# PEP 418 -- Add monotonic time, performance counter, and process time functions

| | |
|---|---|
| **PEP:** | 418 |
| **Title:** | Add monotonic time, performance counter, and process time functions |
| **Author:** | Cameron Simpson <cs at cskk.id.au>, Jim Jewett <jimjjewett at gmail.com>, Stephen J. Turnbull <stephen at xemacs.org>, Victor Stinner <vstinner at python.org> |
| **Status:** | Final |
| **Type:** | Standards Track |
| **Created:** | 26-March-2012 |
| **Python-Version:** | 3.3 |

Contents


- Abstract (#abstract)
- Rationale (#rationale)
- Python functions (#python-functions)
  - New Functions (#new-functions)
    - time.get_clock_info(name) (#time-get-clock-info-name)
    - time.monotonic() (#time-monotonic)
    - time.perf_counter() (#time-perf-counter)
    - time.process_time() (#time-process-time)
  - Existing Functions (#existing-functions)
    - time.time() (#time-time)
    - time.sleep() (#time-sleep)
  - Deprecated Function (#deprecated-function)
    - time.clock() (#time-clock)
- Alternatives: API design (#alternatives-api-design)
  - Other names for time.monotonic() (#other-names-for-time-monotonic)
  - Other names for time.perf_counter() (#other-names-for-time-perf-counter)

- Only expose operating system clocks (#only-expose-operating-system-clocks)
- time.monotonic(): Fallback to system time (#time-monotonic-fallback-to-system-time)
  - One function with a flag: time.monotonic(fallback=True) (#one-function-with-a-flag-time-monotonic-fallback-true)
  - One time.monotonic() function, no flag (#one-time-monotonic-function-no-flag)
- Choosing the clock from a list of constraints (#choosing-the-clock-from-a-list-of-constraints)
- Working around operating system bugs? (#working-around-operating-system-bugs)

- Glossary (#glossary)
- Hardware clocks (#hardware-clocks)
  - List of hardware clocks (#list-of-hardware-clocks)
  - Linux clocksource (#linux-clocksource)
  - FreeBSD timecounter (#freebsd-timecounter)
  - Performance (#performance)
- NTP adjustment (#ntp-adjustment)
- Operating system time functions (#operating-system-time-functions)
  - Monotonic Clocks (#monotonic-clocks)
    - mach_absolute_time (#mach-absolute-time)
    - CLOCK_MONOTONIC, CLOCK_MONOTONIC_RAW, CLOCK_BOOTTIME (#clock-monotonic-clock-monotonic-raw-clock-boottime)
    - Windows: QueryPerformanceCounter (#windows-queryperformancecounter)
    - Windows: GetTickCount(), GetTickCount64() (#windows-gettickcount-gettickcount64)
    - Windows: timeGetTime (#windows-timegettime)
    - Solaris: CLOCK_HIGHRES (#solaris-clock-highres)
    - Solaris: gethrtime (#solaris-gethrtime)
  - System Time (#system-time)
    - Windows: GetSystemTimeAsFileTime (#windows-getsystemtimeasfiletime)
    - System time on UNIX (#system-time-on-unix)
  - Process Time (#process-time)
    - Functions (#functions)
  - Thread Time (#thread-time)
    - Functions (#id9)
  - Windows: QueryUnbiasedInterruptTime (#windows-queryunbiasedinterrupttime)
  - Sleep (#sleep)
    - Functions (#id10)
    - clock_nanosleep (#clock-nanosleep)
    - select() (#select)
    - Other functions (#other-functions)
- System Standby (#system-standby)
- Footnotes (#footnotes)
- Links (#links)

- Acceptance (#acceptance)
- References (#references)
- Copyright (#copyright)


## Abstract (#id18)

This PEP proposes to add `time.get_clock_info(name)`, `time.monotonic()`, `time.perf_counter()` and `time.process_time()` functions to Python 3.3.

## Rationale (#id19)

If a program uses the system time to schedule events or to implement a timeout, it may fail to run events at the right moment or stop the timeout too early or too late when the system time is changed manually or adjusted automatically by NTP. A monotonic clock should be used instead to not be affected by system time updates: `time.monotonic()`.

To measure the performance of a function, `time.clock()` can be used but it is very different on Windows and on Unix. On Windows, `time.clock()` includes time elapsed during sleep, whereas it does not on Unix. `time.clock()` resolution is very good on Windows, but very bad on Unix. The new `time.perf_counter()` function should be used instead to always get the most precise performance counter with a portable behaviour (ex: include time spend during sleep).

Until now, Python did not provide directly a portable function to measure CPU time. `time.clock()` can be used on Unix, but it has bad resolution. `resource.getrusage()` or `os.times()` can also be used on Unix, but they require to compute the sum of time spent in kernel space and user space. The new `time.process_time()` function acts as a portable counter that always measures CPU time (excluding time elapsed during sleep) and has the best available resolution.

Each operating system implements clocks and performance counters differently, and it is useful to know exactly which function is used and some properties of the clock like its resolution. The new `time.get_clock_info()` function gives access to all available information about each Python time function.

New functions:

- `time.monotonic()`: timeout and scheduling, not affected by system clock updates
- `time.perf_counter()`: benchmarking, most precise clock for short period
- `time.process_time()`: profiling, CPU time of the process

Users of new functions:

- time.monotonic(): concurrent.futures, multiprocessing, queue, subprocess, telnet and threading modules to implement timeout
- time.perf_counter(): trace and timeit modules, pybench program
- time.process_time(): profile module
- time.get_clock_info(): pybench program to display information about the timer like the resolution

The `time.clock()` function is deprecated because it is not portable: it behaves differently depending on the operating system. `time.perf_counter()` or `time.process_time()` should be used instead, depending on your requirements. `time.clock()` is marked as deprecated but is not planned for removal.

Limitations:

- The behaviour of clocks after a system suspend is not defined in the documentation of new functions. The behaviour depends on the operating system: see the Monotonic Clocks (#monotonic-clocks) section below. Some recent operating systems provide two clocks, one including time elapsed during system suspend, one not including this time. Most operating systems only provide one kind of clock.
- time.monotonic() and time.perf_counter() may or may not be adjusted. For example, `CLOCK_MONOTONIC` is slewed on Linux, whereas `GetTickCount()` is not adjusted on Windows. `time.get_clock_info('monotonic')['adjustable']` can be used to check if the monotonic clock is adjustable or not.
- No time.thread_time() function is proposed by this PEP because it is not needed by Python standard library nor a common asked feature. Such function would only be available on Windows and Linux. On Linux, it is possible to use `time.clock_gettime(CLOCK_THREAD_CPUTIME_ID)`. On Windows, ctypes or another module can be used to call the `GetThreadTimes()` function.

# Python functions (#id20)

## New Functions (#id21)

### time.get_clock_info(name) (#id22)

Get information on the specified clock. Supported clock names:

- `"clock"`: `time.clock()`
- `"monotonic"`: `time.monotonic()`
- `"perf_counter"`: `time.perf_counter()`
- `"process_time"`: `time.process_time()`
- `"time"`: `time.time()`

Return a `time.clock_info` object which has the following attributes:

- `implementation` (str): name of the underlying operating system function. Examples: `"QueryPerformanceCounter()"`, `"clock_gettime(CLOCK_REALTIME)"`.
- `monotonic` (bool): True if the clock cannot go backward.
- `adjustable` (bool): `True` if the clock can be changed automatically (e.g. by a NTP daemon) or manually by the system administrator, `False` otherwise
- `resolution` (float): resolution in seconds of the clock.

### time.monotonic() (#id23)

Monotonic clock, i.e. cannot go backward. It is not affected by system clock updates. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid and is a number of seconds.

On Windows versions older than Vista, `time.monotonic()` detects `GetTickCount()` integer overflow (32 bits, roll-over after 49.7 days). It increases an internal epoch (reference time by) $2^{32}$ each time that an overflow is detected. The epoch is stored in the process-local state and so the value of `time.monotonic()` may be different in two Python processes running for more than 49 days. On more recent versions

of Windows and on other operating systems, `time.monotonic()` is system-wide.

