Timecounters: Efficient and precise timekeeping in SMP kernels.

Poul-Henning Kamp
The FreeBSD Project

ABSTRACT

The FreeBSD timecounters are an architecture-independent implementation of a binary timescale using whatever hardware support is at hand for tracking time. The binary timescale converts using simple multiplication to canonical timescales based on micro- or nano-seconds and can interface seamlessly to the NTP PLL/FLL facilities for clock synchronisation. Timecounters are implemented using lock-less stable-storage based primitives which scale efficiently in SMP systems. The math and implementation behind timecounters will be detailed as well as the mechanisms used for synchronisation.

Introduction

Despite digging around for it, I have not been able to positively identify the first computer which knew the time of day. The feature probably arrived either from the commercial side so service centres could bill computer cycles to customers or from the technical side so computers could timestamp external events, but I have not been able to conclusively nail the first implementation down.

But there is no doubt that it happened very early in the development of computers and if systems like the ‘‘SAGE’’ [SAGE] did not know what time it was I would be amazed.

On the other hand, it took a long time for a real time clock to become a standard feature:

The ‘‘Apple][’’ computer had neither in hardware or software any notion what time it was.

The original ‘‘IBM PC’’ did know what time it was, provided you typed it in when you booted it, but it forgot when you turned it off.

One of the ‘‘advanced technologies’’ in the ‘‘IBM PC/AT’’ was a battery backed CMOS chip which kept track of time even when the computer was powered off.

Today we expect our computers to know the time, and with network protocols like NTP we will usually find that they do, give and take some milliseconds.

This article is about the code in the FreeBSD kernel which keeps track of time.

Time and timescale basics

Despite the fact that time is the physical quantity (or maybe entity ?) about which we know the least, it is at the same time [sic!] what we can measure with the highest precision of all physical quantities.

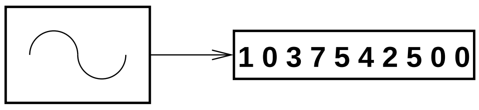
The current crop of atomic clocks will neither gain nor loose a second in the next couple hundred million years, provided we stick to the preventative maintenance schedules. This is a feat roughly in line with to knowing the circumference of the Earth with one micrometer precision, in real time.

While it is possible to measure time by means other than oscillations, for instance transport or consumption of a substance at a well-known rate, such designs play no practical role in time measurement because their performance is significantly inferior to oscillation based designs.

In other words, it is pretty fair to say that all relevant timekeeping is based on oscillating phenomena:

sun-dial	Earths rotation about its axis.
calendar	Ditto + Earths orbit around the sun.
clockwork	Mechanical oscillation of pendulum.
crystals	Mechanical resonance in quartz.
atomic	Quantum-state transitions in atoms.

We can therefore with good fidelity define ‘‘a clock’’ to be the combination of an oscillator and a counting mechanism:



Oscillator + Counter = Clock

The standard second is currently defined as

> The duration of 9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium 133 atom.

and we have frequency standards which are able to mark a sequence of such seconds with an error less than $2 \cdot 10^{-15}$ [DMK2001] with commercially available products doing better than $1 \cdot 10^{-14}$ [AG2002].

Unlike other physical units with a conventionally defined origo, longitude for instance, the ephemeral nature of time prevents us from putting a stake in the ground, so to speak, and measure from there. For measuring time we have to rely on ‘‘dead reckoning’’, just like the navigators before Harrison built his clocks [RGO2002]: We have to tally how far we went from our reference point, keeping a running total at all times, and use that as our estimated position.

The upshot of this is, that we cannot define a timescale by any other means than some other timescale(s).

‘‘Relative time’’ is a time interval between two events, and for this we only need to agree on the rate of the oscillator.

‘‘Absolute time’’ consists of a well defined point in time and the time interval since then, this is a bit more tricky.

The Internationally agreed upon TAI and the UTC timescales starts at (from a physics point of view) arbitrary points in time and progresses in integral intervals of the standard second, with the difference being that UTC does tricks to the counting to stay roughly in sync with Earths rotation [1].

TAI is defined as a sequence of standard seconds (the first timescale), counted from January 1st 1958 (the second timescale).

UTC is defined basically the same way, but every so often a leap-second is inserted (or theoretically deleted) to keep UTC synchronised with Earths rotation.

Both the implementation of these two, and a few others speciality timescales are the result of the combined efforts of several hundred atomic frequency standards in various laboratories and institutions throughout the world, all reporting to the BIPM in Paris who calculate the ‘‘paper clock’’ which TAI and UTC really are using a carefully designed weighting algorithm [2].

