is expected to be widely available. [Drepper 2006]

[3]To save costs, the original PC used a master 14.318 MHz oscillator and fed system components with various ratios of this frequency – in this case, 1/12.

2.2 RTC

The Real-Time Clock was added in XT 286 machines. It also includes a counter running at 32 kHz but does not provide for direct access. Instead, the chip uses the count to keep track of the time of day and also generate periodic interrupts. Its only use nowadays seems to be storing the local time in battery-buffered CMOS, which is then read during boot time and maintained by the OS.

2.3 PMT

The Power Management Timer was added as part of the ACPI standard. Its sole purpose is to deliver timestamps; improvements over the PIT are to be found in its higher frequency (3x), larger counter width (24 or 32 bits) and latch-free operation. It is accessed via 32-bit port I/O (at an address given in the ACPI `FACP` table), which only takes 0.7 μs.

2.4 TSC

The TimeStamp Counter has been available starting with fifth-generation CPUs (Pentiums) and represents a 64-bit count of elapsed CPU cycles since power-on. It can be retrieved with the `RDTSC` instruction, or by directly reading the model-specific[4] register that holds it; both methods are extremely fast, on the order of dozens of CPU cycles. Since the CPU clock frequency is much higher than the that of the other hardware counters, a very fine resolution is attained. Unfortunately this is counterbalanced somewhat by the low quality of typical clock oscillators and their thermal drift.

2.5 HPET

The High-Precision Event Timer is a recent addition to the architecture, specified over 2001–2004 and present in ICH8-based chipsets. It was motivated by troubles with the TSC, the inadequacy of the PMT and the desire to replace the legacy PIT. The main components are a master 32 or 64-bit counter running at a frequency of at least 10 MHz (usually the 14.318 MHz

[4]Note that this particular register will probably remain supported due to its widespread use.

source) and multiple comparator registers. Access is via memory-mapped I/O (at an address indicated by the ACPI `HPET` table), which is very convenient and also fast, taking 0.9 μs.

2.6 Others

There are some other esoteric ways to get the time, e.g. polling the DRAM refresh detect bit, but these aren't relevant here. The interested reader is referred to [Heidenstrom 1995].

3 Windows time APIs

Having introduced the underlying hardware, we now examine the multitude of Windows timing APIs, their overhead, and how they map to these time sources. The relevant functions are:

GTC `GetTickCount` returns a counter incremented during every (PIT) clock interrupt. Overhead is 10 cycles (load, mul, shift).

GSTAFT `GetSystemTimeAsFileTime` is also updated during clock interrupts, but returns the system time (subject to NTP adjustment) in higher resolution units (hectonanoseconds). Each call takes 6 clocks (64-bit load/store + synchronization[5]).

TGT `timeGetTime` is a third interrupt-based timer. Its resolution can be improved to 1 ms via `timeBeginPeriod` by speeding up the interrupt rate[6]. This function takes 40 cycles; I have not analysed what it does in detail.

[5]The system time is mapped into process memory. GSTAFT must make sure it returns consistent results because the value can change at any time and 64-bit reads may not be atomic. This is done here by comparing against a second copy of the MSW, and reading again if it is different (avoids the need for a spin lock).

[6]Note that this comes at the price of system slowdown – interrupt latency goes up and the CPU cache suffers. [MS Timer Guidelines 2002] mentions offhand that 1000 Hz interrupt rates incur 'serious' overhead, corroborated by findings of 14 % indirect slowdown by Tsafrir u. a. [2005]

QPC `QueryPerformanceCounter` is the main platform-independent means of accessing hardware timers. The various versions of Windows differ in their choice of timer: PIT (Win2k), PMT (WinXP), HPET (Vista) or TSC (SMP HAL[7]).

Note that C library functions such as `_ftime` and `clock` are built on top of GSTAFT.

4 Pitfalls

What potential problems do applications face when using these functions? This discussion centers on the above APIs, but most issues are caused by hardware flaws and are therefore not specific to Windows.

GTC and GSTAFT resolution is far too low – the default 10 or 15 ms tick interval isn't enough to resolve individual frames in a game, not to mention far too coarse for profiling.

