Национальный исследовательский ядерный университет «МИФИ» (Московский Инженерно–Физический Институт)
Кафедра №42 «Криптология и кибербезопасность»

Лабораторная работа №2:
«**Компьютер и время**»

Антон Гатченко Б22-525
2025 г.

Используемая рабочая среда:

- Процессор - AMD Ryzen 5 5600H (laptop), 6c/12t
- Оперативная память – DDR4 16 ГБ
- ОС - Windows 10 Pro 22H2 19045.4780, 64 bit
- IDE – GCC/G++ 13.1, OpenMP 201511

Теоретическая часть:

1. `std::chrono::time_point` - конкретный момент времени, привязанный к определённым часам.

Внутри `std::chrono::time_point` хранится значение типа `Duration`, которое представляет собой количество тиков, прошедших с эпохи (начала отсчёта) часов. Эпоха зависит от типа часов.

2. В Linux RTC представлены через устройство `/dev/rtc` или `/dev/rtc0`.

Команды для работы с RTC:

- `hwclock --show` для вывода времени;
- `hwclock --set --date="..."` для установки времени;
- `hwclock --hctosys` для синхронизации системного времени с RTC;
- `hwclock --systohc` для синхронизации RTC с системным временем.

3. (и 4) Типы часов, предоставляемые операционной системой (в C++):

- `std::chrono::system_clock` - системные часы реального времени, время может быть изменено, например синхронизацией времени;
- `std::chrono::steady_clock` - монотонные часы, которые не могут быть отрегулированы назад во времени;
- `std::chrono::high_resolution_clock` - часы с максимально возможным разрешением, зачастую это псевдоним для `system_clock` или `steady_clock`.

5. Следующей високосной секунды, видимо, не будет.

Последняя была добавлена в ночь с 31 декабря 2016 г. на 1 января 2017 года. Согласно графику на рис. 1, в последние годы величина UT1 – UTC растет значительно медленнее, чем раньше, так что пока нет необходимости в вводе дополнительной секунды.

«В связи с уменьшением величины положительного хода UT1 - UTC, возможно, что "високосная" секунда так и не будет введена до 2035 года, до которого, согласно готовящемуся решению Всемирной конференции радиосвязи (WRC-23), должна применяться действующая шкала UTC, определённая в соответствии с Рекомендацией МСЭ-R TF.460-6.» - glonass-iac

«В ноябре 2022 года на очередной Генеральной конференции по мерам и весам (ГКВМ) было предложено отказаться от дополнительных секунд к 2035 году.» - wikipedia

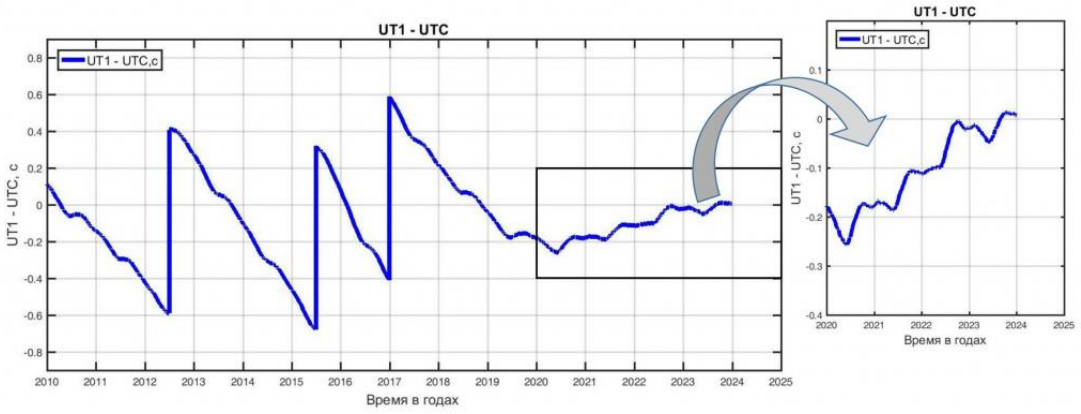

Рисунок 1 - Изменение разницы UT1 и UTC с 2010 по 2024 год (Источник - https://glonass-iac.ru/news/news_gnss/4155)

6. Инструкция RDTSC для использования в C++ доступна в компиляторах MSVC и GCC/Clang с помощью функции `__rdtsc()`. Пример использования `__rdtsc()` находится в Приложении 7.
7. Чтобы перевести значение TSC (Time-Stamp Counter) в секунды, нужно знать частоту TSC (на современных системах не зависит от текущей частоты процессора). Для моего ПК это значение составило ~3.3 ГГц, что совпадает с базовой частотой процессора. Пример вычисления коэффициента и замеров с его помощью находится в Приложении 8.