[1]The first atomic based definition actually operated in a different way: each year would have its own value determined for the frequency of the caesium resonance, selected so as to match the revolution rate of the Earth. This resulted in time-intervals being very unwieldy business, and more and more scientists realized that that the caesium resonance was many times more stable than the angular momentum of the Earth. Eventually the new leap-second method were introduced in 1972. It is interesting to note that the autumn leaves falling on the northern hemisphere affects the angular momentum enough to change the Earths rotational rate measurably.

[2]The majority of these clocks are model 5071A from Agilent (the test and measurement company formerly known as ‘‘Hewlett-Packard’’) which count for as much as 85% of the combined weight. A fact the company deservedly is proud of. The majority of the remaining weight is assigned to a handful of big custom-design units like the PTB2 and NIST7.

Leap seconds are typically announced six to nine months in advance, based on precise observations of median transit times of stars and VLBI radio astronomy of very distant quasars.

The perceived wisdom of leap-seconds have been gradually decreasing in recent years, as devices and products with built-in calendar functionality becomes more and more common and people realize that user input or software upgrades are necessary to instruct the calendar functionality about upcoming leap seconds.

UNIX timescales

UNIX systems use a timescale which pretends to be UTC, but defined as the count of standard seconds since 00:00:00 01-01-1970 UTC, ignoring the leap-seconds. This definition has never been perceived as wise.

Ignoring leap seconds means that unless some trickery is performed when a leap second happens on the UTC scale, UNIX clocks would be one second off. Another implication is that the length of a time interval calculated on UNIX time_t variables, can be up to 22 (and counting) seconds wrong relative to the same time interval measured on the UTC timescale.

Recent efforts have tried to make the NTP protocol make up for this deficiency by transmitting the UTC-TAI offset as part of the protocol. [MILLS2000A]

Fractional seconds are represented two ways in UNIX, ‘‘timeval’’ and ‘‘timespec’’. Both of these formats are two-component structures which record the number of seconds, and the number of microseconds or nanoseconds respectively.

This unfortunate definition makes arithmetic on these two formats quite expensive to perform in terms of computer instructions:

```
/* Subtract timeval from timespec */
t3.tv_sec = t1.tv_sec - t2.tv_sec;
t3.tv_nsec = t1.tv_nsec -
             t2.tv_usec * 1000;
if (t3.tv_nsec >= 1000000000) {
    t3.tv_sec++;
    t3.tv_nsec -= 1000000000;
} else if (t3.tv_nsec < 0) {
    t3.tv_sec--;
    t3.tv_nsec += 1000000000;
}
```

While nanoseconds will probably be enough for most timestamping tasks faced by UNIX computers for a number of years, it is an increasingly uncomfortable situation that CPU clock periods and instruction timings are already not representable in the standard time formats available on UNIX for consumer grade hardware, and the first POSIX mandated API, `clock_getres(3)` has already effectively reached end of life as a result of this.

Hopefully the various standards bodies will address this issue better in the future.

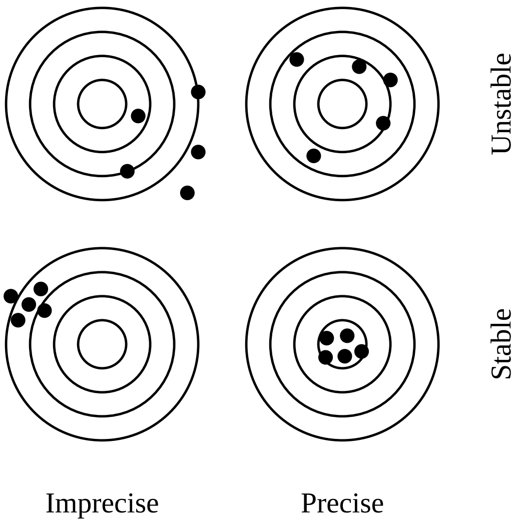
Precision, Stability and Resolution

Three very important terms in timekeeping are ‘‘precision’’, ‘‘stability’’ and ‘‘resolution’’. While the three words may seem to describe somewhat the same property in most uses, their use in timekeeping covers three very distinct and well defined properties of a clock.

Resolution in clocks is simply a matter of the step-size of the counter or in other words: the rate at which it steps. A counter running on a 1 MHz frequency will have a resolution of 1 microsecond.

Precision talks about how close to the intended rate the clock runs, stability about how much the rate varies and resolution about the size of the smallest timeinterval we can measure.