Availability: Windows, Mac OS X, Linux, FreeBSD, OpenBSD, Solaris. Not available on GNU/Hurd.

Pseudo-code [2] (#pseudo):

```
if os.name == 'nt':
    # GetTickCount64() requires Windows Vista, Server 2008 or later
    if hasattr(_time, 'GetTickCount64'):
        def monotonic():
            return _time.GetTickCount64() * 1e-3
    else:
        def monotonic():
            ticks = _time.GetTickCount()
            if ticks < monotonic.last:
                # Integer overflow detected
                monotonic.delta += 2**32
            monotonic.last = ticks
            return (ticks + monotonic.delta) * 1e-3
        monotonic.last = 0
        monotonic.delta = 0

elif sys.platform == 'darwin':
    def monotonic():
        if monotonic.factor is None:
            factor = _time.mach_timebase_info()
            monotonic.factor = timebase[0] / timebase[1] * 1e-9
        return _time.mach_absolute_time() * monotonic.factor
    monotonic.factor = None

elif hasattr(time, "clock_gettime") and hasattr(time, "CLOCK_HIGHRES"):
    def monotonic():
        return time.clock_gettime(time.CLOCK_HIGHRES)

elif hasattr(time, "clock_gettime") and hasattr(time, "CLOCK_MONOTONIC"):
    def monotonic():
        return time.clock_gettime(time.CLOCK_MONOTONIC)
```

On Windows, `QueryPerformanceCounter()` is not used even though it has a better resolution than `GetTickCount()`. It is not reliable and has too many issues.

## time.perf_counter() (#id24)

Performance counter with the highest available resolution to measure a short duration. It does include time elapsed during sleep and is system-wide. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid and is a number of seconds.

It is available on all platforms.

Pseudo-code:

```
if os.name == 'nt':
    def _win_perf_counter():
        if _win_perf_counter.frequency is None:
            _win_perf_counter.frequency = _time.QueryPerformanceFrequency()
        return _time.QueryPerformanceCounter() / _win_perf_counter.frequency
    _win_perf_counter.frequency = None

def perf_counter():
    if perf_counter.use_performance_counter:
        try:
            return _win_perf_counter()
        except OSError:
            # QueryPerformanceFrequency() fails if the installed
            # hardware does not support a high-resolution performance
            # counter
            perf_counter.use_performance_counter = False
    if perf_counter.use_monotonic:
        # The monotonic clock is preferred over the system time
        try:
            return time.monotonic()
        except OSError:
            perf_counter.use_monotonic = False
    return time.time()
perf_counter.use_performance_counter = (os.name == 'nt')
perf_counter.use_monotonic = hasattr(time, 'monotonic')
```

## time.process_time() (#id25)

Sum of the system and user CPU time of the current process. It does not include time elapsed during sleep. It is process-wide by definition. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.

It is available on all platforms.

Pseudo-code [2] (#pseudo):

```
if os.name == 'nt':
    def process_time():
        handle = _time.GetCurrentProcess()
        process_times = _time.GetProcessTimes(handle)
        return (process_times['UserTime'] + process_times['KernelTime']) * 1e-7
else:
    try:
        import resource
    except ImportError:
        has_resource = False
    else:
        has_resource = True

    def process_time():
        if process_time.clock_id is not None:
            try:
                return time.clock_gettime(process_time.clock_id)
            except OSError:
                process_time.clock_id = None
        if process_time.use_getrusage:
            try:
                usage = resource.getrusage(resource.RUSAGE_SELF)
                return usage[0] + usage[1]
            except OSError:
                process_time.use_getrusage = False
        if process_time.use_times:
            try:
                times = _time.times()
                cpu_time = times.tms_utime + times.tms_stime
                return cpu_time / process_time.ticks_per_seconds
            except OSError:
                process_time.use_getrusage = False
        return _time.clock()
    if (hasattr(time, 'clock_gettime')
        and hasattr(time, 'CLOCK_PROF')):
        process_time.clock_id = time.CLOCK_PROF
    elif (hasattr(time, 'clock_gettime')
          and hasattr(time, 'CLOCK_PROCESS_CPUTIME_ID')):
        process_time.clock_id = time.CLOCK_PROCESS_CPUTIME_ID
    else:
        process_time.clock_id = None
```

```
    process_time.use_getrusage = has_resource
    process_time.use_times = hasattr(_time, 'times')
    if process_time.use_times:
        # sysconf("SC_CLK_TCK"), or the HZ constant, or 60
        process_time.ticks_per_seconds = _times.ticks_per_seconds
```

## Existing Functions (#id26)

### time.time() (#id27)

The system time which is usually the civil time. It is system-wide by definition. It can be set manually by the system administrator or automatically by a NTP daemon.

It is available on all platforms and cannot fail.

Pseudo-code [2] (#pseudo):

```
if os.name == "nt":
    def time():
        return _time.GetSystemTimeAsFileTime()
else:
    def time():
        if hasattr(time, "clock_gettime"):
            try:
                return time.clock_gettime(time.CLOCK_REALTIME)
            except OSError:
                # CLOCK_REALTIME is not supported (unlikely)
                pass
        if hasattr(_time, "gettimeofday"):
            try:
                return _time.gettimeofday()
            except OSError:
                # gettimeofday() should not fail
                pass
        if hasattr(_time, "ftime"):
            return _time.ftime()
        else:
            return _time.time()
```

### time.sleep() (#id28)

Suspend execution for the given number of seconds. The actual suspension time may be less than that requested because any caught signal will terminate the `time.sleep()` following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount because of the scheduling of other activity in the system.

Pseudo-code [2] (#pseudo):

```
try:
    import select
except ImportError:
    has_select = False
else:
    has_select = hasattr(select, "select")

if has_select:
    def sleep(seconds):
        return select.select([], [], [], seconds)

elif hasattr(_time, "delay"):
    def sleep(seconds):
        milliseconds = int(seconds * 1000)
        _time.delay(milliseconds)

elif os.name == "nt":
    def sleep(seconds):
        milliseconds = int(seconds * 1000)
        win32api.ResetEvent(hInterruptEvent);
        win32api.WaitForSingleObject(sleep.sigint_event, milliseconds)

    sleep.sigint_event = win32api.CreateEvent(NULL, TRUE, FALSE, FALSE)
    # SetEvent(sleep.sigint_event) will be called by the signal handler of SIGINT

elif os.name == "os2":
    def sleep(seconds):
        milliseconds = int(seconds * 1000)
        DosSleep(milliseconds)

else:
    def sleep(seconds):
        seconds = int(seconds)
        _time.sleep(seconds)
```

## Deprecated Function (#id29)

### time.clock() (#id30)

On Unix, return the current processor time as a floating point number expressed in seconds. It is process-wide by definition. The resolution, and in fact the very definition of the meaning of "processor time", depends on that of the C function of the same name, but in any case, this is the function to use for benchmarking Python or timing algorithms.

On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number, based on the Win32 function `QueryPerformanceCounter()`. The resolution is typically better than one microsecond. It is system-wide.

Pseudo-code [2] (#pseudo):

```
if os.name == 'nt':
    def clock():
        try:
            return _win_perf_counter()
        except OSError:
            # QueryPerformanceFrequency() fails if the installed
            # hardware does not support a high-resolution performance
            # counter
            pass
        return _time.clock()
else:
    clock = _time.clock
```

# Alternatives: API design (#id31)

## Other names for time.monotonic() (#id32)

- time.counter()
- time.metronomic()
- time.seconds()
- time.steady(): "steady" is ambiguous: it means different things to different people. For example, on Linux, CLOCK_MONOTONIC is adjusted. If we uses the real time as the reference clock, we may say that CLOCK_MONOTONIC is steady. But CLOCK_MONOTONIC gets suspended on system suspend, whereas real time includes any time spent in suspend.
- time.timeout_clock()
- time.wallclock(): time.monotonic() is not the system time aka the "wall clock", but a monotonic clock with an unspecified starting point.