TGT can fall behind over time due to delayed or lost clock interrupts[8]. This is reported to have caused problems:

> In retrospect, timeGetTime() slippage is almost certainly what is killing many networked games of StarTopia [..;] of course all our testing in the office was on mainly one or two very similar models of Dell, so we never saw this problem, even after many hours. Curse those hardware designers!
>
> Forsyth [2002]

QPC has multiple potential issues, depending on which time source is in use:

PIT Access latency is inconveniently high, causing undesirable overhead when used for profiling purposes.

PMT Timestamps can jump forward by several hundred ms during heavy PCI bus load. [QPC Jumps 2006]

PMT Results are undefined if the timer is not polled at least once every 4.6 seconds[9]. This is caused by hardware bugs and/or incorrect software handling of counter overflow.

[7] Hardware Abstraction Layer for Symmetric MultiProcessing systems

[8] Interrupts can be lost during periods of heavy system load, which games tend to provoke.

[9] The period of the underlying 24-bit counter with its 3.57 MHz frequency.

TSC MSDN [2005] makes brief and vague mention of problems:

> [Y]ou can get different results on different processors due to bugs in the basic input/output system (BIOS) or the hardware abstraction layer (HAL).

This is because TSCs do not start counting at the same time (due to staggered boot-time processor initialization), nor do they run at the same rate (due to independent throttling of cores). Unfortunately, Windows apparently does not take the requisite measures to guarantee synchronization.

None of the timer APIs are free of potential problems, so we must consider workarounds.

5 Workarounds

There are two principal ways to work around potentially incorrect timers. The first is to consult three or more independent timers and ensure they agree on the time. [Watte] However, it is unclear what to do if the timers vote differently – which one of them is to be considered correct? This problem is compounded by the fact that some timers may always return incorrect results (e.g. QPC using the TSC without proper safeguards).

An alternative to consensus-based operation is to choose one single timer that is known to be safe on the current platform. This approach is preferable because it is more predictable and robust; however, it does require all potential problems to be known. For a full list of the issues that have been considered, see the extensive comments in the timer implementations' `IsSafe` routines. To choose the best possible safe timer, they are evaluated in order of finer resolution and lower measuring overhead.

One additional measure must be taken: periodically polling the timer fixes the PMT wraparound problem mentioned above. As a bonus, this also allows frequency calibration (helpful to reduce the effects of thermal drift of the TSC timer).

In summary, we rely on knowledge of existing bugs and limitations to choose a safe timer.

5.1 An unfortunate constellation

There exists one common combination of software and hardware where no safe high-resolution time source is available. Current dual-core systems fail to provide a synchronized TSC; capabilities such as an 'invariant TSC' [Brunner 2005] and the RDTSCP instruction [Nagendra 2007] are coming into play, but not yet widespread. In the meantime, since Windows does not offer a ubiquitous software solution, the TSC is not reliable. In an unfortunate twist, it is precisely this timer that QPC uses on such dual-core platforms. One can force it to use the PMT by adding `/usepmtimer` to `boot.ini` [QPC Workaround 2006], but making changes to such a critical system file should not be asked of users. Finally, despite Microsoft having issued the HPET specification 5 years ago, support for it has not been patched into Windows XP and it is only available on Vista.

These circumstances result in the moribund state of affairs that was previously described: on many systems today, only a mere millisecond-resolution timer is available. According to Valve Survey [2007], 93 % of the participating systems are running Windows XP, and 22 % have more than one processor or core. This large proportion cannot be ignored, so we will have to find a workaround for the above issue.

5.2 Solution: HPET

Instead of pursuing the very complex task of fixing all problems with the TSC, we take a step back and realize that dual-core systems are newer and often include support for the HPET. Developing a driver for it on Windows XP should resolve the above situation. Note that limited-permission user accounts have only become widespread with Windows Vista and this driver only needs to run on Windows XP. We can therefore preassume Administrator rights, which allow starting the driver at run-time to avoid the need for installation.

Interfacing with the HPET only requires analyzing ACPI tables and accessing memory-mapped registers. It is thus sufficient to write a kernel-mode driver that maps physical memory to virtual addresses on behalf of the application. Since the remaining logic can run in user mode, the driver is simplified and the risk of a defect that crashes the entire OS is much reduced.