From a quality point of view, Stability is a much more valuable property than precision, this is probably best explained using a graphic illustration of the difference between the two concepts:

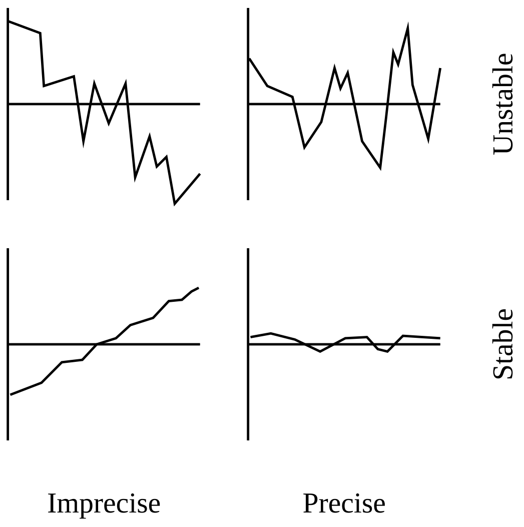


In the top row we have instability, the bullet holes are spread over a large fraction of the target area. In the bottom row, the bullets all hit in a very small area.

On the left side, we have lack of precision, the holes obviously are not centred on the target, a systematic offset exists. In the right side we have precision, the bullets are centred on the target [3].

[3] We cannot easily get resolution into this analogy, the obvious representation as the diameter of the bullet-hole is not correct, it would have to be the grid or other pattern of locations where the bullet could possibly penetrate the target material, but this gets too quantum-mechanical-oid to serve the instructional purpose.

Transposing these four targets to actual clocks, the situation could look like the following plots:



On the x-axis we have time and on the y-axis how wrong the clock was at a given point in time.

The reason atomic standards are such a big deal in timekeeping is that they are incredibly stable: they are able to generate an oscillation where the period varies by roughly a millionth of a billonth of a second in long term measurements.

They are in fact not nearly as precise as they are stable, but as one can see from the graphic above, a stable clock which is not precise can be easily corrected for the offset and thus calibrated is as good as any clock.

This lack of precision is not necessarily a flaw in these kinds of devices, once you get into the $10 \cdot 10^{-15}$ territory things like the blackbody spectrum at the particular absolute temperature of the clocks hardware and general relativistic effects mostly dependent on the altitude above earths center has to be corrected for [4].

Design goals of timecounters

After this brief description of the major features of the local landscape, we can look at the design goals of timecounters in detail:

Provide timestamps in timeval and timespec formats,

This is obviously the basic task we have to solve, but as was noted earlier, this is in no way the performance requirement.

on both the ‘‘uptime’’ and the POSIX timescales,

The ‘‘uptime’’ timescale is convenient for time

[4] This particularly becomes an issue with space-based atomic standards as those found on the ‘‘Navstar’’ GPS satellites.

intervals which are not anchored in UTC time: the run time of processes, the access time of disks and similar.

The uptime timescale counts seconds starting from when the system is booted. The POSIX/UTC timescale is implemented by adding an estimate of the POSIX time when the system booted to the uptime timescale.

using whatever hardware we have available at the time,

Which in a subtle way also implies ‘‘be able to switch from one piece of hardware to another on the fly’’ since we may not know right up front what hardware we have access to and which is preferable to use.

while supporting time the NTP PLL/FLL discipline code,

The NTP kernel PLL/FLL code allows the local clock and timescale to be synchronised or syntonised to an external timescale either via network packets or hardware connection. This also implies that the rate and phase of the timescale must be manoeuvrable with sufficient resolution.

and providing support for the RFC 2783 PPS API,

This is mainly for the benefit of the NTPD daemons communication with external clock or frequency hardware, but it has many other interesting uses as well [PHK2001].

in a SMP efficient way.

Timestamps are used many places in the kernel and often at pretty high rate so it is important that the timekeeping facility does not become a point of CPU or lock contention.

Timecounter timestamp format.

Choosing the fundamental timestamp format for the timecounters is mostly a question of the resolution and steer-ability requirements.

There are two basic options on contemporary hardware: use a 32 bit integer for the fractional part of seconds, or use a 64 bit which is computationally more expensive.

The question therefore reduced to the somewhat simpler: can we get away with using only 32 bit ?

Since 32 bits fractional seconds have a resolution of slightly better than quarter of a nanosecond (.2328 nsec) it can obviously be converted to nanosecond resolution struct timespec timestamps with no loss of precision, but unfortunately not with pure 32 bit arithmetic as that would result in unacceptable rounding errors.

But timecounters also need to represent the clock period of the chosen hardware and this hardware might be the GHz range CPU-clock. The list of clock frequencies we could support with 32 bits are:

$$\begin{array}{lcrl}
2^{32}/1 & = & 4.294 & \text{GHz} \\
2^{32}/2 & = & 2.147 & \text{GHz} \\
2^{32}/3 & = & 1.432 & \text{GHz} \\
\ldots & & & \\
2^{32}/(2^{32}-1) & = & .999 & \text{Hz}
\end{array}$$

We can immediately see that 32 bit is insufficient to faithfully represent clock frequencies even in the low GHz area, much less in the range of frequencies which have already been vapourwared by both IBM, Intel and AMD. QED: 32 bit fractions are not enough.

