NAME

Time::HiRes - High resolution alarm, sleep, gettimeofday, interval timers

SYNOPSIS

```
use Time::HiRes qw( usleep ualarm gettimeofday tv_interval nanosleep
      clock_gettime clock_getres clock_nanosleep clock
                    stat );

usleep ($microseconds);
nanosleep ($nanoseconds);

ualarm ($microseconds);
ualarm ($microseconds, $interval_microseconds);

$t0 = [gettimeofday];
($seconds, $microseconds) = gettimeofday;

$elapsed = tv_interval ( $t0, [$seconds, $microseconds]);
$elapsed = tv_interval ( $t0, [gettimeofday]);
$elapsed = tv_interval ( $t0 );

use Time::HiRes qw ( time alarm sleep );

$now_fractions = time;
sleep ($floating_seconds);
alarm ($floating_seconds);
alarm ($floating_seconds, $floating_interval);

use Time::HiRes qw( setitimer getitimer );

setitimer ($which, $floating_seconds, $floating_interval );
getitimer ($which);

use Time::HiRes qw( clock_gettime clock_getres clock_nanosleep
      ITIMER_REAL ITIMER_VIRTUAL ITIMER_PROF ITIMER_REALPROF );

$realtime   = clock_gettime(CLOCK_REALTIME);
$resolution = clock_getres(CLOCK_REALTIME);

clock_nanosleep(CLOCK_REALTIME, 1.5e9);
clock_nanosleep(CLOCK_REALTIME, time()*1e9 + 10e9, TIMER_ABSTIME);

my $ticktock = clock();

use Time::HiRes qw( stat );

my @stat = stat("file");
my @stat = stat(FH);
```

DESCRIPTION

The `Time::HiRes` module implements a Perl interface to the `usleep`, `nanosleep`, `ualarm`, `gettimeofday`, and `setitimer`/`getitimer` system calls, in other words, high resolution time and timers. See the *EXAMPLES* section below and the test scripts for usage; see your system documentation for the description of the underlying `nanosleep` or `usleep`, `ualarm`, `gettimeofday`, and `setitimer`/`getitimer` calls.

If your system lacks `gettimeofday()` or an emulation of it you don't get `gettimeofday()` or the one-argument form of `tv_interval()`. If your system lacks all of `nanosleep()`, `usleep()`, `select()`, and `poll`, you don't get `Time::HiRes::usleep()`, `Time::HiRes::nanosleep()`, or `Time::HiRes::sleep()`. If your system lacks both `ualarm()` and `setitimer()` you don't get `Time::HiRes::ualarm()` or `Time::HiRes::alarm()`.

If you try to import an unimplemented function in the `use` statement it will fail at compile time.

If your subsecond sleeping is implemented with `nanosleep()` instead of `usleep()`, you can mix subsecond sleeping with signals since `nanosleep()` does not use signals. This, however, is not portable, and you should first check for the truth value of `&Time::HiRes::d_nanosleep` to see whether you have nanosleep, and then carefully read your `nanosleep()` C API documentation for any peculiarities.

If you are using `nanosleep` for something else than mixing sleeping with signals, give some thought to whether Perl is the tool you should be using for work requiring nanosecond accuracies.

Remember that unless you are working on a *hard realtime* system, any clocks and timers will be imprecise, especially so if you are working in a pre-emptive multiuser system. Understand the difference between *wallclock time* and process time (in UNIX-like systems the sum of *user* and *system* times). Any attempt to sleep for X seconds will most probably end up sleeping **more** than that, but don't be surpised if you end up sleeping slightly **less**.

The following functions can be imported from this module. No functions are exported by default.

gettimeofday ()

In array context returns a two-element array with the seconds and microseconds since the epoch. In scalar context returns floating seconds like `Time::HiRes::time()` (see below).

usleep ($useconds)

Sleeps for the number of microseconds (millionths of a second) specified. Returns the number of microseconds actually slept. Can sleep for more than one second, unlike the `usleep` system call. Can also sleep for zero seconds, which often works like a *thread yield*. See also `Time::HiRes::usleep()`, `Time::HiRes::sleep()`, and `Time::HiRes::clock_nanosleep()`.

Do not expect usleep() to be exact down to one microsecond.

nanosleep ($nanoseconds)

Sleeps for the number of nanoseconds (1e9ths of a second) specified. Returns the number of nanoseconds actually slept (accurate only to microseconds, the nearest thousand of them). Can sleep for more than one second. Can also sleep for zero seconds, which often works like a *thread yield*. See also `Time::HiRes::sleep()`, `Time::HiRes::usleep()`, and `Time::HiRes::clock_nanosleep()`.

