# Учебная практика

**Измерение времени**

2016-2017

# Введение

Программисту приходится довольно часто отслеживать время, например в игровых и мультимедийных приложениях, при разработке программ, работающих с аппаратной частью ПК, а также при проведении всевозможных тестов. Кроме того, нередко требуется отладить критичные ко времени исполнения фрагменты кода, для чего нужны «часы» с высокой разрешающей способностью.

**Можно использовать несколько видов измерения интервалов времени. Все они отслеживают сообщения от**

- **таймера или**
- **генератора импульсов.**

# Общие замечания

1. В Windows всегда одновременно выполняется несколько программ, которым выделяется время процессора, поэтому время выполнения одних и тех же фрагментов программ — величина непостоянная. Нельзя делать выводы о какой-то реализации алгоритма после одного тестирования.

   Если вы тестируете алгоритм, то поставьте его в цикл, выполнив его, например, тысячу раз, а потом вычислите среднее время.

2. При проведении вычислительных экспериментов необходимо учитывать, что при работе оптимизирующего компилятора время выполнения приложения при первых запусках значительно больше, чем при последующих – идет оптимизация кода Just In Time (JIT) в соответствующих средах и языках

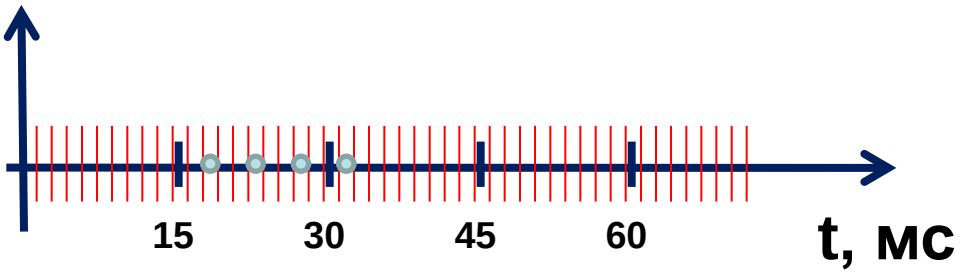
# Общие замечания

3) Время всегда измеряется с погрешностью.

У разных инструментов измерения точность разная.

**Таймер** вызывается периодически с определенной частотой, минимальный период 15,6 мс, поэтому и результат измерения времени таймером кратен 15,6 мс.

**Генератор тактовой частоты** вырабатывает импульсы в соответствии с частотой CPU



# Общие замечания

1. Функции отсчета времени в языке программирования.
2. Функции из библиотеки Windows
3. Можно снимать показания со счетчика импульсов процессора непосредственно

**Функции из библиотек языков программирования**

**Windows API**

**Ассемблер**

Потеря функциональности при переходе с более низкого уровня на более высокий. Каждый «слой» API создаётся для облегчения выполнения некоторого стандартного набора операций. Но при этом реально затрудняется, либо становится принципиально невозможным выполнение некоторых других операций, которые предоставляет более низкий уровень API.

# Примеры инструментов измерения времени

1. Clock
2. Time
3. timeGetTime
4. GetTickCount
5. Функции API
6. RDTCS

Различаются

1. Размерностью переменной и отсюда - точностью
2. Временем до сброса (продолжительностью)
3. Точкой начала отсчета

И т.д.

# ИНСТРУМЕНТЫ ИЗМЕРЕНИЯ ВРЕМЕНИ В C++

# clock

Вычисляет время, затраченное вызывающим процессом, в миллисекундах

```
clock_t clock( void );
```

Требуется подключить `time`

```
#include<time.h>
```

Интервал времени примерно кратен

`1/CLOCKS_PER_SEC` секунд.