With 64 bit fractions the same table looks like:

$$\begin{array}{lcrl}
2^{64}/1 & = & 18.45 \cdot 10^9 & \text{GHz} \\
2^{64}/2 & = & 9.223 \cdot 10^9 & \text{GHz} \\
\ldots & & & \\
2^{64}/2^{32} & = & 4.294 & \text{GHz} \\
\ldots & & & \\
2^{64}/(2^{64}-1) & = & .999 & \text{Hz}
\end{array}$$

And the resolution in the 4 GHz frequency range is approximately one Hz.

The following format have therefore been chosen as the basic format for timecounters operations:

```
struct bintime {
    time_t  sec;
    uint64_t frac;
};
```

Notice that the format will adapt to any size of time_t variable, keeping timecounters safely out of the ‘‘We SHALL prepare for the Y2.038K problem’’ war zone.

One beauty of the bintime format, compared to the timeval and timespec formats is that it is a binary number, not a pseudo-decimal number. If compilers and standards allowed, the representation would have been ‘‘int128_t’’ or at least ‘‘int96_t’’, but since this is currently not possible, we have to express the simple concept of multiword addition in the C language which has no concept of a ‘‘carry bit’’.

To add two bintime values, the code therefore looks like this [5]:

```
uint64_t u;

u = bt1->frac;
bt3->frac = bt1->frac + bt2->frac;
bt3->sec  = bt1->sec  + bt2->sec;
if (u > bt3->frac)
    bt3->sec += 1;
```

[5] If the reader suspects the ’>’ is a typo, further study is suggested.

An important property of the bintime format is that it can be converted to and from timeval and timespec formats with simple multiplication and shift operations as shown in these two actual code fragments:

```
void
bintime2timespec(struct bintime *bt,
                 struct timespec *ts)
{

    ts->tv_sec = bt->sec;
    ts->tv_nsec =
      ((uint64_t)1000000000 *
      (uint32_t)(bt->frac >> 32)) >> 32;
}
```

```
void
timespec2bintime(struct timespec *ts,
                 struct bintime *bt)
{

    bt->sec = ts->tv_sec;
    /* 18446744073 =
      int(2^64 / 1000000000) */
    bt->frac = ts->tv_nsec *
      (uint64_t)18446744073LL;
}
```

How timecounters work

To produce a current timestamp the timecounter code reads the hardware counter, subtracts a reference count to find the number of steps the counter has progressed since the reference timestamp. This number of steps is multiplied with a factor derived from the counters frequency, taking into account any corrections from the NTP PLL/FLL and this product is added to the reference timestamp to get a timestamp.

This timestamp is on the ‘‘uptime’’ time scale, so if UNIX/UTC time is requested, the estimated time of boot is added to the timestamp and finally it is scaled to the timeval or timespec if that is the desired format.

A fairly large number of functions are provided to produce timestamps, depending on the desired timescale and output format:

Desired Format	uptime timescale	UTC/POSIX timescale
bintime	binuptime()	bintime()
timespec	nanouptime()	nanotime()
timeval	microuptime()	microtime()

Some applications need to timestamp events, but are not particular picky about the precision. In many cases a precision of tenths or hundreds of seconds is sufficient.

A very typical case is UNIX file timestamps: There is little point in spending computational resources getting an exact nanosecond timestamp, when the data is written to a mechanical device which has several milliseconds of unpredictable delay before the operation is completed.

Therefore a complementary shadow family of timestamping functions with the prefix ‘‘get’’ have been added.

These functions return the reference timestamp from the current timehands structure without going to the hardware to determine how much time has elapsed since then. These timestamps are known to be correct to within rate at which the periodic update runs, which in practice means 1 to 10 milliseconds.

Timecounter math

The delta-count operation is straightforward subtraction, but we need to logically AND the result with a bit-mask with the same number (or less) bits as the counter implements, to prevent higher order bits from getting set when the counter rolls over:

$$\Delta Count = (Count_{now} - Count_{ref})\ BITAND\ mask$$

The scaling step is straightforward.

$$T_{now} = \Delta Count \cdot R_{counter} + T_{ref}$$

The scaling factor $R_{counter}$ will be described below.

