[Comments on timing short code sections on Intel processors (utexas.edu)](https://sites.utexas.edu/jdm4372/2018/07/23/comments-on-timing-short-code-sections-on-intel-processors/)  
  
  
После ухода работодателя, у меня появилась возможность отдохнуть и с чистой совестью заняться чем-нибудь для себя.

Очень часто комментарии к постам читать интереснее и полезнее чем породивший их текст. Надеюсь и написанное мной будет прокомментировано более грамотными программистами.

Ниже иногда будут приводиться конкретные числа. Если не указано другое, то имеются в виду замеры для компьютера на процессоре Intel i7-6700K с ОС Ubuntu 22.04 LTS или Win10 Pro 22H2.

Приводимые мной числа предназначены исключительно для сравнения. Замеры, сделанные на десктопных ОС, носят лишь оценочный характер, разброс таких замеров очень велик, не придирайтесь пожалуйста.

Вместо термина «точность» измерения времени здесь будет использоваться термин «дискретность», то есть минимальное изменение возвращаемых функцией значений. Дискретность можно померить, а оценка точности - тема для отдельного большого исследования. Во всяком случае точность не может быть лучше, чем дискретность.

От функций измерения интервалов времени требуется: минимальная дискретность, минимальная длительность вызова (накладные расходы), желательны монотонность и кроссплатформенность. Не для всех функций в документации указывается дискретность, а длительность вызова зависит железа, поэтому их пришлось измерять.

Краткие (если получится) замечания по наиболее популярным функциям измерения времени:

1. clock() [С89] – древнейшая функция появилась в первой редакции ANSI C и предназначалась как раз для точного измерения интервалов времени. По стандарту функция должна возвращать «processor time used by the program so far» но по историческим причинам её реализация у Microsoft не соответствует стандарту и возвращает «the elapsed time since the CRT initialization at the start of the process». Это различие не имело значения в одно-процессных ОС (например, в MS DOS) но принципиально для современных многозадачных ОС. Так что для портируемых приложений эта функция не годится. Кроме того, «правильная» реализация функции в Linux очень медленная (длительность вызова 960ns!!! против 36ns на Windows). Вообще все функции, возвращающие время, связанное с процессом или потоком, очень неспешны, видимо их реализация требует перехода в режим ядра, а это, как известно, операция дорогая. Дискретность этой функции сегодня тоже не удовлетворительна (1ms в Windows и 1us в POSIX).
2. timespec\_get() [C11] – Позволяет получать системное время в наносекундах. Но не следует путать единицы измерения и точность – в Windows эта функция имеет дискретность 100ns а в Linux зависит от железа (от 22ns на i7-6700K до 1,1us на Raspberry Pi). То, что функция возвращает системное время делает её не монотонной – система может корректировать время (даже назад).
3. high\_resolution\_clock и steady\_clock [C++11] - функции now() этих классов достаточно лёгкие и позволяют получать время в наносекундах (см. комментарий к timespec\_get()). В стандарте оговаривается, что high\_resolution\_clock предоставляет наиточнейшее измерение времени, но есть нюанс, эти часы не обязаны быть steady. В Linux high\_resolution\_clock является синонимом для system\_clock а значит зависит от коррекции времени, как и timespec\_get, короче говоря, в Linux время измеренное high\_resolution\_clock может уменьшаться! Зачем так сделано не знаю, в Windows этот класс является синонимом steady\_clock и потому является монотонным. На исследованных платформах дискретность и длительность измерения времени с помощью всех классов std::chrono оказались одинаковы и соответствуют timespec\_get(). Возможно в большинстве случаев можно прямо использовать steady\_clock и не связываться с high\_resolution\_clock.
4. gettimeofday() [POSIX] – позволяет получать время в микросекундах. На сегодня считается устаревшей - «POSIX.1-2008 marks gettimeofday() as obsolete, recommending the use of clock\_gettime() instead». Упоминаю её лишь потому, что она часто используется в примерах из интернета.
5. clock\_gettime(CLOCK\_MONOTONIC\_RAW) [Linux] – видимо лучший вариант LinuxAPI. Точность и время выполнения около 22ns.
6. QueryPerformanceCounter() [WinAPI] – хороший вариант из WinAPI. Функция монотонна, дискретность – 100ns, время выполнения - 13ns.
7. RDTSC, RDTSCP - для измерения интервалов времени на x86 процессорах в интернете часто рекомендуют эти инструкции. Здесь могут быть две проблемы: первая - TSC может обновляться с разной скоростью; вторая - процессор может переупорядочивать инструкции.  
   Первая проблема для современных процессоров уже не актуальна: «For Pentium 4 processors, Intel Xeon processors (family [0FH], models [03H and higher]); for Intel Core Solo and Intel Core Duo processors (family [06H], model [0EH]); for the Intel Xeon processor 5100 series and Intel Core 2 Duo processors (family [06H], model [0FH]); for Intel Core 2 and Intel Xeon processors (family [06H], DisplayModel [17H]); for Intel Atom processors (family [06H], DisplayModel [1CH]): the **time-stamp counter increments at a constant rate**”(Intel® 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. Vol. 3B. P.18-42)  
   Вторую проблему рекомендуют уменьшить двумя способами:
   1. Использовать сочетание инструкций RDTSCP и LFANCE: «The RDTSCP instruction is not a serializing instruction, but it does wait until all previous instructions have executed <…> If software requires RDTSCP to be executed prior to execution of any subsequent instruction (including any memory accesses), it can execute LFENCE immediately after RDTSCP» (Intel® 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. Vol. 2B. P.4-553). Если я правильно понял, то рекомендуется нечто подобное этому:

Для MASM:

public vi\_get\_tsc

vi\_get\_tsc PROC

rdtscp

lfence

shl rdx, 32

or rax, rdx

ret 0

vi\_get\_tsc ENDP

Для GCC:

inline uint64\_t vi\_asm\_rdtscp\_lfence()

{ uint32\_t aux;

uint64\_t low, high;

\_\_asm\_\_ volatile("rdtscp" : "=a" (low), "=d" (high), "=c" (aux));

\_\_asm\_\_ volatile("lfence" ::: "memory");

return (high << 32) | low;

}

Встраиваемый код с использованием intrinsic функций MSVC и GCC:

inline uint64\_t vi\_get\_tsc()

{

uint32\_t aux; // Будет выкинут оптимизатором

uint64\_t result = \_\_rdtscp(&aux);

\_mm\_lfence();

return result;

}

Для всех реализаций длительность вызова функции и дискретность измерения времени получились около 14ns.

* 1. Другой популярный в интернете совет сериализации инструкций в ассемблере – использовать CPUID:
     1. MASM:

vi\_get\_tsc PROC

push rbx

rdtscp

shl rdx, 32

or rax, rdx

mov r8, rax

xor eax, eax

xor ecx, ecx

cpuid

mov rax, r8

pop rbx

ret 0

vi\_get\_tsc ENDP

* + 1. GCC:

inline uint64\_t vi\_get\_tsc ()

{

unsigned long aux;

unsigned long long low, high;

\_\_asm\_\_ volatile("rdtscp" : "=a" (low), "=d" (high), "=c" (aux));

\_\_asm\_\_ volatile("xor %%rax, %%rax"::: "%rax");

\_\_asm\_\_ volatile("cpuid" ::: "%rax", "%rbx", "%rcx", "%rdx");

return (high << 32) | low;

}

* + 1. MSVC & GCC:

uint64\_t vi\_get\_tsc()

{

int32\_t cpuInfo[4];

uint32\_t aux;

const auto result = \_\_rdtscp(&aux);

\_\_cpuid(cpuInfo, 0);

return result;

}

Для всех реализаций длительность вызова функции и дискретность измерения времени в Windows и Linux получились около 32ns.

Но в случае сериализации с использованием CPUID я столкнулся с проблемой. Замеры длительности выполнения инструкции CPUID в обычном режиме (а не в Safe mode в котором я проводил все замеры) дали непонятные мне 350ns длительности и дискретности. Этот эффект повторялся на всех четырёх компьютерах, на которых я производил замеры (процессоры: i7-6700K@4.00GHz, Ryzen 5 2400G@3.60GHz, i7-4770T@2.50GHz, i7-10510U@1.80GHz).

You

Способы измерения коротких интервалов времени в программах на C++?

Copilot

Для измерения коротких интервалов времени в программах на **C++** существует несколько методов. Вот некоторые из них:

1. **Использование**std::chrono:
   * Стандартная библиотека C++ предоставляет классы std::chrono::steady\_clock и std::chrono::high\_resolution\_clock для измерения временных интервалов.
   * Пример использования:

**#include <chrono>**

**#include <iostream>**

**int main() {**

**auto start\_time = std::chrono::steady\_clock::now();**

**// Ваш код**

**auto end\_time = std::chrono::steady\_clock::now();**

**auto elapsed\_ns = std::chrono::duration\_cast<std::chrono::nanoseconds>(end\_time - start\_time);**

**std::cout << "Прошло " << elapsed\_ns.count() << " наносекунд\n";**

**}**

* + Обратите внимание, что точность этих часов зависит от операционной системы, и high\_resolution\_clock может быть более точным, но не всегда монотонным[1](https://ru.stackoverflow.com/questions/440326/%D0%9A%D0%B0%D0%BA-%D0%B8%D0%B7%D0%BC%D0%B5%D1%80%D0%B8%D1%82%D1%8C-%D1%81%D0%BA%D0%BE%D1%80%D0%BE%D1%81%D1%82%D1%8C-%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%8B-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D1%8B)[2](https://pro-prof.com/forums/topic/cplusplus-time-function).

1. **Инструкция**rdtsc:
   * Для очень маленьких интервалов времени можно использовать инструкцию процессора rdtsc (ReaD TicS Counter), которая возвращает количество тактов процессора.
   * Примерный код (для Windows, MS VC++):

**#include <intrin.h>**

**#include <iostream>**

**#include <thread>**

**template<typename F>**

**long long measure(F&& f) {**

**const auto N = 10;**

**long long results[N];**

**for (auto& r : results) {**

**std::this\_thread::yield();**

**\_\_asm xor eax, eax**

**\_\_asm cpuid**

**auto start\_time = \_\_rdtsc();**

**f();**

**auto end\_time = \_\_rdtsc();**

**r = end\_time - start\_time;**

**}**

**std::sort(results, results + N);**

**return results[N / 2];**

**}**

**int main() {**

**auto elapsed\_cycles = measure( {**

**// Ваш код**

**});**

**std::cout << "Прошло " << elapsed\_cycles << " тактов процессора\n";**

**}**

* + Пожалуйста, обратите внимание на условия использования rdtsc, такие как выполнение инструкции cpuid, отключение Hyper-Threading и другие[3](https://web.snauka.ru/issues/2017/05/83188).

Выберите подход, который лучше всего подходит для вашей конкретной задачи, и убедитесь, что он соответствует вашим требованиям точности и надежности. 😊