Since the HPET Specification [2004] is quite clear, an additional description of its operation is deemed unnecessary. For further reference, see the straightforward implementation in `hpet.cpp`.

6 Implementation

With the principles of our approach to safe high-resolution timing having been established, discussion now turns to various notes on its implementation.

Recall that an update thread is required to prevent hardware counter overflow. This works by periodically latching the value and calculating the current time via ticks elapsed since then. How can these updates to the timer state be synchronized with calls to retrieve the time? It turns out that high-overhead and error-prone locks can be avoided with an approach similar to RCU. [McKenney u. a. 2001] All timer state is maintained in a record and accessed via pointer. When the update thread becomes active, it linearizes[10] the state change by first building a separate record and then atomically updating the pointer. State reads need only retrieve all fields and ensure an update did not occur during that time. Discarding previous state information is achieved by simply swapping the records every update; this avoids the thorny problem of freeing data that another thread may still be using.

The timer interface is very simple. `whrt_GetTime` returns the elapsed time since startup, conveniently given as double-precision floating-point values representing seconds. Assuming a reasonable timer frequency of 10 MHz (the minimum HPET tick rate), stored timestamps can span a continuously measured period of 28 years without any error, which indicates plenty of headroom. `whrt_Resolution` gives the timer resolution, which allows choosing a minimum duration for calibration/delay loops (e.g. for measuring the CPU frequency) that will still provide a given accuracy.

We note `DeviceIoControl`-based access to the PMT is much slower than `QueryPerformanceCounter` when it uses that source. The overhead is likely due to slower kernel entry (possibly not using `sysenter`) and a less direct path to the hardware. As a consequence, the PMT is only accessed directly if it is determined that QPC is unsafe.

[10] Makes an operation appear to happen in one instant, namely the linearization point.

7 Conclusion

Timing on PCs is a surprisingly thorny task. Various known problems have been discussed and a serviceable solution found. The contribution of this work is a means of choosing and safely using the best available hardware timer, as well as to unlock the reliable HPET timer on Windows XP systems. It provides a stable time reference with a resolution of at least 0.1 μs. Hopefully this discussion and the attached (Free) source code will help prevent further timing issues in games and elsewhere; pay it forward!

Questions/comments/suggestions/bug reports are welcome.
u9rkt@stud1.uni-karlsruhe.de (remove digits)

A Synchronizing with time-of-day

The above discussion has yielded a precise high-resolution timer. Ideally, it should be accurate as well, i.e. disciplined to UTC: this would obviate separate timekeeping and synchronization schemes.

One simple method of achieving this would be to let the timer free-run until it differs too much from the system time, and then bring them in line. However, this is unacceptable due to the possibility of observed time moving backwards – timestamps must increase monotonically.

A better method is to slew the timer frequency according to the current time difference with respect to UTC. The success of this method hinges on exact observations of the system time. Without OS support, the best we can do is rely on the assumption that GSTAFT and the scheduler are driven by the same clock, i.e. a high-priority thread wakes up immediately after an update to the system time.[11] Given the now hopefully exact system time, we must build a stable phase-lock loop that converges on it, and doesn't overshoot too much / bounce back and forth. [Mills 1994] The central task is finding an error correction function $h(\Delta) \mapsto$ slew. The simple method given in [Kenny 2000] is a p-controller [12] with clamped maximum adjustment. In my tests, its gain was way too high, and it was bouncing back and forth at the maximum

[11] Polling for the start of the next tick is unacceptable because it would incur worst-case delays of 10 ms.

[12] Proportional: $adjust = gain \times error$. A typical example is a float regulator. Disadvantages: depending on gain, it either doesn't track the signal rapidly, or it is unstable, and diverges. Also, if slightly above/below the target value, it will never correct the difference, because its adjust goes to 0.

allowed adjustment value. A PID controller [13] is more suitable: the timer can be kept within several μs (!) of the system time.