# clock

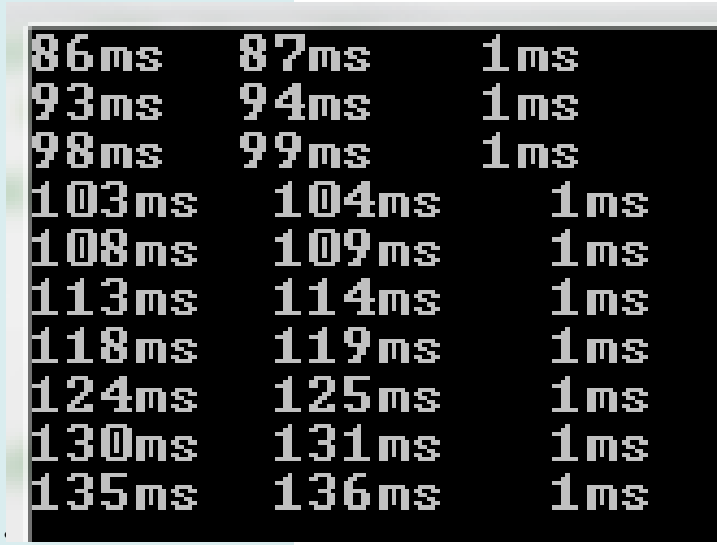
```
#include<ctime>
…
temp1=double(clock( )); // Функция clock() возвращает процессорное
                        //  время в единицах,
                          // которые зависят от реализации языка.
Bubble_Sort(B,n);                 //Вызов функции
temp2=double(clock( ));

double t = ((temp2- temp1) /CLOCKS_PER_SEC); // время в секундах

// символическая константа CLOCKS_PER_SEC используется для
// перевода значения, возвращаемого функцией clock( ), в секунды;
```

# Пример использования clock()

```
#include<iostream>
#include<time.h>
using namespace std;
void main(){
//Измерение времени на C++  clock()
        double time1, time2;
        for(int i=0;i<10;i++) {
                time1=clock();
                while (clock() == time1);
                        time2=clock();
                cout<<time1<<"ms  "<<time2<<"ms  "
                        <<time2-time1<<"ms"<<endl;
        }
        cout<<"  "<<endl; system ("PAUSE");
}
```




Минимальный интервал - миллисекунда

# time, _time32, _time64

Возвращает системное время

```
time_t time( time_t *timer );
_ _time32_t _time32( _ _time32_t *timer );
_ _time64_t _time64( _ _time64_t *timer );
```

Возвращает время как секунды, прошедшие с 0 часов 1 января 1970 г. или -1 в случае ошибки

Требуется подключить `time`

```
#include<time.h>
```

# Заголовочный файл time.h

```
#ifndef _TIME32_T_DEFINED
typedef _W64 long __time32_t;   /* 32-bit time
value*/
#define _TIME32_T_DEFINED
#endif  /* _TIME32_T_DEFINED */

#ifndef _TIME64_T_DEFINED
typedef __int64 __time64_t;
/* 64-bit time value*/
#define _TIME64_T_DEFINED
#endif  /* _TIME64_T_DEFINED
```

```
#ifndef _TIME_T_DEFINED
#ifdef _USE_32BIT_TIME_T
typedef __time32_t time_t;      /* time
   value */
#else  /* _USE_32BIT_TIME_T */
typedef __time64_t time_t;      /* time
   value */
#endif  /* _USE_32BIT_TIME_T */
#define _TIME_T_DEFINED         /*
   avoid multiple def's of time_t */
#endif  /* _TIME_T_DEFINED */
```

# time, _time32, _time64

Округленные продолжительность и точность

73 года ± 1 секунда

```
time_t StartTime, ElapsedTime;
StartTime = time(NULL);
<  код  >
ElapsedTime= time(NULL) – StartTime;
```

# Пример. Сравнение time и clock

```
// Функция С-библиотеки времени выполнения
double time1,time2;
time_t StartTime1, ElapsedTime1;
for(int i=0;i<10;i++) {
        time1=clock();
        StartTime1 = time(NULL);
        while (time(NULL) == StartTime1);
                time2=clock();
        ElapsedTime1= time(NULL) - StartTime1;
cout<<time1<<"ms  "<<time2<<"ms   "<<
        time2-time1<<"ms"<<endl;
        cout<<StartTime1<<"  "<<ElapsedTime1<<endl;
}
```

# Пример. Сравнение time и clock

```
clock   85ms  882ms  797ms
time    1317625125          1
clock   896ms 1882ms  986ms
time    1317625126          1
clock   1894ms  2882ms  988ms
time    1317625127          1
clock   2906ms  3882ms  976ms
time    1317625128          1
clock   3890ms  4889ms  999ms
time    1317625129          1
clock   4897ms  5882ms  985ms
time    1317625130          1
clock   5890ms  6881ms  991ms
time    1317625131          1
clock   6895ms  7881ms  986ms
time    1317625132          1
clock   7891ms  8881ms  990ms
time    1317625133          1
clock   8895ms  9881ms  986ms
time    1317625134          1
Для продолжения нажмите люб
```

# ПРОЦЕДУРЫ WINDOWS

# Windows API

Windows API был изначально спроектирован для использования в программах, написанных на языке Си или C++.