The name "time.try_monotonic()" was also proposed for an older version of time.monotonic() which would fall back to the system time when no monotonic clock was available.

## Other names for time.perf_counter() (#id33)

- time.high_precision()
- time.highres()
- time.hires()
- time.performance_counter()
- time.timer()

## Only expose operating system clocks (#id34)

To not have to define high-level clocks, which is a difficult task, a simpler approach is to only expose operating system clocks. time.clock_gettime() and related clock identifiers were already added to Python 3.3 for example.

## time.monotonic(): Fallback to system time (#id35)

If no monotonic clock is available, time.monotonic() falls back to the system time.

Issues:

- It is hard to define such a function correctly in the documentation: is it monotonic? Is it steady? Is it adjusted?
- Some users want to decide what to do when no monotonic clock is available: use another clock, display an error, or do something else.

Different APIs were proposed to define such function.

### One function with a flag: time.monotonic(fallback=True) (#id36)

- time.monotonic(fallback=True) falls back to the system time if no monotonic clock is available or if the monotonic clock failed.
- time.monotonic(fallback=False) raises OSError if monotonic clock fails and NotImplementedError if the system does not provide a monotonic clock

A keyword argument that gets passed as a constant in the caller is usually poor API.

Raising NotImplementedError for a function is something uncommon in Python and should be avoided.

### One time.monotonic() function, no flag (#id37)

time.monotonic() returns (time: float, is_monotonic: bool).

An alternative is to use a function attribute: time.monotonic.is_monotonic. The attribute value would be None before the first call to time.monotonic().

## Choosing the clock from a list of constraints (#id38)

The PEP as proposed offers a few new clocks, but their guarantees are deliberately loose in order to offer useful clocks on different platforms. This inherently embeds policy in the calls, and the caller must thus choose a policy.

The "choose a clock" approach suggests an additional API to let callers implement their own policy if necessary by making most platform clocks available and letting the caller pick amongst them. The PEP's suggested clocks are still expected to be available for the common simple use cases.

To do this two facilities are needed: an enumeration of clocks, and metadata on the clocks to enable the user to evaluate their suitability.

The primary interface is a function make simple choices easy: the caller can use `time.get_clock(*flags)` with some combination of flags. This includes at least:

- time.MONOTONIC: clock cannot go backward
- time.STEADY: clock rate is steady
- time.ADJUSTED: clock may be adjusted, for example by NTP
- time.HIGHRES: clock with the highest resolution

It returns a clock object with a .now() method returning the current time. The clock object is annotated with metadata describing the clock feature set; its .flags field will contain at least all the requested flags.

time.get_clock() returns None if no matching clock is found and so calls can be chained using the or operator. Example of a simple policy decision:

```
T = get_clock(MONOTONIC) or get_clock(STEADY) or get_clock()
t = T.now()
```

The available clocks always at least include a wrapper for `time.time()`, so a final call with no flags can always be used to obtain a working clock.

Examples of flags of system clocks:

- QueryPerformanceCounter: MONOTONIC | HIGHRES
- GetTickCount: MONOTONIC | STEADY
- CLOCK_MONOTONIC: MONOTONIC | STEADY (or only MONOTONIC on Linux)
- CLOCK_MONOTONIC_RAW: MONOTONIC | STEADY
- gettimeofday(): (no flag)

The clock objects contain other metadata including the clock flags with additional feature flags above those listed above, the name of the underlying OS facility, and clock precisions.

`time.get_clock()` still chooses a single clock; an enumeration facility is also required. The most obvious method is to offer `time.get_clocks()` with the same signature as `time.get_clock()`, but returning a sequence of all clocks matching the requested flags. Requesting no flags would thus enumerate all available clocks, allowing the caller to make an arbitrary choice amongst them based on their metadata.

Example partial implementation: clockutils.py (http://hg.python.org/peps/file/tip/pep-0418/clockutils.py).

## Working around operating system bugs? (#id39)

Should Python ensure that a monotonic clock is truly monotonic by computing the maximum with the clock value and the previous value?

Since it's relatively straightforward to cache the last value returned using a static variable, it might be interesting to use this to make sure that the values returned are indeed monotonic.

- Virtual machines provide less reliable clocks.
- QueryPerformanceCounter() has known bugs (only one is not fixed yet)

Python may only work around a specific known operating system bug: KB274323 (http://support.microsoft.com/?id=274323) [4] (#id15) contains a code example to workaround the bug (use GetTickCount() to detect QueryPerformanceCounter() leap).

Issues with "correcting" non-monotonicities:

- if the clock is accidentally set forward by an hour and then back again, you wouldn't have a useful clock for an hour
- the cache is not shared between processes so different processes wouldn't see the same clock value

# Glossary (#id40)

**Accuracy:** The amount of deviation of measurements by a given instrument from true values. See also Accuracy and precision (http://en.wikipedia.org/wiki/Accuracy_and_precision). Inaccuracy in clocks may be caused by lack of precision, drift, or an incorrect initial setting of the clock (e.g., timing of threads is inherently inaccurate because perfect synchronization in resetting counters is quite difficult).

**Adjusted:** Resetting a clock to the correct time. This may be done either with a <Step> or by <Slewing>.

**Civil Time:** Time of day; external to the system. 10:45:13am is a Civil time; 45 seconds is not. Provided by existing function `time.localtime()` and `time.gmtime()`. Not changed by this PEP.

**Clock:** An instrument for measuring time. Different clocks have different characteristics; for example, a clock with nanosecond <precision> may start to <drift> after a few minutes, while a less precise clock remained accurate for days. This PEP is primarily concerned with clocks which use a unit of seconds.

**Counter:** A clock which increments each time a certain event occurs. A counter is strictly monotonic, but not a monotonic clock. It can be used to generate a unique (and ordered) timestamp, but these timestamps cannot be mapped to <civil time>; tick creation may well be bursty, with several advances in the same millisecond followed by several days without any advance.

**CPU Time:** A measure of how much CPU effort has been spent on a certain task. CPU seconds are often normalized (so that a variable number can occur in the same actual second). CPU seconds can be important when profiling, but they do not map directly to user response time, nor are they directly comparable to (real time) seconds.

**Drift:** The accumulated error against "true" time, as defined externally to the system. Drift may be due to imprecision, or to a difference between the average rate at which clock time advances and that of real time.

**Epoch:** The reference point of a clock. For clocks providing <civil time>, this is often midnight as the day (and year) rolled over to January 1, 1970. For a <clock_monotonic> clock, the epoch may be undefined (represented as None).

**Latency:** Delay. By the time a clock call returns, the <real time> has advanced, possibly by more than the precision of the clock.

**Monotonic:** The characteristics expected of a monotonic clock in practice. Moving in at most one direction; for clocks, that direction is forward. The <clock> should also be <steady>, and should be convertible to a unit of seconds. The tradeoffs often include lack of a defined <epoch> or mapping to <Civil Time>.

**Precision:** The amount of deviation among measurements of the same physical value by a single instrument. Imprecision in clocks may be caused by a fluctuation of the rate at which clock time advances relative to real time, including clock adjustment by slewing.

**Process Time:** Time elapsed since the process began. It is typically measured in <CPU time> rather than <real time>, and typically does not advance while the process is suspended.

**Real Time:** Time in the real world. This differs from <Civil time> in that it is not <adjusted>, but they should otherwise advance in lockstep. It is not related to the "real time" of "Real Time [Operating] Systems". It is sometimes called "wall clock time" to avoid that ambiguity; unfortunately, that introduces different ambiguities.

**Resolution:** The smallest difference between two physical values that results in a different measurement by a given instrument.

**Slew:** A slight change to a clock's speed, usually intended to correct <drift> with respect to an external authority.

**Stability:** Persistence of accuracy. A measure of expected <drift>.

**Steady:** A clock with high <stability> and relatively high <accuracy> and <precision>. In practice, it is often used to indicate a <clock_monotonic> clock, but places greater emphasis on the consistency of the duration between subsequent ticks.

**Step:** An instantaneous change in the represented time. Instead of speeding or slowing the clock (<slew>), a single offset is permanently added.