Нативно на Windows можно использовать QPC (QueryPerformanceCounter), который официально рекомендуют использовать (и не рекомендуют использовать `rdtsc`)

«We strongly discourage using the RDTSC or RDTSCP processor instruction to directly query the TSC because you won't get reliable results on some versions of Windows, across live migrations of virtual machines, and on hardware systems without invariant or tightly synchronized TSCs. Instead, we encourage you to use QPC to leverage the abstraction, consistency, and portability that it offers.» - microsoft

Пример использования QPC находится в Приложении 9.

Практическая часть:

Так как на Windows нельзя получить доступ к RTC, измерения, связанные с ними, проводились в WSL (дистрибутив Ubuntu 22.04 LTS).

Доступные типы часов операционной системы:

В C: `CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_THREAD_CPUTIME_ID, CLOCK_PROCESS_CPUTIME_ID, CLOCK_REALTIME_COARSE`.

`CLOCK_REALTIME_COARSE` — это оптимизированный тип часов, предназначенный для быстрого получения грубого значения текущего времени. Он особенно полезен в высокопроизводительных приложениях, где точность времени не является критической. Значения времени кэшируются в памяти процессора и не требуют обращения к аппаратным часам или системным вызовам. Возможно, должно работать только на Linux? Однако при выполнении кода на Windows ошибок не возникало.

В C++: `std::chrono::system_clock, std::chrono::steady_clock, std::chrono::high_resolution_clock`.

К сожалению, `chrono` не предоставляет возможности узнать разрешение через какие-либо методы(.

Таблица 1 - Измерение разрешения и погрешностей таймеров

Имя таймера	**Точность, нс (по системному вызову)**	**Разрешение, нс (экспериментально)**	**Погрешность разрешения, нс (средняя абсолютная)**	**Погрешность разрешения, нс (среднеквадратичная)**
Часы реального времени (RTC)	-	1e9	0	0
`CLOCK_REALTIME`	100	100.4	0.85	102.6
`CLOCK_MONOTONIC`	100	100.5	0.97	114.8
`CLOCK_THREAD_CPUTIME_ID`	15625000	15625000	0	0
`CLOCK_PROCESS_CPUTIME_ID`	15625000	15625000	0	0
`CLOCK_REALTIME_COARSE`	15625000	992192	15391.65	81929.12
`std::chrono::system_clock`	-	100.4	0.38	77.9
`std::chrono::steady_clock`	-	100.4	0.39	91.4
`std::chrono::high_resolution_clock`	-	100.5	0.51	134.9
RDTSC	-	9.7	1.02	28.6
QPC	-	100	0.97	100.2

Таблица 2 - Измерение времени инициализации и возврата ответа

Имя таймера	**Время инициализации**, **нс**	**Время возврата ответа**, **нс**
Часы реального времени (RTC)	-	
CLOCK_REALTIME	32	0.1
CLOCK_MONOTONIC	45	0.1
CLOCK_THREAD_CPUTIME_ID	60	51
CLOCK_PROCESS_CPUTIME_ID	83	47
CLOCK_REALTIME_COARSE	0.5	
RDTSC	8.98	
QPC	20.39	

Из-за низкого разрешения значения для RTC не получилось измерить. Значения для RDTSC и QPC получены только как сумма времени инициализации и возврата ответа, так как их начальные значения отличаются от всех остальных, из-за чего таким методом нельзя получить верные данные по этим временам отдельно.

Замеры в таблице 2 могут быть (и скорее всего так оно и есть) довольно неточными, поскольку погрешности не дают найти конкретное решение системы (из-за них выходит, что система не имеет решения). Также некоторые часы отличаются по своему начальному значению, приходится считать это отличие экспериментально, что влияет на точность измерений. С оригинальным выводом программы по замеру этих времен можно ознакомиться в Приложении 6.