Работа через Windows API — это наиболее близкий к системе способ взаимодействия с ней из прикладных программ.

Версии

**Win16** — первая версия **Windows API** для 16-разрядных версий **Windows**.

**Win32** — 32-разрядный **API** для современных версий **Windows**. Популярная версия.

Базовые функции этого **API** реализованы в динамически подключаемых библиотеках **kernel32.dll** и **advapi32.dll**;

базовые модули графического интерфейса пользователя — в **user32.dll** и **gdi32.dll**.

**Win64** — 64-разрядная версия **Win32**, содержащая дополнительные функции для использования на 64-разрядных компьютерах.

# timeGetTime

## Мультимедийный таймер Windows

Функция `timeGetTime` возвращает системное время в миллисекундах. Системное время – время, прошедшее с момента запуска `Windows`.

```
DWORD timeGetTime(void);
```

Точность от ± 5 миллисекунд. Счетчик сбрасывается каждые 2^32 мс ≈ 49.71 дней.

```
DWORD StartTime, ElapsedTime;
StartTime = timeGetTime();
ElapsedTime = timeGetTime() – StartTime;
```

Требуется подключить **`Mmsystem.h`**, включает **`Windows.h`**

# GetTickCount

Возвращает количество миллисекунд, прошедших от момента старта системы

**Синтаксис**

```
DWORD WINAPI GetTickCount(void);
```

Точность ограничена точностью системного таймера, обычно 10-16 миллисекунд. Сброс примерно через 49,7 дней.

```
ULONGLONG WINAPI GetTickCount64(void);
```

Больший период до сброса

**Требуется подключение**

`Winbase.h` (включает в себя `Windows.h`),

библиотеки `Kernel32.lib`,

DLL `Kernel32.dll`

# Пример. Сравнение GetTickCount и clock

```
DWORD StartTime5, ElapsedTime5;
for(int i=0;i<10;i++) {
  time1=clock();
  StartTime5 = GetTickCount();
  while (GetTickCount() == StartTime5);
    time2=clock();
  ElapsedTime5 = GetTickCount() - StartTime5;
  cout << "clock  " << time1 << "ms " << time2 << "ms "
       << time2- time1<<"ms"<<endl;
  cout << "GetTickCount = " << StartTime5 << "   " <<
       ElapsedTime5 << endl;
```

# Результат сравнения GetTickCount и clock

```
clock  56ms 58ms 2ms
GetTickCount = 389138253  16
clock  72ms 74ms 2ms
GetTickCount = 389138269  16
clock  88ms 89ms 1ms
GetTickCount = 389138285  15
clock  103ms 105ms 2ms
GetTickCount = 389138300  16
clock  115ms 120ms 5ms
GetTickCount = 389138316  15
clock  135ms 136ms 1ms
GetTickCount = 389138331  16
clock  150ms 152ms 2ms
GetTickCount = 389138347  16
```

# Генератор тактовой частоты

Генератор тактовой частоты (генератор тактовых импульсов) генерирует электрические импульсы заданной частоты для синхронизации различных процессов в цифровых устройствах

***Тактовые импульсы часто используются как эталонная величина — считая их количество, можно измерять временные интервалы.***

В микропроцессорной технике один тактовый импульс, как правило, соответствует одной атомарной операции. Обработка одной инструкции может производиться за один или несколько тактов работы микропроцессора, в зависимости от архитектуры и типа инструкции. Частота тактовых импульсов определяет скорость вычислений.

