Subject: usecs
Posted by mirek on Mon, 12 Nov 2018 10:31:04 GMT
View Forum Message <> Reply to Message

In addition to "msecs", we have now microseconds precision "usecs". It is really a trivial wrapper around std::chrono.

Subject: Re: usecs
Posted by Tom1 on Wed, 05 Dec 2018 11:59:32 GMT
View Forum Message <> Reply to Message

Hi,

Is usecs() a monotonically increasing timer? (I.e. no jumps backward or forward ever, even if OS time is adjusted by user or NTP.)

Best regards,

Tom

Subject: Re: usecs
Posted by Zbych on Wed, 05 Dec 2018 22:36:20 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Wed, 05 December 2018 12:59Is usecs() a monotonically increasing timer? (I.e. no jumps backward or forward ever, even if OS time is adjusted by user or NTP.)

According to the docs: https://en.cppreference.com/w/cpp/chrono/high_resolution_clo ck
it is implementation dependent.
It also means that new implementation of msecs() might not be monotonic as well :(and all timeouts in many places (sockets etc.) might randomly get shortened or extended.

If you need monotonic clock use std::chrono::steady_clock.

Subject: Re: usecs
Posted by Tom1 on Thu, 06 Dec 2018 17:40:17 GMT
View Forum Message <> Reply to Message

Hi Zbych,

And thanks for your input! It seems I need to stick with my previous timer solutions which are monotonic. I was just looking at an easier way out than using QueryPerformanceCounter on Windows. Linux is easy enough as it is.

Best regards,

Tom

Subject: Re: usecs
Posted by mirek on Thu, 06 Dec 2018 18:46:03 GMT
View Forum Message <> Reply to Message

Tom1 wrote on Thu, 06 December 2018 18:40Hi Zbych,

And thanks for your input! It seems I need to stick with my previous timer solutions which are monotonic. I was just looking at an easier way out than using QueryPerformanceCounter on Windows. Linux is easy enough as it is.

Best regards,

Tom

It is not written in the stone. I can change msecs / usecs... A the moment, it seemed like a good solution to use what C++ lib provides.

Subject: Re: usecs
Posted by Zbych on Thu, 06 Dec 2018 19:19:49 GMT
View Forum Message <> Reply to Message

mirek wrote on Thu, 06 December 2018 19:46A the moment, it seemed like a good solution to use what C++ lib provides.

I strongly disagree. I've made a test and std::chrono::high_resolution_clock::is_steady returned false in Linux/Gcc.
That means that new version of msecs is not reliable. Maybe for usecs it doesn't matter, but msecs is used to measure timeouts in many places in Upp.

My proposition is to use steady_clock for msecs instead:

```
int msecs(int prev)
{
 auto p2 = std::chrono::steady_clock::now();
 return (int)std::chrono::duration_cast<std::chrono::milliseconds>(p2.time_since_epoch()).count() - prev;
}
```

I compared returned value with old implementation (clock_gettime(CLOCK_MONOTONIC...) and

they are exactly the same.
Resolution of steady_clock in Linux is about 1ms, so it doesn't make sense to use it in usecs.

Subject: Re: usecs
Posted by mirek on Thu, 06 Dec 2018 19:56:55 GMT
View Forum Message <> Reply to Message

Zbych wrote on Thu, 06 December 2018 20:19mirek wrote on Thu, 06 December 2018 19:46A the moment, it seemed like a good solution to use what C++ lib provides.

I strongly disagree.

Yet you are proposing using C++ lib as well :)

Quote:
My proposition is to use steady_clock for msecs instead:

```
int msecs(int prev)
{
 auto p2 = std::chrono::steady_clock::now();
 return (int)std::chrono::duration_cast<std::chrono::milliseconds>(p2.time_since_epoch()).count() - prev;
}
```

Accepted. I think this is a good idea, we just need to remember that msecs is steady and usecs is not. I guess there is little harm that way - we can anticipate the use of msecs to synchronize things like sockets and usecs to benchmark things (and that is notoriously unstable for other reasons too).

Mirek

Subject: Re: usecs
Posted by Zbych on Thu, 06 Dec 2018 20:24:52 GMT
View Forum Message <> Reply to Message

mirek wrote on Thu, 06 December 2018 20:56Yet you are proposing using C++ lib as well :)

You got me :)

BTW. I've made a mistake. Actual resolution of steady clock in Linux seems to be way below 1us.

I don't know how it looks in windows, but steady_clock seems to be good candidate for usecs as well.

```
int64 nsecs(int64 prev = 0)
{
 auto p2 = std::chrono::high_resolution_clock::now();
 return std::chrono::duration_cast<std::chrono::nanoseconds>(p2.time_since_epoch()).count() -
prev;
}


int64 steady_time(int64 prev = 0)
{
 auto p2 = std::chrono::steady_clock::now();
 return
(int64)std::chrono::duration_cast<std::chrono::nanoseconds>(p2.time_since_epoch()).count() -
prev;
}


CONSOLE_APP_MAIN
{
 int64 max_delay = 0;
 int64 av_delay = 0;
 constexpr int loops = 10000000;

 for (int i = 0; i < loops; i++){

  auto start = nsecs();
  auto ms = steady_time();
  while (steady_time(ms) == 0);
  auto duration = nsecs(start);
  max_delay = std::max(duration, max_delay);
  av_delay += duration;
 }
 RLOG("Max delay: " << max_delay << "ns");
 RLOG("Av delay: " << av_delay/loops << "ns");
}
```

Subject: Re: usecs
Posted by Tom1 on Fri, 07 Dec 2018 09:02:00 GMT

Hi,

In my opinion, two timing methods are needed. One is the 'wall clock', i.e. something that OS supplies as UTC (or local time, but rather as UTC). This may be synchronized by e.g. NTP or adjusted by user and as such it is always just an approximation of time. There are no guarantees of its correctness or quality.

The other timing source needed is a counter for interval measurement and monotonicity (or 'steadyness' as they seem to call it in C++ lib) is a vital property of it. Monotonic counters are in the end the only solid reference for any serious timing.

(I use these monotonic counters for e.g. creating a 'UTC wall clock' that is synchronized way beyond the accuracy of any OS based clock. The timing requirements of my application are far beyond what e.g. NTP can supply over network.)

For the above reasons, I would suggest using only such APIs for msecs() and usecs() that guarantee in documentation the monotonicity of the clock. Using a source that looks fine on my system and yours does not guarantee similar behavior on the other client's system. After all these are highly hardware dependent in addition to the software platform used.

On POSIX I would stick with "clock_gettime(CLOCK_MONOTONIC,&now);". (This is what I would like to have on Windows too, but that's just a wish...)

For Windows, I would carefully read:

https://docs.microsoft.com/en-us/windows/desktop/SysInfo/acquiring-high-resolution-time-stamps

and then go with QueryPerformanceCounter (QPC).

-

How about adding a future proof solution to U++: "int64 nsecs_monotonic()"? I guess the name says it all. The resolution will be as good as the platform can supply and monotonicity will guarantee its usefulness for any application.

Best regards,

Tom