Do not expect nanosleep() to be exact down to one nanosecond. Getting even accuracy of one thousand nanoseconds is good.

ualarm ($useconds [, $interval_useconds])

Issues a `ualarm` call; the `$interval_useconds` is optional and will be zero if unspecified, resulting in `alarm`-like behaviour.

ualarm(0) will cancel an outstanding ualarm().

Note that the interaction between alarms and sleeps is unspecified.

tv_interval

tv_interval ($ref_to_gettimeofday [, $ref_to_later_gettimeofday])

Returns the floating seconds between the two times, which should have been returned by `gettimeofday()`. If the second argument is omitted, then the current time is used.

time ()

Returns a floating seconds since the epoch. This function can be imported, resulting in a nice drop-in replacement for the `time` provided with core Perl; see the *EXAMPLES* below.

NOTE 1: This higher resolution timer can return values either less or more than the core `time()`, depending on whether your platform rounds the higher resolution timer values up, down, or to the nearest second to get the core `time()`, but naturally the difference should be never more than half a second. See also *clock_getres*, if available in your system.

NOTE 2: Since Sunday, September 9th, 2001 at 01:46:40 AM GMT, when the `time()` seconds since epoch rolled over to 1_000_000_000, the default floating point format of Perl and the seconds since epoch have conspired to produce an apparent bug: if you print the value of `Time::HiRes::time()` you seem to be getting only five decimals, not six as promised (microseconds). Not to worry, the microseconds are there (assuming your platform supports such granularity in the first place). What is going on is that the default floating point format of Perl only outputs 15 digits. In this case that means ten digits before the decimal separator and five after. To see the microseconds you can use either `printf`/`sprintf` with `"%.6f"`, or the `gettimeofday()` function in list context, which will give you the seconds and microseconds as two separate values.

sleep ($floating_seconds)

Sleeps for the specified amount of seconds. Returns the number of seconds actually slept (a floating point value). This function can be imported, resulting in a nice drop-in replacement for the `sleep` provided with perl, see the *EXAMPLES* below.

Note that the interaction between alarms and sleeps is unspecified.

alarm ($floating_seconds [, $interval_floating_seconds])

The `SIGALRM` signal is sent after the specified number of seconds. Implemented using `ualarm()`. The `$interval_floating_seconds` argument is optional and will be zero if unspecified, resulting in `alarm()`-like behaviour. This function can be imported, resulting in a nice drop-in replacement for the `alarm` provided with perl, see the *EXAMPLES* below.

NOTE 1: With some combinations of operating systems and Perl releases `SIGALRM` restarts `select()`, instead of interrupting it. This means that an `alarm()` followed by a `select()` may together take the sum of the times specified for the the `alarm()` and the `select()`, not just the time of the `alarm()`.

Note that the interaction between alarms and sleeps is unspecified.

setitimer ($which, $floating_seconds [, $interval_floating_seconds])

Start up an interval timer: after a certain time, a signal ($which) arrives, and more signals may keep arriving at certain intervals. To disable an "itimer", use `$floating_seconds` of zero. If the `$interval_floating_seconds` is set to zero (or unspecified), the timer is disabled **after** the next delivered signal.

Use of interval timers may interfere with `alarm()`, `sleep()`, and `usleep()`. In standard-speak the "interaction is unspecified", which means that *anything* may happen: it may work, it may not.

In scalar context, the remaining time in the timer is returned.

In list context, both the remaining time and the interval are returned.

There are usually three or four interval timers (signals) available: the `$which` can be

ITIMER_REAL, ITIMER_VIRTUAL, ITIMER_PROF, or ITIMER_REALPROF. Note that which ones are available depends: true UNIX platforms usually have the first three, but (for example) Win32 and Cygwin have only ITIMER_REAL, and only Solaris seems to have ITIMER_REALPROF (which is used to profile multithreaded programs).

ITIMER_REAL results in alarm()-like behaviour. Time is counted in *real time*; that is, wallclock time. SIGALRM is delivered when the timer expires.

ITIMER_VIRTUAL counts time in (process) *virtual time*; that is, only when the process is running. In multiprocessor/user/CPU systems this may be more or less than real or wallclock time. (This time is also known as the *user time*.) SIGVTALRM is delivered when the timer expires.