Диаграммы по таблице 2:

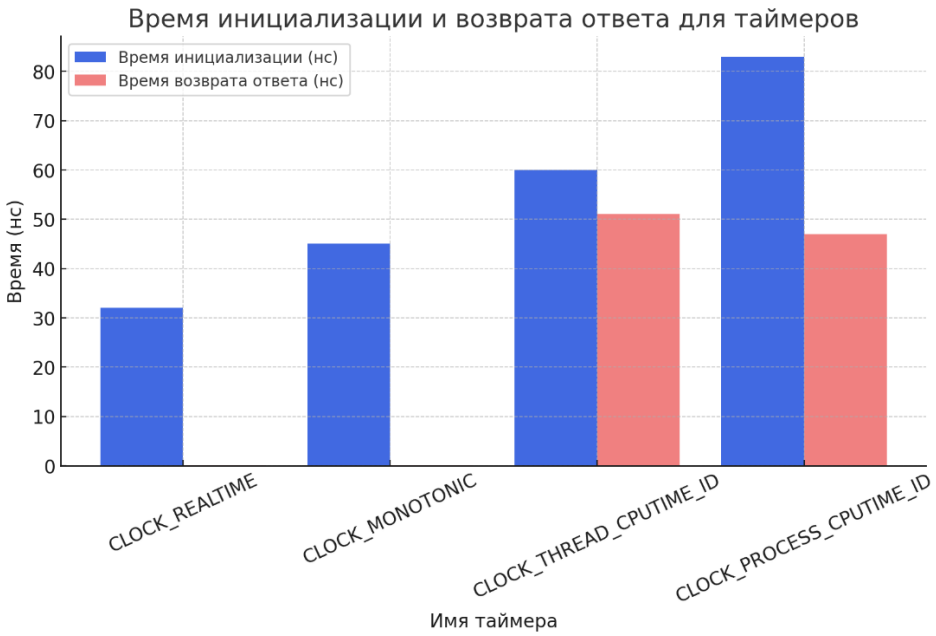

Рисунок 2 - время инициализации и возврата ответа (раздельные)

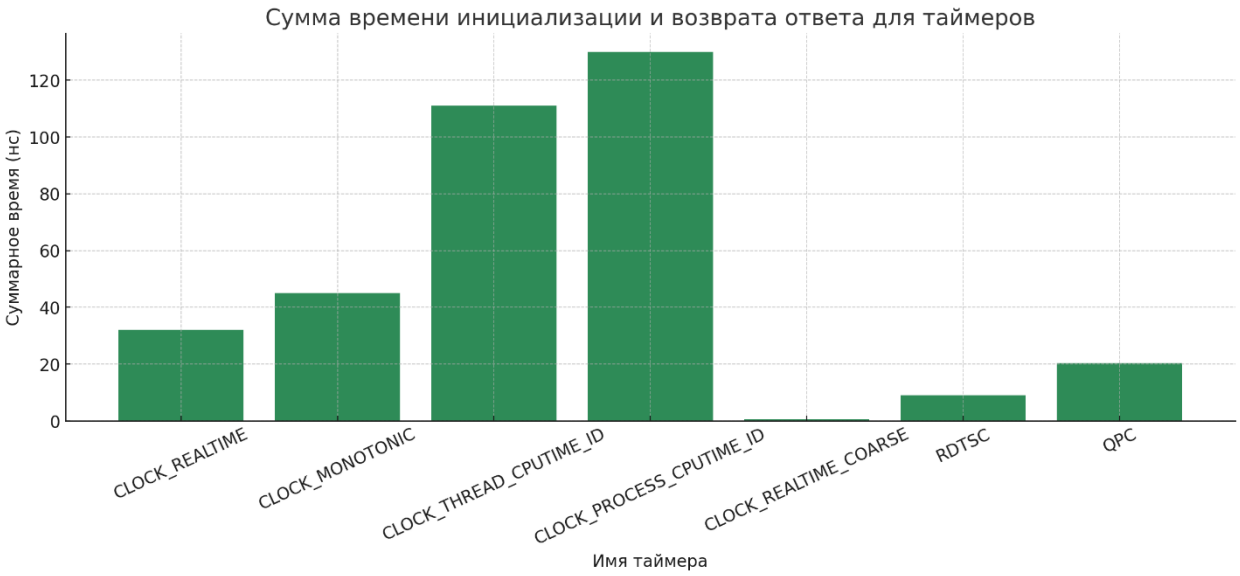

Рисунок 3 - время инициализации и возврата ответа (суммарные)

Заключение:

В ходе данной лабораторной работы были изучены различные типы часов, доступные в операционных системах Windows и Linux, а также проведены измерения их характеристик, таких как разрешение, погрешности и время инициализации и возврата ответа. По результатам этих измерений были построены таблицы 1 и 2 и диаграммы.