# Функции Windows

В компьютере есть генератор импульсов и 64-хразрядный счетчик циклов тактовой частоты, показания которого считываются программно

Функции Windows API:

`QueryPerformanceCounter` возвращает количество импульсов, прошедшее с момента включения компьютера

`QueryPerformanceFrequency` возвращает частоту генератора импульсов (число тиков в секунду).

# QueryPerformanceCounter

```
BOOL WINAPI QueryPerformanceCounter( __out  LARGE_INTEGER
*lpPerformanceCount );
```

Параметр - указатель на переменную, содержащую текущее значение тиков

При успешном выполнении возвращает не 0, иначе - 0

Требуется подключение заголовка `Winbase.h` (включает `Windows.h`) и `Kernel32.dll`, `Kernel32.lib`

Многопроцессорные компьютеры:

On a multiprocessor computer, it should not matter which processor is called. However, you can get different results on different processors due to bugs in the basic input/output system (BIOS) or the hardware abstraction layer (HAL). To specify processor affinity for a thread, use the `SetThreadAffinityMask` function.

# Тип union – совмещение переменных по адресам

```
#include<iostream>
using namespace std;
void main() {
  typedef union __union {
            int i;
            float a;
     };
    __union _union;
   _union.i=1;     // печать
   _union.a=1;     // печать
   _union.i=127;   // печать
   _union.a=127;   // печать
   _union.a=132;   // печать
system("pause");
}
```

```
int  i;
float  a;
```

**00000000 00000000 00000000 00000001**

```
// печать
cout<<"union.i = "<<_union.i<<"
   union.a = "<<_union.a << endl;
    for(int j = 0;j < 32;j++) {
    cout << (_union.i & 1);
    _union.i=_union.i>>1; }
    cout << endl;
system("pause");
}
```

# Тип union – совмещение переменных по адресам

Переменные `union` совмещены по адресам.

Одно и то же содержимое четырех байт интерпретируется по-разному для `int` и `float`

```
union.i = 1   union.a = 1.4013e-045
10000000000000000000000000000000
union.i = 1065353216   union.a = 1
00000000000000000000000111111100
union.i = 127   union.a = 1.77965e-043
11111110000000000000000000000000
union.i = 1123942400   union.a = 127
00000000000000001111111101000010
union.i = 1124335616   union.a = 132
00000000000000001000001100000010
Для продолжения нажмите любую клавишу . . .
```

# QueryPerformanceFrequency

```
BOOL WINAPI QueryPerformanceFrequency ( __out  LARGE_INTEGER
*lpFrequency );
```

Параметр – указатель на переменную, содержащую частоту процессора в тиках в секунду

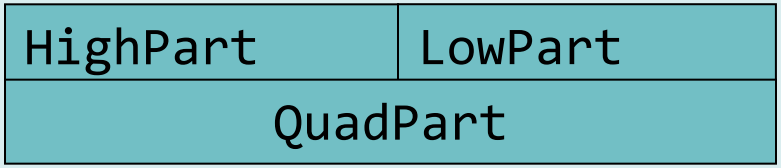
При успешном выполнении возвращает не 0, иначе 0

Требуется подключение заголовка **Winbase.h** (включает **Windows.h**) и **Kernel32.dll**, **Kernel32.lib**

# LARGE_INTEGER Union

```
typedef union _LARGE_INTEGER {
        struct {
                DWORD LowPart;
                LONG HighPart;
                } ;
        struct {
                DWORD LowPart;
                LONG HighPart;
                } u;
        LONGLONG QuadPart;
}
LARGE_INTEGER, *PLARGE_INTEGER;
```

Переменные совмещены по адресам




Объявляем переменную и указатель типа _LARGE_INTEGER

# LARGE_INTEGER Union

`LowPart` младшие 32 бита

`HighPart` старшие 32 бита

u `LowPart` младшие 32 бита переменной u.

u `HighPart` старшие 32 бита переменной u.