ITIMER_PROF counts time when either the process virtual time or when the operating system is running on behalf of the process (such as I/O). (This time is also known as the *system time*.) (The sum of user time and system time is known as the *CPU time*.) SIGPROF is delivered when the timer expires. SIGPROF can interrupt system calls.

The semantics of interval timers for multithreaded programs are system-specific, and some systems may support additional interval timers. For example, it is unspecified which thread gets the signals. See your setitimer() documentation.

getitimer ($which)

Return the remaining time in the interval timer specified by $which.

In scalar context, the remaining time is returned.

In list context, both the remaining time and the interval are returned. The interval is always what you put in using setitimer().

clock_gettime ($which)

Return as seconds the current value of the POSIX high resolution timer specified by $which. All implementations that support POSIX high resolution timers are supposed to support at least the $which value of CLOCK_REALTIME, which is supposed to return results close to the results of gettimeofday, or the number of seconds since 00:00:00:00 January 1, 1970 Greenwich Mean Time (GMT). Do not assume that CLOCK_REALTIME is zero, it might be one, or something else. Another potentially useful (but not available everywhere) value is CLOCK_MONOTONIC, which guarantees a monotonically increasing time value (unlike time(), which can be adjusted). See your system documentation for other possibly supported values.

clock_getres ($which)

Return as seconds the resolution of the POSIX high resolution timer specified by $which. All implementations that support POSIX high resolution timers are supposed to support at least the $which value of CLOCK_REALTIME, see *clock_gettime*.

clock_nanosleep ($which, $nanoseconds, $flags = 0)

Sleeps for the number of nanoseconds (1e9ths of a second) specified. Returns the number of nanoseconds actually slept. The $which is the "clock id", as with clock_gettime() and clock_getres(). The flags default to zero but TIMER_ABSTIME can specified (must be exported explicitly) which means that $nanoseconds is not a time interval (as is the default) but instead an absolute time. Can sleep for more than one second. Can also sleep for zero seconds, which often works like a *thread yield*. See also Time::HiRes::sleep(), Time::HiRes::usleep(), and Time::HiRes::nanosleep().

Do not expect clock_nanosleep() to be exact down to one nanosecond. Getting even accuracy of one thousand nanoseconds is good.

clock()

Return as seconds the *process time* (user + system time) spent by the process since the first call to clock() (the definition is **not** "since the start of the process", though if you are lucky these times may be quite close to each other, depending on the system). What this means is

that you probably need to store the result of your first call to clock(), and subtract that value from the following results of clock().

The time returned also includes the process times of the terminated child processes for which wait() has been executed. This value is somewhat like the second value returned by the times() of core Perl, but not necessarily identical. Note that due to backward compatibility limitations the returned value may wrap around at about 2147 seconds or at about 36 minutes.

stat

stat FH

stat EXPR

As *"stat" in perlfunc* but with the access/modify/change file timestamps in subsecond resolution, if the operating system and the filesystem both support such timestamps. To override the standard stat():

```
use Time::HiRes qw(stat);
```

Test for the value of &Time::HiRes::d_hires_stat to find out whether the operating system supports subsecond file timestamps: a value larger than zero means yes. There are unfortunately no easy ways to find out whether the filesystem supports such timestamps. UNIX filesystems often do; NTFS does; FAT doesn't (FAT timestamp granularity is **two** seconds).

A zero return value of &Time::HiRes::d_hires_stat means that Time::HiRes::stat is a no-op passthrough for CORE::stat(), and therefore the timestamps will stay integers. The same thing will happen if the filesystem does not do subsecond timestamps, even if the &Time::HiRes::d_hires_stat is non-zero.

In any case do not expect nanosecond resolution, or even a microsecond resolution. Also note that the modify/access timestamps might have different resolutions, and that they need not be synchronized, e.g. if the operations are

```
write
stat # t1
read
stat # t2
```

the access time stamp from t2 need not be greater-than the modify time stamp from t1: it may be equal or *less*.