**System Time:** Time as represented by the Operating System.

**Thread Time:** Time elapsed since the thread began. It is typically measured in <CPU time> rather than <real time>, and typically does not advance while the thread is idle.

**Wallclock:** What the clock on the wall says. This is typically used as a synonym for <real time>; unfortunately, wall time is itself ambiguous.

# Hardware clocks (#id41)

## List of hardware clocks (#id42)

- HPET: A High Precision Event Timer (HPET) chip consists of a 64-bit up-counter (main counter) counting at least at 10 MHz and a set of up to 256 comparators (at least 3). Each HPET can have up to 32 timers. HPET can cause around 3 seconds of drift per day.
- TSC (Time Stamp Counter): Historically, the TSC increased with every internal processor clock cycle, but now the rate is usually constant (even if the processor changes frequency) and usually equals the maximum processor frequency. Multiple cores have different TSC values. Hibernation of system will reset TSC value. The RDTSC instruction can be used to read this counter. CPU frequency scaling for power saving.
- ACPI Power Management Timer: ACPI 24-bit timer with a frequency of 3.5 MHz (3,579,545 Hz).
- Cyclone: The Cyclone timer uses a 32-bit counter on IBM Extended X-Architecture (EXA) chipsets which include computers that use the IBM "Summit" series chipsets (ex: x440). This is available in IA32 and IA64 architectures.
- PIT (programmable interrupt timer): Intel 8253/8254 chipsets with a configurable frequency in range 18.2 Hz - 1.2 MHz. It uses a 16-bit counter.

- RTC (Real-time clock). Most RTCs use a crystal oscillator with a frequency of 32,768 Hz.

## Linux clocksource (#id43)

There were 4 implementations of the time in the Linux kernel: UTIME (1996), timer wheel (1997), HRT (2001) and hrtimers (2007). The latter is the result of the "high-res-timers" project started by George Anzinger in 2001, with contributions by Thomas Gleixner and Douglas Niehaus. The hrtimers implementation was merged into Linux 2.6.21, released in 2007.

hrtimers supports various clock sources. It sets a priority to each source to decide which one will be used. Linux supports the following clock sources:

- tsc
- hpet
- pit
- pmtmr: ACPI Power Management Timer
- cyclone

High-resolution timers are not supported on all hardware architectures. They are at least provided on x86/x86_64, ARM and PowerPC.

clock_getres() returns 1 nanosecond for `CLOCK_REALTIME` and `CLOCK_MONOTONIC` regardless of underlying clock source. Read Re: clock_getres() and real resolution (http://lkml.org/lkml/2012/2/9/100) from Thomas Gleixner (9 Feb 2012) for an explanation.

The `/sys/devices/system/clocksource/clocksource0` directory contains two useful files:

- `available_clocksource`: list of available clock sources
- `current_clocksource`: clock source currently used. It is possible to change the current clocksource by writing the name of a clocksource into this file.

`/proc/timer_list` contains the list of all hardware timers.

Read also the time(7) manual page (http://www.kernel.org/doc/man-pages/online/pages/man7/time.7.html): "overview of time and timers".

## FreeBSD timecounter (#id44)

kern.timecounter.choice lists available hardware clocks with their priority. The sysctl program can be used to change the timecounter. Example:

```
# dmesg | grep Timecounter
Timecounter "i8254" frequency 1193182 Hz quality 0
Timecounter "ACPI-safe" frequency 3579545 Hz quality 850
Timecounter "HPET" frequency 100000000 Hz quality 900
Timecounter "TSC" frequency 3411154800 Hz quality 800
Timecounters tick every 10.000 msec
# sysctl kern.timecounter.choice
kern.timecounter.choice: TSC(800) HPET(900) ACPI-safe(850) i8254(0) dummy(-1000000)
# sysctl kern.timecounter.hardware="ACPI-fast"
kern.timecounter.hardware: HPET -> ACPI-fast
```

Available clocks:

- "TSC": Time Stamp Counter of the processor
- "HPET": High Precision Event Timer
- "ACPI-fast": ACPI Power Management timer (fast mode)
- "ACPI-safe": ACPI Power Management timer (safe mode)
- "i8254": PIT with Intel 8254 chipset

The commit 222222 (http://svnweb.freebsd.org/base?view=revision&revision=222222) (May 2011) decreased ACPI-fast timecounter quality to 900 and increased HPET timecounter quality to 950: "HPET on modern platforms usually have better resolution and lower latency than ACPI timer".

Read Timecounters: Efficient and precise timekeeping in SMP kernels (http://phk.freebsd.dk/pubs/timecounter.pdf) by Poul-Henning Kamp (2002) for the FreeBSD Project.

## Performance (#id45)

Reading a hardware clock has a cost. The following table compares the performance of different hardware clocks on Linux 3.3 with Intel Core i7-2600 at 3.40GHz (8 cores). The bench_time.c (http://hg.python.org/peps/file/tip/pep-0418/bench_time.c) program was used to fill these tables.

| Function | TSC | ACPI PM | HPET |
|---|---|---|---|
| time() | 2 ns | 2 ns | 2 ns |
| CLOCK_REALTIME_COARSE | 10 ns | 10 ns | 10 ns |
| CLOCK_MONOTONIC_COARSE | 12 ns | 13 ns | 12 ns |
| CLOCK_THREAD_CPUTIME_ID | 134 ns | 135 ns | 135 ns |
| CLOCK_PROCESS_CPUTIME_ID | 127 ns | 129 ns | 129 ns |
| clock() | 146 ns | 146 ns | 143 ns |
| gettimeofday() | 23 ns | 726 ns | 637 ns |
| CLOCK_MONOTONIC_RAW | 31 ns | 716 ns | 607 ns |
| CLOCK_REALTIME | 27 ns | 707 ns | 629 ns |

| Function | TSC | ACPI PM | HPET |
|---|---|---|---|
| CLOCK_MONOTONIC | 27 ns | 723 ns | 635 ns |

FreeBSD 8.0 in kvm with hardware virtualization:

| Function | TSC | ACPI-Safe | HPET | i8254 |
|---|---|---|---|---|
| time() | 191 ns | 188 ns | 189 ns | 188 ns |
| CLOCK_SECOND | 187 ns | 184 ns | 187 ns | 183 ns |
| CLOCK_REALTIME_FAST | 189 ns | 180 ns | 187 ns | 190 ns |
| CLOCK_UPTIME_FAST | 191 ns | 185 ns | 186 ns | 196 ns |
| CLOCK_MONOTONIC_FAST | 188 ns | 187 ns | 188 ns | 189 ns |
| CLOCK_THREAD_CPUTIME_ID | 208 ns | 206 ns | 207 ns | 220 ns |
| CLOCK_VIRTUAL | 280 ns | 279 ns | 283 ns | 296 ns |
| CLOCK_PROF | 289 ns | 280 ns | 282 ns | 286 ns |
| clock() | 342 ns | 340 ns | 337 ns | 344 ns |
| CLOCK_UPTIME_PRECISE | 197 ns | 10380 ns | 4402 ns | 4097 ns |
| CLOCK_REALTIME | 196 ns | 10376 ns | 4337 ns | 4054 ns |
| CLOCK_MONOTONIC_PRECISE | 198 ns | 10493 ns | 4413 ns | 3958 ns |
| CLOCK_UPTIME | 197 ns | 10523 ns | 4458 ns | 4058 ns |
| gettimeofday() | 202 ns | 10524 ns | 4186 ns | 3962 ns |
| CLOCK_REALTIME_PRECISE | 197 ns | 10599 ns | 4394 ns | 4060 ns |
| CLOCK_MONOTONIC | 201 ns | 10766 ns | 4498 ns | 3943 ns |

Each function was called 100,000 times and CLOCK_MONOTONIC was used to get the time before and after. The benchmark was run 5 times, keeping the minimum time.