At regular intervals, scheduled by `hardclock()`, a housekeeping routine is run which does the following:

A timestamp with associated hardware counter reading is elevated to be the new reference timecount:

$$\Delta Count = (Count_{now} - Count_{ref})\ BITAND\ mask$$

$$T_{now} = \Delta Count \cdot R_{counter}$$

$$Count_{ref} = Count_{now}$$

$$T_{ref} = T_{now}$$

If a new second has started, the NTP processing routines are called and the correction they return and the counters frequency is used to calculate the new scaling factor $R_{counter}$:

$$R_{counter} = \frac{2^{64}}{Freq_{counter}} \cdot R_{NTP}$$

Since we only have access to 64 bit arithmetic, dividing something into 2^{64} is a problem, so in the name of code clarity and efficiency, we sacrifice the low order bit and instead calculate:

$$R_{counter} = 2 \cdot \frac{2^{63}}{Freq_{counter}} \cdot R_{NTP}$$

The R_{NTP} correct factor arrives as the signed number of nanoseconds (with 32 bit binary fractions) to adjust per second. This quasi-decimal number is a bit of a square peg in our round binary hole, and a conversion factor is needed. Ideally we want to multiply this factor by:

$$\frac{2^{64}}{10^9 \cdot 2^{32}} = 4.294967296$$

This is not a nice number to work with. Fortunately, the precision of this correction is not critical, we are within an factor of a million of the 10^{-15} performance level of state of the art atomic clocks, so we can use an approximation on this term without anybody noticing.

Deciding which fraction to use as approximation needs to carefully consider any possible overflows that could happen. In this case the correction may be as large as ± 5000 PPM which leaves us room to multiply with about 850 in a multiply-before-divide setting. Unfortunately, there are no good fractions which multiply with less than 850 and at the same time divide by a power of two, which is desirable since it can be implemented as a binary shift instead of an expensive full division.

A divide-before-multiply approximation necessarily results in a loss of lower order bits, but in this case dividing by 512 and multiplying by 2199 gives a good approximation where the lower order bit loss is not a concern:

$$\frac{2199}{512} = 4.294921875$$

The resulting error is an systematic under compensation of 10.6PPM of the requested change, or $1.06 \cdot 10^{-14}$ per nanosecond of correction. This is perfectly acceptable.

Putting it all together, including the one bit we put on the alter for the Goddess of code clarity, the formula looks like this:

$$R_{counter} = 2 \cdot \frac{2^{63} + 2199 \cdot \frac{R_{NTP}}{1024}}{Freq_{counter}}$$

Presented here in slightly unorthodox format to show the component arithmetic operations as they are carried out in the code.

Frequency of the periodic update

The hardware counter should have a long enough period, ie, number of distinct counter values divided by frequency, to not roll over before our periodic update function has had a chance to update the reference timestamp data.

The periodic update function is called from `hardclock()` which runs at a rate which is controlled by the kernel parameter *HZ*.

By default HZ is 100 which means that only hardware with a period longer than 10 msec is usable. If HZ is configured higher than 1000, an internal divider is activated to keep the timecounter periodic update running no more often than 2000 times per second.

Let us take an example: At HZ=100 a 16 bit counter can run no faster than:

$$2^{16} \cdot 100Hz = 6.5536MHz$$

Similarly, if the counter runs at 10MHz, the minimum HZ is

$$\frac{10MHz}{2^{16}} = 152.6Hz$$

Some amount of margin is of course always advisable, and a factor two is considered prudent.

Locking, lack of ...

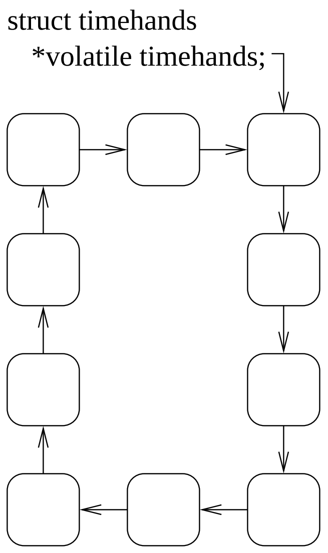
Provided our hardware can be read atomically, that our arithmetic has enough bits to not roll over and that our clock frequency is perfectly, or at least sufficiently, stable, we could avoid the periodic update function, and consequently disregard the entire issue of locking. We are seldom that lucky in practice.

The straightforward way of dealing with meta data updates is to put a lock of some kind on the data and grab hold of that before doing anything. This would however be a very heavy-handed approach. First of all, the updates are infrequent compared to simple references, second it is not important which particular state of meta data a consumer gets hold of, as long as it is consistent: as long as the $Count_{ref}$ and T_{ref} are a matching pair, and not old enough to cause an ambiguity with hardware counter rollover, a valid timestamp can be derived from them.

A pseudo-stable-storage with generation count method has been chosen instead. A ring of ten ‘‘timehands’’ data structures are used to hold the state of the timecounter system, the periodic update function updates the next structure with the new reference data and scaling factor and makes it the current timehands.