In retrospect, however, it is unclear whether conjoining high-resolution timestamps and the time of day is actually desirable. Many uses of system time (e.g. recording file modification times) usually do not need to be high-resolution. Conversely, high-resolution time applications (e.g. profilers) do not require synchronization or freedom from long-term drift. In fact, saddling the timer with time-of-day anomalies may degrade the value of the timestamps due to non-uniform updates. Finally, it has been observed that the system time had not yet been updated after thread wakeup[14], thus violating the central assumption above. Therefore, we keep the timer separate from UTC and transfer the responsibility of synchronizing clocks to users who actually need this feature.

References

[Brunner 2005] Brunner, Rich: *TSC and Power Management Events on AMD Processors*. November 2 2005. – URL `http://lkml.org/lkml/2005/11/4/173`. – Posted on Linux Kernel Mailing List

[Drepper 2006] Drepper, Ulrich: *POSIX Option Groups*. August 20 2006. – URL `http://people.redhat.com/drepper/posix-option-groups.html`

[Forsyth 2002] Forsyth, Tom: *RE: [GD-Windows] More timer fun!* gmane.games.devel.windows Newsgroup. September 2 2002. – URL `http://article.gmane.org/gmane.games.devel.windows/105/`

[Heidenstrom 1995] Heidenstrom, Kris: *Timing on the PC family under DOS*. Dezember 1995. – URL `ftp://garbo.uwasa.fi/pc/programming/pctim003.zip`

[HPET Specification 2004] Intel Corporation (Veranst.): *IA-PC HPET (High Precision Event Timers) Specification*. Oktober 2004. – URL `http://www.intel.com/hardwaredesign/hpetspec_1.pdf`

[Kenny 2000] Kenny, Kevin: *Increased resolution for TclpGetTime on*

[13] P, and Integral / Differential parts. I sums up previous error values, so that slight offsets can be corrected. D sees a 'trend' early and adjusts for it, thereby tracking the signal more rapidly.

[14] Speculation: does this happen in a DPC?

Windows. Oktober 26 2000. – URL http://www.tcl.tk/cgi-bin/tct/tip/7.html

[McKenney u. a. 2001] McKenney, Paul E. ; Kleen, Andi ; Krieger, Orran ; Russell, Rusty ; Sarma, Dipankar ; Soni, Maneesh: *Read-Copy Update*. Oktober 23 2001. – URL http://www.eecg.toronto.edu/parallel/pubs_tornado/ols2001.pdf

[Mills 1994] Mills, David: *A Kernel Model for Precision Timekeeping*. März 1994. – URL http://www.eecis.udel.edu/~mills/database/rfc/rfc1589.txt

[MS Timer Guidelines 2002] : *Guidelines For Providing Multimedia Timer Support*. September 20 2002. – URL http://www.microsoft.com/whdc/system/CEC/mm-timer.mspx

[MSDN 2005] Microsoft Corporation (Veranst.): *MSDN: QueryPerformanceCounter Function*. 2005

[Nagendra 2007] Nagendra, Bhavana: *AMD TSC Drift Solutions in Red Hat Enterprise Linux*. 2007. – URL http://developer.amd.com/articles.jsp?id=92&num=7

[QPC Jumps 2006] : *Performance counter value may unexpectedly leap forward*. November 21 2006. – URL http://support.microsoft.com/default.aspx?scid=KB;EN-US;Q274323&

[QPC Workaround 2006] : *Programs that use the QueryPerformanceCounter function may perform poorly in Windows Server 2003 and in Windows XP*. November 14 2006. – URL http://support.microsoft.com/kb/895980/en-us

[Tsafrir u. a. 2005] Tsafrir, Dan ; Etsion, Yoav ; Feitelson, Dror: General-Purpose Timing: The Failure of Periodic Timers / The Hebrew University of Jerusalem. School of Computer Science and Engineering, Februar 2005 (6). – Forschungsbericht. – URL http://www.cs.huji.ac.il/~dants/papers/Timing05TR.pdf

[Valve Survey 2007] : *Valve Survey Summary*. Juni 9 2007. – URL http://www.steampowered.com/status/survey.html

[Watte] Watte, Jon: *PC Timers*. – URL http://www.mindcontrol.org/~hplus/pc-timers.html