## NTP adjustment (#id46)

NTP has different methods to adjust a clock:

- "slewing": change the clock frequency to be slightly faster or slower (which is done with `adjtime()`). Since the slew rate is limited to 0.5 millisecond per second, each second of adjustment requires an amortization interval of 2000 seconds. Thus, an adjustment of many seconds can take hours or days to amortize.
- "stepping": jump by a large amount in a single discrete step (which is done with `settimeofday()`)

By default, the time is slewed if the offset is less than 128 ms, but stepped otherwise.

Slewing is generally desirable (i.e. we should use CLOCK_MONOTONIC, not CLOCK_MONOTONIC_RAW) if one wishes to measure "real" time (and not a time-like object like CPU cycles). This is because the clock on the other end of the NTP connection from you is probably better at keeping time: hopefully that thirty-five thousand dollars of Cesium timekeeping goodness is doing something better than your PC's $3 quartz crystal, after all.

Get more detail in the documentation of the NTP daemon (http://doc.ntp.org/4.1.2/ntpd.htm).

# Operating system time functions (#id47)

## Monotonic Clocks (#id48)

| Name | C Resolution | Adjusted | Include Sleep | Include Suspend |
|---|---|---|---|---|
| gethrtime() | 1 ns | No | Yes | Yes |
| CLOCK_HIGHRES | 1 ns | No | Yes | Yes |
| CLOCK_MONOTONIC | 1 ns | Slewed on Linux | Yes | No |
| CLOCK_MONOTONIC_COARSE | 1 ns | Slewed on Linux | Yes | No |
| CLOCK_MONOTONIC_RAW | 1 ns | No | Yes | No |
| CLOCK_BOOTTIME | 1 ns | ? | Yes | Yes |
| CLOCK_UPTIME | 1 ns | No | Yes | ? |
| mach_absolute_time() | 1 ns | No | Yes | No |
| QueryPerformanceCounter() | - | No | Yes | ? |
| GetTickCount[64]() | 1 ms | No | Yes | Yes |
| timeGetTime() | 1 ms | No | Yes | ? |

The "C Resolution" column is the resolution of the underlying C structure.

Examples of clock resolution on x86_64:

| Name | Operating system | OS Resolution | Python Resolution |
|---|---|---|---|
| QueryPerformanceCounter | Windows Seven | 10 ns | 10 ns |
| CLOCK_HIGHRES | SunOS 5.11 | 2 ns | 265 ns |
| CLOCK_MONOTONIC | Linux 3.0 | 1 ns | 322 ns |
| CLOCK_MONOTONIC_RAW | Linux 3.3 | 1 ns | 628 ns |
| CLOCK_BOOTTIME | Linux 3.3 | 1 ns | 628 ns |
| mach_absolute_time() | Mac OS 10.6 | 1 ns | 3 µs |
| CLOCK_MONOTONIC | FreeBSD 8.2 | 11 ns | 5 µs |
| CLOCK_MONOTONIC | OpenBSD 5.0 | 10 ms | 5 µs |
| CLOCK_UPTIME | FreeBSD 8.2 | 11 ns | 6 µs |
| CLOCK_MONOTONIC_COARSE | Linux 3.3 | 1 ms | 1 ms |
| CLOCK_MONOTONIC_COARSE | Linux 3.0 | 4 ms | 4 ms |
| GetTickCount64() | Windows Seven | 16 ms | 15 ms |

The "OS Resolution" is the resolution announced by the operating system. The "Python Resolution" is the smallest difference between two calls to the time function computed in Python using the clock_resolution.py (http://hg.python.org/peps/file/tip/pep-0418/clock_resolution.py) program.

## mach_absolute_time (#id49)

Mac OS X provides a monotonic clock: mach_absolute_time(). It is based on absolute elapsed time since system boot. It is not adjusted and cannot be set.

mach_timebase_info() gives a fraction to convert the clock value to a number of nanoseconds. See also the Technical Q&A QA1398 (https://developer.apple.com/library/mac/#qa/qa1398/).

mach_absolute_time() stops during a sleep on a PowerPC CPU, but not on an Intel CPU: Different behaviour of mach_absolute_time() on i386/ppc (http://lists.apple.com/archives/PerfOptimization-dev/2006/Jul/msg00024.html).

## CLOCK_MONOTONIC, CLOCK_MONOTONIC_RAW, CLOCK_BOOTTIME (#id50)

CLOCK_MONOTONIC and CLOCK_MONOTONIC_RAW represent monotonic time since some unspecified starting point. They cannot be set. The resolution can be read using `clock_getres()`.

Documentation: refer to the manual page of your operating system. Examples:

- FreeBSD clock_gettime() manual page (http://www.freebsd.org/cgi/man.cgi?query=clock_gettime)
- Linux clock_gettime() manual page (http://linux.die.net/man/3/clock_gettime)

CLOCK_MONOTONIC is available at least on the following operating systems:

- DragonFly BSD, FreeBSD >= 5.0, OpenBSD, NetBSD
- Linux
- Solaris

The following operating systems don't support CLOCK_MONOTONIC:

- GNU/Hurd (see open issues/ clock_gettime (http://www.gnu.org/software/hurd/open_issues/clock_gettime.html))
- Mac OS X
- Windows

On Linux, NTP may adjust the CLOCK_MONOTONIC rate (slewed), but it cannot jump backward.

CLOCK_MONOTONIC_RAW is specific to Linux. It is similar to CLOCK_MONOTONIC, but provides access to a raw hardware-based time that is not subject to NTP adjustments. CLOCK_MONOTONIC_RAW requires Linux 2.6.28 or later.

Linux 2.6.39 and glibc 2.14 introduces a new clock: CLOCK_BOOTTIME. CLOCK_BOOTTIME is identical to CLOCK_MONOTONIC, except that it also includes any time spent in suspend. Read also Waking systems from suspend (http://lwn.net/Articles/429925/) (March, 2011).

CLOCK_MONOTONIC stops while the machine is suspended.

Linux provides also CLOCK_MONOTONIC_COARSE since Linux 2.6.32. It is similar to CLOCK_MONOTONIC, less precise but faster.

`clock_gettime()` fails if the system does not support the specified clock, even if the standard C library supports it. For example, CLOCK_MONOTONIC_RAW requires a kernel version 2.6.28 or later.

## Windows: QueryPerformanceCounter (#id51)

High-resolution performance counter. It is monotonic. The frequency of the counter can be read using QueryPerformanceFrequency(). The resolution is 1 / QueryPerformanceFrequency().

It has a much higher resolution, but has lower long term precision than GetTickCount() and timeGetTime() clocks. For example, it will drift compared to the low precision clocks.

Documentation:

- MSDN: QueryPerformanceCounter() documentation (http://msdn.microsoft.com/en-us/library/windows/desktop/ms644904%28v=vs.85%29.aspx)
- MSDN: QueryPerformanceFrequency() documentation (http://msdn.microsoft.com/en-us/library/windows/desktop/ms644905%28v=vs.85%29.aspx)

Hardware clocks used by QueryPerformanceCounter:

- Windows XP: RDTSC instruction of Intel processors, the clock frequency is the frequency of the processor (between 200 MHz and 3 GHz, usually greater than 1 GHz nowadays).
- Windows 2000: ACPI power management timer, frequency = 3,549,545 Hz. It can be forced through the "/usepmtimer" flag in boot.ini.

QueryPerformanceFrequency() should only be called once: the frequency will not change while the system is running. It fails if the installed hardware does not support a high-resolution performance counter.

QueryPerformanceCounter() cannot be adjusted: SetSystemTimeAdjustment() (http://msdn.microsoft.com/en-us/library/windows/desktop/ms724943(v=vs.85).aspx) only adjusts the system time.