The beauty of this arrangement lies in the fact that even though a particular ‘‘timehands’’ data structure has been bumped from being the ‘‘currents state’’ by its successor, it still contains valid data for some amount of time into the future.

Therefore, a process which has started the timestamping process but suffered an interrupt which resulted in the above periodic processing can continue unaware of this afterwards and not suffer corruption or miscalculation even though it holds no locks on the shared meta-data.



This scheme has an inherent risk that a process may be de-scheduled for so long time that it will not manage to complete the timestamping process before the entire ring of timehands have been recycled. This case is covered by each timehand having a private generation number which is temporarily set to zero during the periodic processing, to mark inconsistent data, and incremented to one more than the previous value when the update has finished and the timehands is again consistent.

The timestamping code will grab a copy of this generation number and compare this copy to the generation in the timehands after completion and if they differ it will restart the timestamping calculation.

```
do {
    th = timehands;
    gen = th->th_generation;
    /* calculate timestamp */
} while (gen == 0 ||
   gen != th->th_generation);
```

Each hardware device supporting timecounting is represented by a small data structure called a timecounter, which documents the frequency, the number of bits implemented by the counter and a method function to read the counter.

Part of the state in the timehands structure is a pointer to the relevant timecounter structure, this makes it possible to change to a one piece of hardware to another "on the fly" by updating the current timehands pointer in a manner similar to the periodic update function.

In practice this can be done with sysctl(8):

```
sysctl kern.timecounter.hardware=TSC
```

at any time while the system is running.

Suitable hardware

A closer look on "suitable hardware" is warranted at this point. It is obvious from the above description that the ideal hardware for timecounting is a wide binary counter running at a constant high frequency and atomically readable by all CPUs in the system with a fast instruction(-sequence).

When looking at the hardware support on the PC platform, one is somewhat tempted to sigh deeply and mutter "so much for theory", because none of the above parameters seems to have been on the drawing board together yet.

All IBM PC derivatives contain a device more or less compatible with the venerable Intel i8254 chip. This device contains 3 counters of 16 bits each, one of which is wired so it can interrupt the CPU when the programmable terminal count is reached.

The problem with this device is that it only has 8bit bus-width, so reading a 16 bit timestamp takes 3 I/O operations: one to latch the count in an internal register, and two to read the high and low parts of that register respectively.

Obviously, on multi-CPU systems this cannot be done without some kind of locking mechanism preventing the other CPUs from trying to do the same thing at the same time.

Less obviously we find it is even worse than that: Since a low priority kernel thread might be reading a timestamp when an interrupt comes in, and since the interrupt thread might attempt to generate a timestamp also, we need to totally block interrupts out while doing those three I/O instructions.

And just to make life even more complicated, FreeBSD uses the same counter to provide the periodic interrupts which schedule the `hardclock()` routine, so in addition the code has to deal with the fact that the counter does not count down from a power of two and that an interrupt is generated right after the reloading of the counter when it reaches zero.

Ohh, and did I mention that the interrupt rate for hardclock() will be set to a higher frequency if profiling is active ? [6]

It hopefully doesn't ever get more complicated than that, but it shows, in its own bizarre and twisted way, just how little help the timecounter code needs from the actual hardware.

The next kind of hardware support to materialise was the "CPU clock counter" called "TSC" in

[6] I will not even mention the fact that it can be set also to ridiculous high frequencies in order to be able to use the binary driven "beep" speaker in the PC in a PCM fashion to output "real sounds".

official data-sheets. This is basically a on-CPU counter, which counts at the rate of the CPU clock.

Unfortunately, the electrical power needed to run a CPU is pretty precisely proportional with the clock frequency for the prevailing CMOS chip technology, so the advent of computers powered by batteries prompted technologies like APM, ACPI, SpeedStep and others which varies or throttles the CPU clock to match computing demand in order to minimise the power consumption [7].

Another wiggle for the TSC is that it is not usable on multi-CPU systems because the counter is implemented inside the CPU and not readable from other CPUs in the system.

The counters on different CPUs are not guaranteed to run syntonously (ie: show the same count at the same time). For some architectures like the DEC/alpha architecture they do not even run synchronously (ie: at the same rate) because the CPU clock frequency is generated by a small SAW device on the chip which is very sensitive to temperature changes.

The ACPI specification finally brings some light: it postulates the existence of a 24 or 32 bit counter running at a standardised constant frequency and specifically notes that this is intended to be used for timekeeping.

The frequency chosen, 3.5795454... MHz[8]
is not quite as high as one could have wished for, but it is certainly a big improvement over the i8254 hardware in terms of access path.