`QuadPart` знаковое целое 64 бита

Примерчик
( 4 + 4 бита):
0100 1001

`HighPart` = 4 `LowPart` = 9

`QuadPart` ≠ 49
`QuadPart` = $01001001_2$ = 73

Если компилятор поддерживает 64-битовые целые (например, MicroSoft поддерживает `__int64`. ), используем `QuadPart`.

```
__int64 TTint64;
LARGE_INTEGER TT; …
TTint64 = TT.QuadPart;
```

# Процедуры Windows

Если известна разница между результатами двух вызовов QueryPerformanceCounter,

Частоту CPU достаточно измерить один раз

```
QueryPerformanceFrequency (Frequency);
QueryPerformanceCounter(startt);
…
QueryPerformanceCounter(finishh);
dCount = finishh -  startt;
```

то время

```
dCount / Frequency * 1000000        // можем получить 0
dCount * 1000000 / Frequency        // микросекунды
dCount * 1000000000 / Frequency     // наносекунды
```

```
LARGE_INTEGER Fr, StT, FT, TT;
QueryPerformanceFrequency(&Fr);
time1=clock();
QueryPerformanceCounter(&StT);
// код
QueryPerformanceCounter(&FT);
time2=clock();
cout<<"Fr "<<Fr.HighPart<<"  "<<Fr.LowPart<<"  " <<Fr.QuadPart<<"
Hz"<<endl;
cout<<"FT  "<<FT.HighPart<<"  "<<FT.LowPart<<"  " <<FT.QuadPart<<"
Ticks"<<endl;
cout<<"StT "<<StT.HighPart<<"  "<<StT.LowPart<<"  " <<StT.QuadPart<<"
Ticks"<<endl;
TT.QuadPart=(FT.QuadPart-StT.QuadPart);
cout<<"TT "<<TT.HighPart<<"  "<<TT.LowPart<<"  " <<TT.QuadPart<<endl;
TT.QuadPart=TT.QuadPart*1000000000/Fr.QuadPart;
cout<<"TT "<<TT.HighPart<<"  "<<TT.LowPart<<" ns  " <<TT.QuadPart<<"
ns  "<<endl;
cout<<"clock"<<time1<<"ms"<<time2<<"ms"<<time2-time1<<" ms"<<endl;
```

# Результат

```
Fr 0   2467812   2467812 Hz
FT   222   1601383770   955084123482 Ticks
StT 222   1600637168   955083376880 Ticks
TT 0   746602   746602
TT 0   302536011 ns   302536011 ns
TTint64=302536011
TTlonglong=302536011
clock 58 ms   360 ms   302 ms
Для продолжения нажмите любую клавишу .
```

# АССЕМБЛЕР

# Регистры

**Регистр** — последовательное или параллельное логическое устройство, используемое для хранения n-разрядных двоичных чисел и выполнения преобразований над ними.

**Типичные операции**:

1. приём слова в регистр;
2. передача слова из регистра;
3. поразрядные логические операции;
4. сдвиг слова влево или вправо на заданное число разрядов;
5. преобразование последовательного кода слова в параллельный и обратно;
6. установка регистра в начальное состояние (сброс).

# Регистры CPU

**Регистр процессора** — блок ячеек памяти, образующий сверхбыструю оперативную память (СОЗУ) внутри процессора; используется самим процессором и большой частью недоступен программисту

**Классификация по назначению**

1. ***аккумулятор*** — используется для хранения промежуточных результатов арифметических и логических операций и инструкций ввода-вывода;
2. ***флаговые*** — хранят признаки результатов арифметических и логических операций;
3. ***общего назначения*** — хранят операнды арифметических и логических выражений, индексы и адреса;
4. ***индексные*** — хранят индексы исходных и целевых элементов массива;
5. ***указательные*** — хранят указатели на специальные области памяти (указатель текущей операции, указатель базы, указатель стэка);
6. ***сегментные*** — хранят адреса и селекторы сегментов памяти;
7. ***управляющие*** — хранят информацию, управляющую состоянием процессора, а также адреса системных таблиц.

# Регистры CPU

При выборке из памяти очередной команды она помещается в ***регистр команд***, к которому программист обратиться не может.

Имеются также регистры, которые в принципе программно доступны, но обращение к ним осуществляется из программ операционной системы, например***, управляющие регистры*** и ***теневые регистры дескрипторов сегментов***. Этими регистрами пользуются в основном разработчики операционных систем.