Bugs:

- The performance counter value may unexpectedly leap forward because of a hardware bug, see KB274323 (http://support.microsoft.com/?id=274323) [4] (#id15).
- On VirtualBox, QueryPerformanceCounter() does not increment the high part every time the low part overflows, see Monotonic timers (http://code-factor.blogspot.fr/2009/11/monotonic-timers.html) (2009).
- VirtualBox had a bug in its HPET virtualized device: QueryPerformanceCounter() did jump forward by approx. 42 seconds (issue #8707 (https://www.virtualbox.org/ticket/8707)).
- Windows XP had a bug (see KB896256 (http://support.microsoft.com/?id=896256) [3] (#id13)): on a multiprocessor computer, QueryPerformanceCounter() returned a different value for each processor. The bug was fixed in Windows XP SP2.
- Issues with processor with variable frequency: the frequency is changed depending on the workload to reduce memory consumption.
- Chromium don't use QueryPerformanceCounter() on Athlon X2 CPUs (model 15) because "QueryPerformanceCounter is unreliable" (see base/time_win.cc in Chromium source code)

## Windows: GetTickCount(), GetTickCount64() (#id52)

GetTickCount() and GetTickCount64() are monotonic, cannot fail and are not adjusted by SetSystemTimeAdjustment(). MSDN documentation: GetTickCount() (http://msdn.microsoft.com/en-us/library/windows/desktop/ms724408(v=vs.85).aspx), GetTickCount64() (http://msdn.microsoft.com/en-us/library/windows/desktop/ms724411(v=vs.85).aspx). The resolution can be read using GetSystemTimeAdjustment().

The elapsed time retrieved by GetTickCount() or GetTickCount64() includes time the system spends in sleep or hibernation.

GetTickCount64() was added to Windows Vista and Windows Server 2008.

It is possible to improve the precision using the undocumented NtSetTimerResolution() function (http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/Time/NtSetTimerResolution.html). There are applications using this undocumented function, example: Timer Resolution (http://www.lucashale.com/timer-resolution/).

WaitForSingleObject() uses the same timer as GetTickCount() with the same precision.

## Windows: timeGetTime (#id53)

The timeGetTime function retrieves the system time, in milliseconds. The system time is the time elapsed since Windows was started. Read the timeGetTime() documentation (http://msdn.microsoft.com/en-us/library/windows/desktop/dd757629(v=vs.85).aspx).

The return type of timeGetTime() is a 32-bit unsigned integer. As GetTickCount(), timeGetTime() rolls over after 2^32 milliseconds (49.7 days).

The elapsed time retrieved by timeGetTime() includes time the system spends in sleep.

The default precision of the timeGetTime function can be five milliseconds or more, depending on the machine.

timeBeginPeriod() can be used to increase the precision of timeGetTime() up to 1 millisecond, but it negatively affects power consumption. Calling timeBeginPeriod() also affects the granularity of some other timing calls, such as CreateWaitableTimer(), WaitForSingleObject() and Sleep().

> **Note:**
> timeGetTime() and timeBeginPeriod() are part the Windows multimedia library and so require to link the program against winmm or to dynamically load the library.

## Solaris: CLOCK_HIGHRES (#id54)

The Solaris OS has a CLOCK_HIGHRES timer that attempts to use an optimal hardware source, and may give close to nanosecond resolution. CLOCK_HIGHRES is the nonadjustable, high-resolution clock. For timers created with a clockid_t value of CLOCK_HIGHRES, the system will attempt to use an optimal hardware source.

The resolution of CLOCK_HIGHRES can be read using `clock_getres()`.

## Solaris: gethrtime (#id55)

The gethrtime() function returns the current high-resolution real time. Time is expressed as nanoseconds since some arbitrary time in the past; it is not correlated in any way to the time of day, and thus is not subject to resetting or drifting by way of adjtime() or settimeofday(). The hires timer is ideally suited to performance measurement tasks, where cheap, accurate interval timing is required.

The linearity of gethrtime() is not preserved across a suspend-resume cycle (Bug 4272663 (http://wesunsolve.net/bugid/id/4272663)).

Read the gethrtime() manual page of Solaris 11 (http://docs.oracle.com/cd/E23824_01/html/821-1465/gethrtime-3c.html#scrolltoc).

On Solaris, gethrtime() is the same as clock_gettime(CLOCK_MONOTONIC).

## System Time (#id56)

| Name | C Resolution | Include Sleep | Include Suspend |
|---|---|---|---|
| CLOCK_REALTIME | 1 ns | Yes | Yes |
| CLOCK_REALTIME_COARSE | 1 ns | Yes | Yes |
| GetSystemTimeAsFileTime | 100 ns | Yes | Yes |
| gettimeofday() | 1 µs | Yes | Yes |
| ftime() | 1 ms | Yes | Yes |
| time() | 1 sec | Yes | Yes |

The "C Resolution" column is the resolution of the underlying C structure.

Examples of clock resolution on x86_64:

| Name | Operating system | OS Resolution | Python Resolution |
|---|---|---|---|
| CLOCK_REALTIME | SunOS 5.11 | 10 ms | 238 ns |
| CLOCK_REALTIME | Linux 3.0 | 1 ns | 238 ns |
| gettimeofday() | Mac OS 10.6 | 1 µs | 4 µs |
| CLOCK_REALTIME | FreeBSD 8.2 | 11 ns | 6 µs |
| CLOCK_REALTIME | OpenBSD 5.0 | 10 ms | 5 µs |
| CLOCK_REALTIME_COARSE | Linux 3.3 | 1 ms | 1 ms |
| CLOCK_REALTIME_COARSE | Linux 3.0 | 4 ms | 4 ms |
| GetSystemTimeAsFileTime() | Windows Seven | 16 ms | 1 ms |
| ftime() | Windows Seven | - | 1 ms |

The "OS Resolution" is the resolution announced by the operating system. The "Python Resolution" is the smallest difference between two calls to the time function computed in Python using the clock_resolution.py (http://hg.python.org/peps/file/tip/pep-0418/clock_resolution.py) program.

## Windows: GetSystemTimeAsFileTime (#id57)

The system time can be read using GetSystemTimeAsFileTime(), ftime() and time(). The resolution of the system time can be read using GetSystemTimeAdjustment().

Read the GetSystemTimeAsFileTime() documentation (http://msdn.microsoft.com/en-us/library/windows/desktop/ms724397(v=vs.85).aspx).

The system time can be set using SetSystemTime().

### System time on UNIX (#id58)

gettimeofday(), ftime(), time() and clock_gettime(CLOCK_REALTIME) return the system time. The resolution of CLOCK_REALTIME can be read using clock_getres().

The system time can be set using settimeofday() or clock_settime(CLOCK_REALTIME).

Linux provides also CLOCK_REALTIME_COARSE since Linux 2.6.32. It is similar to CLOCK_REALTIME, less precise but faster.

Alexander Shishkin proposed an API for Linux to be notified when the system clock is changed: timerfd: add TFD_NOTIFY_CLOCK_SET to watch for clock changes (http://lwn.net/Articles/432395/) (4th version of the API, March 2011). The API is not accepted yet, but CLOCK_BOOTTIME provides a similar feature.

## Process Time (#id59)

The process time cannot be set. It is not monotonic: the clocks stop while the process is idle.

| Name | C Resolution | Include Sleep | Include Suspend |
|---|---|---|---|
| GetProcessTimes() | 100 ns | No | No |
| CLOCK_PROCESS_CPUTIME_ID | 1 ns | No | No |
| getrusage(RUSAGE_SELF) | 1 µs | No | No |
| times() | - | No | No |
| clock() | - | Yes on Windows, No otherwise | No |

The "C Resolution" column is the resolution of the underlying C structure.