But trust it to Murphys Law: The majority of implementations so far have failed to provide latching suitable to avoid meta-stability problems, and several readings from the counter is necessary to get a reliable timestamp. In difference from the i8254 mentioned above, we do not need to any locking while doing so, since each individual read is atomic.

An initialization routine tries to test if the ACPI counter is properly latched by examining the width of a histogram over read delta-values.

[7]This technology also found ways into stationary computers from two different vectors. The first vector was technical: Cheaper cooling solutions can be used for the CPU if they are employed resulting in cheaper commodity hardware. The second vector was political: For reasons beyond reason, energy conservation became an issue with personal computers, despite the fact that practically north American households contains 4 to 5 household items which through inefficient designs waste more power than a personal computer use.

[8]The reason for this odd-ball frequency has to be sought in the ghastly colours offered by the original IBM PC Color Graphics Adapter: It delivered NTSC format output and therefore introduced the NTSC colour sync frequency into personal computers.

Other architectures are similarly equipped with means for timekeeping, but generally more carefully thought out compared to the haphazard developments of the IBM PC architecture.

One final important wiggle of all this, is that it may not be possible to determine which piece of hardware is best suited for clock use until well into or even after the bootstrap process.

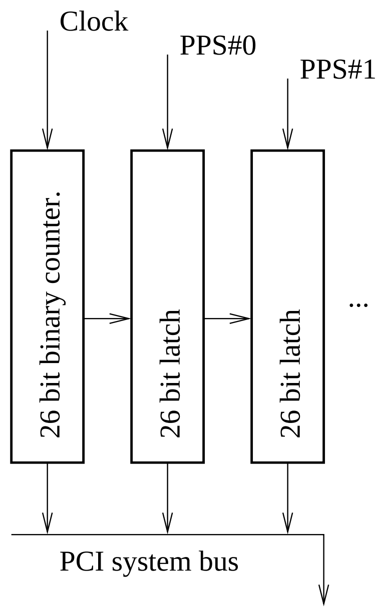
One example of this is the Loran-C receiver designed by Prof. Dave Mills [MILLS1992] which is unsuitable as timecounter until the daemon program which implements the software-half of the receiver has properly initialised and locked onto a Loran-C signal.

Ideal timecounter hardware

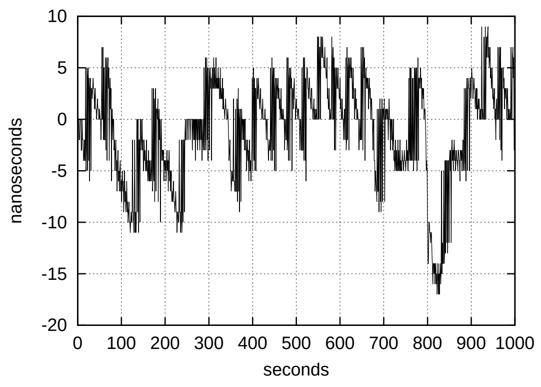
As proof of concept, a sort of an existentialist protest against the sorry state describe above, the author undertook a project to prove that it is possible to do better than that, since none of the standard hardware offered a way to fully validate the timecounter design.

Using a COTS product, ‘‘HOT1’’, from Virtual Computers Corporation [VCC2002] containing a FPGA chip on a PCI form factor card, a 26 bit timecounter running at 100MHz was successfully implemented.

In order to show that timestamping does not necessarily have to be done using unpredictable and uncalibratable interrupts, an array of latches were implemented as well, which allow up to 10 external signals to latch the reading of the counter when an external PPS signal transitions from logic high to logic low or vice versa.

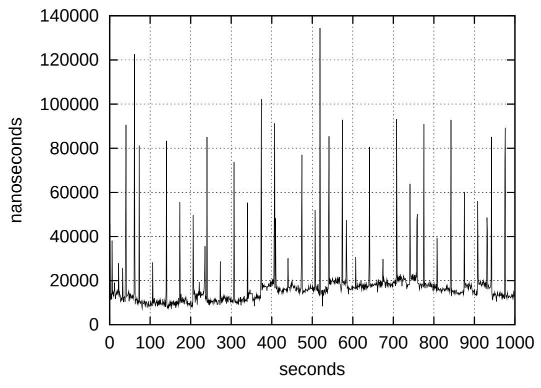


Using this setup, an standard 133 MHz Pentium based PC is able to timestamp the PPS output of the Motorola UT+ GPS receiver with a precision of ± 10 nanoseconds ± one count which in practice averages out to roughly ± 15 nanoseconds[9]:



It shold be noted that the author is no hardware wizard and a number of issues in the implementation results in less than ideal noise performance.