EXAMPLES

```
use Time::HiRes qw(usleep ualarm gettimeofday tv_interval);

$microseconds = 750_000;
usleep($microseconds);

# signal alarm in 2.5s & every .1s thereafter
ualarm(2_500_000, 100_000);
# cancel that ualarm
ualarm(0);

# get seconds and microseconds since the epoch
($s, $usec) = gettimeofday();

# measure elapsed time
# (could also do by subtracting 2 gettimeofday return values)
$t0 = [gettimeofday];
```

```
# do bunch of stuff here
$t1 = [gettimeofday];
# do more stuff here
$t0_t1 = tv_interval $t0, $t1;

$elapsed = tv_interval ($t0, [gettimeofday]);
$elapsed = tv_interval ($t0); # equivalent code


#
# replacements for time, alarm and sleep that know about
# floating seconds
#
use Time::HiRes;
$now_fractions = Time::HiRes::time;
Time::HiRes::sleep (2.5);
Time::HiRes::alarm (10.6666666);

use Time::HiRes qw ( time alarm sleep );
$now_fractions = time;
sleep (2.5);
alarm (10.6666666);

# Arm an interval timer to go off first at 10 seconds and
# after that every 2.5 seconds, in process virtual time

use Time::HiRes qw ( setitimer ITIMER_VIRTUAL time );

$SIG{VTALRM} = sub { print time, "\n" };
setitimer(ITIMER_VIRTUAL, 10, 2.5);

use Time::HiRes qw( clock_gettime clock_getres CLOCK_REALTIME );
# Read the POSIX high resolution timer.
my $high = clock_getres(CLOCK_REALTIME);
# But how accurate we can be, really?
my $reso = clock_getres(CLOCK_REALTIME);

use Time::HiRes qw( clock_nanosleep TIMER_ABSTIME );
clock_nanosleep(CLOCK_REALTIME, 1e6);
clock_nanosleep(CLOCK_REALTIME, 2e9, TIMER_ABSTIME);

use Time::HiRes qw( clock );
my $clock0 = clock();
... # Do something.
my $clock1 = clock();
my $clockd = $clock1 - $clock0;

use Time::HiRes qw( stat );
my ($atime, $mtime, $ctime) = (stat("istics"))[8, 9, 10];
```

C API

In addition to the perl API described above, a C API is available for extension writers. The following C functions are available in the modglobal hash:

```
name             C prototype
---------------  ---------------------
Time::NVtime     double (*)()
Time::U2time     void (*)(pTHX_ UV ret[2])
```

Both functions return equivalent information (like `gettimeofday`) but with different representations. The names `NVtime` and `U2time` were selected mainly because they are operating system independent. (`gettimeofday` is Unix-centric, though some platforms like Win32 and VMS have emulations for it.)

Here is an example of using `NVtime` from C:

```
double (*myNVtime)(); /* Returns -1 on failure. */
SV **svp = hv_fetch(PL_modglobal, "Time::NVtime", 12, 0);
if (!svp)         croak("Time::HiRes is required");
if (!SvIOK(*svp)) croak("Time::NVtime isn't a function pointer");
myNVtime = INT2PTR(double(*)(), SvIV(*svp));
printf("The current time is: %f\n", (*myNVtime)());
```

DIAGNOSTICS

useconds or interval more than ...

In ualarm() you tried to use number of microseconds or interval (also in microseconds) more than 1_000_000 and setitimer() is not available in your system to emulate that case.

negative time not invented yet

You tried to use a negative time argument.

internal error: useconds < 0 (unsigned ... signed ...)

Something went horribly wrong-- the number of microseconds that cannot become negative just became negative. Maybe your compiler is broken?

CAVEATS

Notice that the core `time()` maybe rounding rather than truncating. What this means is that the core `time()` may be reporting the time as one second later than `gettimeofday()` and `Time::HiRes::time()`.

Adjusting the system clock (either manually or by services like ntp) may cause problems, especially for long running programs that assume a monotonously increasing time (note that all platforms do not adjust time as gracefully as UNIX ntp does). For example in Win32 (and derived platforms like Cygwin and MinGW) the Time::HiRes::time() may temporarily drift off from the system clock (and the original time()) by up to 0.5 seconds. Time::HiRes will notice this eventually and recalibrate. Note that since Time::HiRes 1.77 the clock_gettime(CLOCK_MONOTONIC) might help in this (in case your system supports CLOCK_MONOTONIC).

SEE ALSO

Perl modules *BSD::Resource*, *Time::TAI64*.

Your system documentation for `clock`, `clock_gettime`, `clock_getres`, `clock_nanosleep`, `clock_settime`, `getitimer`, `gettimeofday`, `setitimer`, `sleep`, `stat`, `ualarm`.

AUTHORS

D. Wegscheid <wegscd@whirlpool.com> R. Schertler <roderick@argon.org> J. Hietaniemi <jhi@iki.fi> G. Aas <gisle@aas.no>

COPYRIGHT AND LICENSE