Examples of clock resolution on x86_64:

| Name | Operating system | OS Resolution | Python Resolution |
|---|---|---|---|
| CLOCK_PROCESS_CPUTIME_ID | Linux 3.3 | 1 ns | 1 ns |
| CLOCK_PROF | FreeBSD 8.2 | 10 ms | 1 µs |
| getrusage(RUSAGE_SELF) | FreeBSD 8.2 | - | 1 µs |
| getrusage(RUSAGE_SELF) | SunOS 5.11 | - | 1 µs |
| CLOCK_PROCESS_CPUTIME_ID | Linux 3.0 | 1 ns | 1 µs |
| getrusage(RUSAGE_SELF) | Mac OS 10.6 | - | 5 µs |

| Name | Operating system | OS Resolution | Python Resolution |
|---|---|---|---|
| clock() | Mac OS 10.6 | 1 µs | 5 µs |
| CLOCK_PROF | OpenBSD 5.0 | - | 5 µs |
| getrusage(RUSAGE_SELF) | Linux 3.0 | - | 4 ms |
| getrusage(RUSAGE_SELF) | OpenBSD 5.0 | - | 8 ms |
| clock() | FreeBSD 8.2 | 8 ms | 8 ms |
| clock() | Linux 3.0 | 1 µs | 10 ms |
| times() | Linux 3.0 | 10 ms | 10 ms |
| clock() | OpenBSD 5.0 | 10 ms | 10 ms |
| times() | OpenBSD 5.0 | 10 ms | 10 ms |
| times() | Mac OS 10.6 | 10 ms | 10 ms |
| clock() | SunOS 5.11 | 1 µs | 10 ms |
| times() | SunOS 5.11 | 1 µs | 10 ms |
| GetProcessTimes() | Windows Seven | 16 ms | 16 ms |
| clock() | Windows Seven | 1 ms | 1 ms |

The "OS Resolution" is the resolution announced by the operating system. The "Python Resolution" is the smallest difference between two calls to the time function computed in Python using the clock_resolution.py (http://hg.python.org/peps/file/tip/pep-0418/clock_resolution.py) program.

## Functions (#id60)

- Windows: GetProcessTimes() (http://msdn.microsoft.com/en-us/library/windows/desktop/ms683223(v=vs.85).aspx). The resolution can be read using GetSystemTimeAdjustment().
- clock_gettime(CLOCK_PROCESS_CPUTIME_ID): High-resolution per-process timer from the CPU. The resolution can be read using clock_getres().
- clock(). The resolution is 1 / CLOCKS_PER_SEC.
  - Windows: The elapsed wall-clock time since the start of the process (elapsed time in seconds times CLOCKS_PER_SEC). Include time elapsed during sleep. It can fail.
  - UNIX: returns an approximation of processor time used by the program.
- getrusage(RUSAGE_SELF) returns a structure of resource usage of the currenet process. ru_utime is user CPU time and ru_stime is the system CPU time.
- times(): structure of process times. The resolution is 1 / ticks_per_seconds, where ticks_per_seconds is sysconf(_SC_CLK_TCK) or the HZ constant.

Python source code includes a portable library to get the process time (CPU time): Tools/pybench/systimes.py (http://hg.python.org/cpython/file/tip/Tools/pybench/systimes.py).

See also the QueryProcessCycleTime() function (http://msdn.microsoft.com/en-us/library/windows/desktop/ms684929(v=vs.85).aspx) (sum of the cycle time of all threads) and clock_getcpuclockid() (http://www.kernel.org/doc/man-pages/online/pages/man3/clock_getcpuclockid.3.html).

## Thread Time (#id61)

The thread time cannot be set. It is not monotonic: the clocks stop while the thread is idle.

| Name | C Resolution | Include Sleep | Include Suspend |
|---|---|---|---|
| CLOCK_THREAD_CPUTIME_ID | 1 ns | Yes | Epoch changes |
| GetThreadTimes() | 100 ns | No | ? |

The "C Resolution" column is the resolution of the underlying C structure.

Examples of clock resolution on x86_64:

| Name | Operating system | OS Resolution | Python Resolution |
|---|---|---|---|
| CLOCK_THREAD_CPUTIME_ID | FreeBSD 8.2 | 1 µs | 1 µs |
| CLOCK_THREAD_CPUTIME_ID | Linux 3.3 | 1 ns | 649 ns |
| GetThreadTimes() | Windows Seven | 16 ms | 16 ms |

The "OS Resolution" is the resolution announced by the operating system. The "Python Resolution" is the smallest difference between two calls to the time function computed in Python using the clock_resolution.py (http://hg.python.org/peps/file/tip/pep-0418/clock_resolution.py) program.

### Functions (#id62)

- Windows: GetThreadTimes() (http://msdn.microsoft.com/en-us/library/windows/desktop/ms683237(v=vs.85).aspx). The resolution can be read using GetSystemTimeAdjustment().
- clock_gettime(CLOCK_THREAD_CPUTIME_ID): Thread-specific CPU-time clock. It uses a number of CPU cycles, not a number of seconds. The resolution can be read using of clock_getres().

See also the QueryThreadCycleTime() function (http://msdn.microsoft.com/en-us/library/windows/desktop/ms684943(v=vs.85).aspx) (cycle time for the specified thread) and pthread_getcpuclockid().

## Windows: QueryUnbiasedInterruptTime (#id63)

Gets the current unbiased interrupt time from the biased interrupt time and the current sleep bias amount. This time is not affected by power management sleep transitions.

The elapsed time retrieved by the QueryUnbiasedInterruptTime function includes only time that the system spends in the working state. QueryUnbiasedInterruptTime() is not monotonic.

QueryUnbiasedInterruptTime() was introduced in Windows 7.

See also QueryIdleProcessorCycleTime() function (http://msdn.microsoft.com/en-us/library/windows/desktop/ms684922(v=vs.85).aspx) (cycle time for the idle thread of each processor)

## Sleep (#id64)

Suspend execution of the process for the given number of seconds. Sleep is not affected by system time updates. Sleep is paused during system suspend. For example, if a process sleeps for 60 seconds and the system is suspended for 30 seconds in the middle of the sleep, the sleep duration is 90 seconds in the real time.

Sleep can be interrupted by a signal: the function fails with EINTR.

| Name | C Resolution |
|---|---|
| nanosleep() | 1 ns |
| clock_nanosleep() | 1 ns |
| usleep() | 1 µs |
| delay() | 1 µs |
| sleep() | 1 sec |

Other functions:

| Name | C Resolution |
|---|---|
| sigtimedwait() | 1 ns |
| pthread_cond_timedwait() | 1 ns |
| sem_timedwait() | 1 ns |
| select() | 1 µs |
| epoll() | 1 ms |
| poll() | 1 ms |
| WaitForSingleObject() | 1 ms |

The "C Resolution" column is the resolution of the underlying C structure.

## Functions (#id65)

- sleep(seconds)
- usleep(microseconds)
- nanosleep(nanoseconds, remaining): Linux manpage of nanosleep() (http://www.kernel.org/doc/man-pages/online/pages/man2/nanosleep.2.html)
- delay(milliseconds)

## clock_nanosleep (#id66)

clock_nanosleep(clock_id, flags, nanoseconds, remaining): Linux manpage of clock_nanosleep() (http://www.kernel.org/doc/man-pages/online/pages/man2/clock_nanosleep.2.html).

If flags is TIMER_ABSTIME, then request is interpreted as an absolute time as measured by the clock, clock_id. If request is less than or equal to the current value of the clock, then clock_nanosleep() returns immediately without suspending the calling thread.

POSIX.1 specifies that changing the value of the CLOCK_REALTIME clock via clock_settime(2) shall have no effect on a thread that is blocked on a relative clock_nanosleep().

### select() (#id67)

select(nfds, readfds, writefds, exceptfs, timeout).

Since Linux 2.6.28, select() uses high-resolution timers to handle the timeout. A process has a "slack" attribute to configure the precision of the timeout, the default slack is 50 microseconds. Before Linux 2.6.28, timeouts for select() were handled by the main timing subsystem at a jiffy-level resolution. Read also High- (but not too high-) resolution timeouts (http://lwn.net/Articles/296578/) and Timer slack (http://lwn.net/Articles/369549/).

### Other functions (#id68)

- poll(), epoll()
- sigtimedwait(). POSIX: "If the Monotonic Clock option is supported, the CLOCK_MONOTONIC clock shall be used to measure the time interval specified by the timeout argument."
- pthread_cond_timedwait(), pthread_condattr_setclock(). "The default value of the clock attribute shall refer to the system time."
- sem_timedwait(): "If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock. If the Timers option is not supported, the timeout shall be based on the system time as returned by the time() function. The precision of the timeout shall be the precision of the clock on which it is based."
- WaitForSingleObject(): use the same timer than GetTickCount() with the same precision.