***Регистры общего назначения*** (РОН) используются без ограничения в арифметических операциях, но имеющие определенные ограничения, например в строковых

**Специальные регистры** содержат данные, необходимые для работы процессора — смещения базовых таблиц, уровни доступа и т. д.

Часть специальных регистров принадлежит *устройству управления*, которое управляет процессором путём генерации последовательности микрокоманд.

Доступ к значениям, хранящимся в регистрах, в несколько раз быстрее, чем доступ к ячейкам оперативной памяти, но объём оперативной памяти намного превосходит суммарный объём регистров *общего назначения/данных*

# РОН

**Регистры данных** — служат для хранения промежуточных вычислений.

RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8 — R15 — ***64-битные***

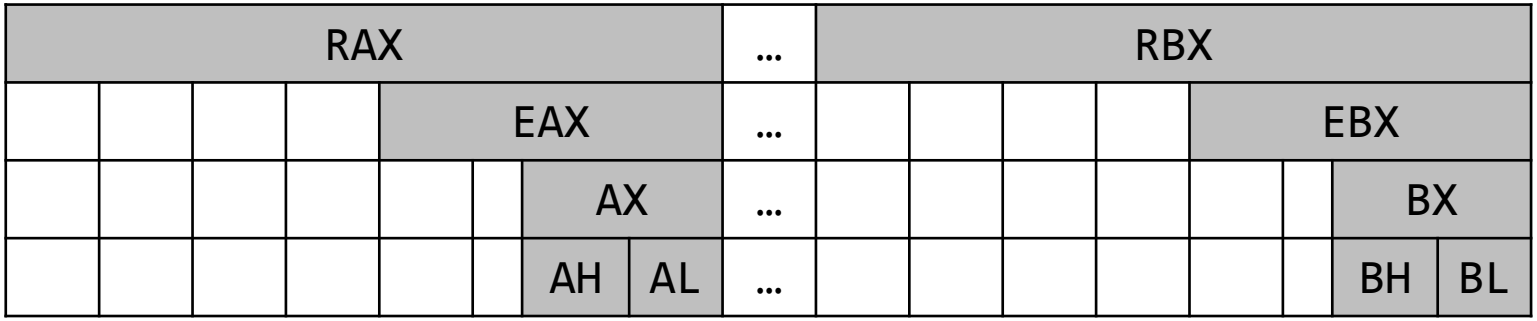
EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, R8D — R15D — ***32-битные*** (extended AX)

AX, CX, DX, BX, SP, BP, SI, DI, R8W — R15W — ***16-битные***

AH, AL, CH, CL, DH, DL, BH, BL, SPL, BPL, SIL, DIL, R8B — R15B — ***8-битные*** (половинки 16-ти битных регистров)

например, AH — high AX — старшая половинка 8 бит

AL — low AX — младшая половинка 8 бит




RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, Rx, RxD, RxW, RxB, SPL, BPL, SIL, DIL доступны только в 64-битном режиме работы процессора

# Такты 32-х разрядного процессора (в тактах CPU)

Период ___ _____, точность ± 0,001 мкс (процессор с тактовой частотой 3,4 ГГц)

Ассемблерные вставки начинаются с директивы _asm

**RDTSC (ReaD Time Stamp Counter)** — ассемблерная инструкция для платформы x86, читающая **счётчик TSC (Time Stamp Counter)** и возвращающая в регистре EAX 32-битное количество тактов с момента последнего сброса процессора.

В процессорах Intel счетчик TSC не зависит от использования технологий энергосбережения и увеличивается на единицу каждый такт, независимо от того, работал ли процессор или находился в состоянии сна.

В некоторых реализациях счетчики TSC могут иметь синхронные значения на многоядерной системе.