Now compare this to ''ideal'' timecounter to the normal setup where the PPS signal is used to trigger an interrupt via the DCD pin on a serial port, and the interrupt handler calls `nanotime()` to timestamp the external event [10]:



It is painfully obvious that the interrupt latency is the dominant noise factor in PPS timestamping in the second case. The asymetric distribution of the noise in the second plot also more or less entirely invalidates the design assumption in the NTP PLL/FLL kernel code that timestamps are dominated by gaussian noise with few spikes.

[9]The reason the plot does not show a very distinct 10 nanosecond quantization is that the GPS receiver produces the PPS signal from a clock with a roughly 55 nanosecond period and then predicts in the serial data stream how many nanoseconds this will be offset from the ideal time. This plot shows the timestamps corrected for this ''negative sawtooth correction''.

[10]In both cases, the computers clock frequency controlled with a Rubidium Frequency standard. The average quality of crystals used for computers would totally obscure the curves due to their temperature coefficient.

Status and availability

The timecounter code has been developed and used in FreeBSD for a number of years and has now reached maturity. The source-code is located almost entirely in the kernel source file kern_tc.c, with a few necessary adaptations in code which interfaces to it, primarily the NTP PLL/FLL code.

The code runs on all FreeBSD platforms including i386, alpha, PC98, sparc64, ia64 and s/390 and contains no wordsize or endianess issues not specifically handled in the sourcecode.

The timecounter implementation is distributed under the ''BSD'' open source license or the even more free ''Beer-ware'' license.

While the ability to accurately model and compensate for inaccuracies typical of atomic frequency standards are not catering to the larger userbase, but this ability and precision of the code guarntees solid support for the widespread deployment of NTP as a time synchronization protocol, without rounding or accumulative errors.

Adding support for new hardware and platforms have been done several times by other developers without any input from the author, so this particular aspect of timecounters design seems to work very well.

Future work

At this point in time, no specific plans exist for further development of the timecounters code.

Various micro-optimizations, mostly to compensate for inadequate compiler optimization could be contemplated, but the author resists these on the basis that they significantly decrease the readability of the source code.

Acknowledgements

The author would like to thank:

Bruce Evans for his invaluable assistance in taming the evil i8254 timecounter, as well as the enthusiastic resistance he has provided throughout.

Professor Dave Mills of University of Delaware for his work on NTP, for lending out the neglected twin Loran-C receiver and for picking up the glove when timecounters made it clear that the old ''microkernel'' NTP timekeeping code were not up to snuff [MILLS2000B].

Tom Van Baak for helping out, despite the best efforts of the National Danish Posts center for Customs and Dues to prevent it.

Corby Dawson for helping with the care and feeding for caesium standards.

The staff at the NELS Loran-C control station in Bø, Norway for providing information

about step-changes.

The staff at NELS Loran-C station Eiðe, Faeroe Islands for permission to tour their installation.

The FreeBSD users for putting up with ‘‘micro uptime went backwards’’.

References

[AG2002] Published specifications for Agilent model 5071A Primary Frequency Standard on http://www.agilent.com

[DMK2001] "Accuracy Evaluation of a Cesium Fountain Primary Frequency Standard at NIST." D. M. Meekhof, S. R. Jefferts, M. Stephanovic, and T. E. Parker IEEE Transactions on instrumentation and measurement, VOL. 50, NO. 2, APRIL 2001.

[PHK2001] "Monitoring Natural Gas Usage" Poul-Henning Kamp http://phk.freebsd.dk/Gasdims/

[MILLS1992] "A computer-controlled LORAN-C receiver for precision timekeeping." Mills, D.L. Electrical Engineering Department Report 92-3-1, University of Delaware, March 1992, 63 pp.

[MILLS2000A] Levine, J., and D. Mills. "Using the Network Time Protocol to transmit International Atomic Time (TAI)". Proc. Precision Time and Time Interval (PTTI) Applications and Planning Meeting (Reston VA, November 2000), 431-439.

[MILLS2000B] "The nanokernel." Mills, D.L., and P.-H. Kamp. Proc. Precision Time and Time Interval (PTTI) Applications and Planning Meeting (Reston VA, November 2000), 423-430.

[RGO2002] For an introduction to Harrison and his clocks, see for instance
http://www.rog.nmm.ac.uk/museum/harrison/
or for a more detailed and possibly better researched account: Dava Sobels excellent book, "Longitude: The True Story of a Lone Genius Who Solved the Greatest Scientific Problem of His Time" Penguin USA (Paper); ISBN: 0140258795.

[SAGE] This ‘‘gee-wiz’’ kind of article in Dr. Jobbs Journal is a goot place to start:
http://www.ddj.com/documents/s=1493/ddj0001hc/0085a.htm

[VCC2002] Please consult Virtual Computer Corporations homepage:
http://www.vcc.com