## System Standby (#id69)

The ACPI power state "S3" is a system standby mode, also called "Suspend to RAM". RAM remains powered.

On Windows, the `WM_POWERBROADCAST` message is sent to Windows applications to notify them of power-management events (ex: owner status has changed).

For Mac OS X, read Registering and unregistering for sleep and wake notifications (http://developer.apple.com/library/mac/#qa/qa1340/_index.html) (Technical Q&A QA1340).

## Footnotes (#id70)

[2] *(1 (#id1), 2 (#id2), 3 (#id3), 4 (#id4), 5 (#id5))* "_time" is a hypothetical module only used for the example. The time module is implemented in C and so there is no need for such a module.

## Links (#id71)

Related Python issues:

- Issue #12822: NewGIL should use CLOCK_MONOTONIC if possible. (http://bugs.python.org/issue12822)

- Issue #14222: Use time.steady() to implement timeout (http://bugs.python.org/issue14222)
- Issue #14309: Deprecate time.clock() (http://bugs.python.org/issue14309)
- Issue #14397: Use GetTickCount/GetTickCount64 instead of QueryPerformanceCounter for monotonic clock (http://bugs.python.org/issue14397)
- Issue #14428: Implementation of the PEP 418 (http://bugs.python.org/issue14428)
- Issue #14555: clock_gettime/settime/getres: Add more clock identifiers (http://bugs.python.org/issue14555)

Libraries exposing monotonic clocks:

- Java: System.nanoTime (http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/System.html#nanoTime())
- Qt library: QElapsedTimer (http://qt-project.org/doc/qt-4.8/qelapsedtimer.html)
- glib library: g_get_monotonic_time () (http://developer.gnome.org/glib/2.30/glib-Date-and-Time-Functions.html#g-get-monotonic-time) uses GetTickCount64()/GetTickCount() on Windows, clock_gettime(CLOCK_MONOTONIC) on UNIX or falls back to the system clock
- python-monotonic-time (http://code.google.com/p/python-monotonic-time/) (github (https://github.com/gavinbeatty/python-monotonic-time))
- Monoclock.nano_count() (https://github.com/ludios/Monoclock) uses clock_gettime(CLOCK_MONOTONIC) and returns a number of nanoseconds
- monotonic_clock (https://github.com/ThomasHabets/monotonic_clock) by Thomas Habets
- Perl: Time::HiRes (http://perldoc.perl.org/Time/HiRes.html) exposes clock_gettime(CLOCK_MONOTONIC)
- Ruby: AbsoluteTime.now (https://github.com/bwbuchanan/absolute_time/): use clock_gettime(CLOCK_MONOTONIC), mach_absolute_time() or gettimeofday(). "AbsoluteTime.monotonic?" method indicates if AbsoluteTime.now is monotonic or not.
- libpthread (http://code.google.com/p/libpthread/): POSIX thread library for Windows (clock.c (http://code.google.com/p/libpthread/source/browse/src/clock.c))
- Boost.Chrono (http://www.boost.org/doc/libs/1_49_0/doc/html/chrono.html) uses:
  - system_clock:
    - mac = gettimeofday()
    - posix = clock_gettime(CLOCK_REALTIME)
    - win = GetSystemTimeAsFileTime()
  - steady_clock:
    - mac = mach_absolute_time()
    - posix = clock_gettime(CLOCK_MONOTONIC)
    - win = QueryPerformanceCounter()
  - high_resolution_clock:
    - steady_clock, if available system_clock, otherwise

Time:

- Twisted issue #2424: Add reactor option to start with monotonic clock (http://twistedmatrix.com/trac/ticket/2424)
- gettimeofday() should never be used to measure time (http://blog.habets.pp.se/2010/09/gettimeofday-should-never-be-used-to-measure-time) by Thomas Habets (2010-09-05)
- hrtimers - subsystem for high-resolution kernel timers (http://www.kernel.org/doc/Documentation/timers/hrtimers.txt)
- C++ Timeout Specification (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3128.html) by Lawrence Crowl (2010-08-19)
- Windows: Game Timing and Multicore Processors (http://msdn.microsoft.com/en-us/library/ee417693.aspx) by Chuck Walbourn (December 2005)
- Implement a Continuously Updating, High-Resolution Time Provider for Windows (http://msdn.microsoft.com/en-us/magazine/cc163996.aspx) by Johan Nilsson (March 2004)

- clockspeed (http://cr.yp.to/clockspeed.html) uses a hardware tick counter to compensate for a persistently fast or slow system time, by D. J. Bernstein (1998)
- Retrieving system time (http://en.wikipedia.org/wiki/System_time#Retrieving_system_time) lists hardware clocks and time functions with their resolution and epoch or range
- On Windows, the JavaScript runtime of Firefox interpolates GetSystemTimeAsFileTime() with QueryPerformanceCounter() to get a higher resolution. See the Bug 363258 - bad millisecond resolution for (new Date).getTime() / Date.now() on Windows (https://bugzilla.mozilla.org/show_bug.cgi?id=363258).
- When microseconds matter (http://www.ibm.com/developerworks/library/i-seconds/): How the IBM High Resolution Time Stamp Facility accurately measures itty bits of time, by W. Nathaniel Mills, III (Apr 2002)
- Win32 Performance Measurement Options (http://drdobbs.com/windows/184416651) by Matthew Wilson (May, 2003)
- Counter Availability and Characteristics for Feed-forward Based Synchronization (http://www.cubinlab.ee.unimelb.edu.au/~jrid/Publications/ridoux_ispcs09.pdf) by Timothy Broomhead, Julien Ridoux, Darryl Veitch (2009)
- System Management Interrupt (SMI) issues:
  - System Management Interrupt Free Hardware (http://linuxplumbersconf.org/2009/slides/Keith-Mannthey-SMI-plumers-2009.pdf) by Keith Mannthey (2009)
  - IBM Real-Time "SMI Free" mode driver (http://lwn.net/Articles/318725/) by Keith Mannthey (Feb 2009)
  - Fixing Realtime problems caused by SMI on Ubuntu (http://wiki.linuxcnc.org/cgi-bin/wiki.pl?FixingSMIIssues)
  - [RFC] simple SMI detector (http://lwn.net/Articles/316622/) by Jon Masters (Jan 2009)
  - [PATCH 2.6.34-rc3] A nonintrusive SMI sniffer for x86 (http://marc.info/?l=linux-kernel&m=127058720921201&w=1) by Joe Korty (2010-04)

## Acceptance (#id72)

The PEP was accepted on 2012-04-28 by Guido van Rossum [1] (#id12). The PEP implementation has since been committed to the repository.

## References (#id73)


[1] (#id11) https://mail.python.org/pipermail/python-dev/2012-April/119094.html (https://mail.python.org/pipermail/python-dev/2012-April/119094.html)

[3] (#id14) http://support.microsoft.com/?id=896256 (http://support.microsoft.com/?id=896256)

[4] *(1 (#id16), 2 (#id17))* http://support.microsoft.com/?id=274323 (http://support.microsoft.com/?id=274323)


## Copyright (#id74)



Source: https://github.com/python/peps/blob/master/pep-0418.txt (https://github.com/python/peps/blob/master/pep-0418.txt)

## Tweets by @ThePSF

**Python Software Foundation**
@ThePSF

Support the Python community by becoming a #pycon2020 Patron! It comes with a full conference pass and your name on our new "Patron Wall"! Check out all our sponsorship options here: us.pycon.org/2020/sponsors/…

23h

**Python Software Foundation**
@ThePSF

Start using 2FA and API tokens on PyPI ift.tt/2NwuVUj

Jan 17, 2020

**Python Software Foundation**
@ThePSF

The PSF January Newsletter is out! Sign up now to receive community news: python.org/psf/newsletter/.

**PSF Newsletter Signup**
The official home of the Python Programming Language
python.org

Embed View on Twitter

## The PSF

The Python Software Foundation is the organization behind Python. Become a member of the PSF and help advance the software and our mission.

Back to Top

Back to Top



/