**? От чего зависит точность?**
**? От чего зависит период (до сброса) ?**

# Команды ассемблера

| | |
|---|---|
| **`mov имя_перем, регистр`** | передает данные из регистра в переменную |
| `sub регистр, имя_перем` | разность между значениями регистра и переменной |
| `Функция RDTSC` | снимает данные счетчика тактов |

```
for(int i = 0; i < 10; i++)
      {     time1=clock();
            DWORD StartTime3, ElapsedTime3;
            _asm
            {     RDTSC
                  mov StartTime3, eax
            }
      //<    код   >
            _asm
            {     RDTSC
                  sub eax, StartTime3
                  mov ElapsedTime3, eax
            }
time2=clock();
cout<<"32-CPU  "<<StartTime3<<"  "<<ElapsedTime3<<"  ";
      cout<<"clock   "<<time1<<"ms  "<<time2<<"ms
"<<time2-time1<<"ms"<<endl;
      }     cout<<"  "<<endl;
```

Снятие показания счетчика тактов. Результат – в регистр eax (eax =TSC)

Из регистра eax – в переменн StartTime3 (StartTime3=eax )

Вычисляется разность между eax и StartTime3. Результат – в регистр eax (eax=eax-StartTime3)

Из регистра eax – в переменную ElapsedTime3 (ElapsedTime3 = eax )

# Результат

```
32-CPU   3114060569   46   clock   84ms   84ms    0ms
32-CPU   3143483892   66   clock   95ms   95ms    0ms
32-CPU   3167845666   66   clock   105ms   105ms    0ms
32-CPU   3195895118   112   clock   116ms   116ms    0ms
32-CPU   3221744198   60   clock   126ms   126ms    0ms
32-CPU   3245429491   152   clock   136ms   136ms    0ms
32-CPU   3271161495   96   clock   146ms   146ms    0ms
32-CPU   3296708733   66   clock   156ms   156ms    0ms
32-CPU   3319356581   86   clock   165ms   165ms    0ms
32-CPU   3344233322   76   clock   175ms   175ms    0ms
```

# Такты 64-х разрядного процессора (в тактах CPU)

172 года; точность ± 0,001 мкс (процессор с тактовой частотой 3,4 ГГц)

**RDTSC (ReaD Time Stamp Counter)** — ассемблерная инструкция для платформы x86, читающая счётчик **TSC (Time Stamp Counter)** и возвращающая в регистрах EDX:EAX 64-битное количество тактов с момента последнего сброса процессора.

```
_ _int64 StartTime4, EndTime4;
  for(int i=0;i<10;i++)
  {
       time1=clock();
       _asm
       {      RDTSC
              mov DWORD PTR StartTime4, eax
              mov DWORD PTR StartTime4+4, edx
       }
//<   код   >
       _asm
       {      RDTSC
              mov DWORD PTR EndTime4, eax
              mov DWORD PTR EndTime4+4, edx
       }
       time2=clock();
cout<<"CPU-64  "<<StartTime4<<"  "<<EndTime4<<"  "<<EndTime4-
StartTime4<<" Tick “;
cout<<"clock  "<<time1<<"ms "<<time2<<"ms "<<time2-time1<< "ms"
<< endl;
 }     cout<<"  "<<endl;;
```

# Результат

```
CPU-64  1936315881477 52  1936315881477 98  46 Tick clock  78ms  78ms   0ms
CPU-64  1936316371581 03  1936316371581 69  66 Tick clock  97ms  97ms   0ms
CPU-64  1936316811699 63  1936316811701 05  142 Tick clock  114ms  114ms   0ms
CPU-64  1936317313288 26  1936317313289 94  168 Tick clock  134ms  134ms   0ms
CPU-64  1936317802714 46  1936317802715 42  96 Tick clock  154ms  154ms   0ms
CPU-64  1936318314187 43  1936318314192 29  486 Tick clock  174ms  174ms   0ms
CPU-64  1936318821277 81  1936318821279 23  142 Tick clock  194ms  194ms   0ms
CPU-64  1936319268147 31  1936319268150 82  351 Tick clock  212ms  212ms   0ms
CPU-64  1936319751370 66  1936319751371 72  106 Tick clock  231ms  231ms   0ms
CPU-64  1936320227987 77  1936320227989 45  168 Tick clock  250ms  250ms   0ms
```

# Вывод

1) Наиболее точные измерения получаем при прямом обращении к счётчику тактов **TSC (Time Stamp Counter)** и при использовании функций Windows API

```
QueryPerformanceFrequency();
QueryPerformanceCounter();
```

Удобнее использовать функции более высокого уровня

2) Точно измерить «чистое» время выполнения кода невозможно, т.к. процессор одновременно выполняет другие задачи