На Windows доступны `CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_THREAD_CPUTIME_ID, CLOCK_PROCESS_CPUTIME_ID, CLOCK_REALTIME_COARSE` , `std::chrono::system_clock, std::chrono::steady_clock, std::chrono::high_resolution_clock, RDTSC, QPC.`

На Linux также доступны эти часы за исключением QPC, и есть доступ к RTC.

RTC обладает крайне низким разрешением по сравнению с остальными часами (1 секунда, что в 10^7 раз больше чем, например, у `CLOCK_REALTIME`), поэтому для точных измерений он совершенно не подходит.

`CLOCK_REALTIME` и `CLOCK_MONOTONIC` показали хорошее разрешение около 100 нс с малой погрешностью < 1 нс, а также приемлемыми значениями времени инициализации около 30 – 45 нс, и временем возврата 0.1 нс. Их вполне можно использовать для замеров выполнения производительных программ. Лучше отдать предпочтение `CLOCK_MONOTONIC` из-за того, что он гарантирует неизменение назад во времени.

Если требуется измерить чистое процессорное время выполнения кода, `CLOCK_THREAD_CPUTIME_ID` и `CLOCK_PROCESS_CPUTIME_ID` могут быть предпочтительнее остальных вариантов, однако нужно учитывать их разрешение в 15.625 мс, что накладывает ограничения на измерение коротких частей программы. Их время инициализации примерно в 2 раза выше, чем у `CLOCK_REALTIME` и `CLOCK_MONOTONIC` (60 и 83 нс соответственно), время возврата около 50 нс.

`CLOCK_REALTIME_COARSE` показал разрешение в 1 мс, и самую высокую погрешность измерения, 15 мкс и 82 мкс средняя абсолютная / среднеквадратичная погрешности соответственно, однако относительно разрешения это приемлемый результат. Время инициализации и ответа в сумме – 0.5 нс, крайне быстро из-за кэширования на стороне процессора. Может быть отличным вариантом для измерения не самых быстрых частей программы.

Системные часы из C++, `std::chrono::system_clock, std::chrono::steady_clock, std::chrono::high_resolution_clock,` показали разрешение около 100 нс с низкой погрешностью, 0.4 – 0.5 нс / 78 – 135 нс. Поскольку они основаны на `CLOCK_REALTIME` и `CLOCK_MONOTONIC`, это ожидаемый результат, и к ним применяется все вышесказанное об этих часах.

RDTSC показал отличные результаты, самое высокое разрешение около 10 нс с низкими погрешностями 1 и 29 нс и хорошим временем инициализации и возврата ответа, суммарно 9 нс. Это предпочтительный вариант для измерения высокопроизводительных и крайне быстрых частей программы. Однако он может давать некорректные результаты на многоядерных системах.

QPC показал результаты, схожие с `CLOCK_REALTIME` и `CLOCK_MONOTONIC` – разрешение 100 нс, погрешности 1 / 100 нс, и обошел их по времени инициализации и возврата ответа – 20 нс в сумме против 30 – 45 у них. Несколько разочаровывающий результат, учитывая, что Microsoft рекомендует использовать QPC вместо RDTSC, что не представляется возможным из-за

серьезных отличий в разрешении – RDTSC на порядок точнее. Для приложений под Windows QPC может быть предпочтительнее CLOCK_REALTIME и CLOCK_MONOTONIC, но если необходима кроссплатформенность, то их можно использовать, в общем-то ничего не теряя.

Подытоживая данные замеры, я бы использовал RDTSC в случае замеров для коротких частей однопоточной программы, где нужна максимальная точность. Для многопоточных программ стоит присмотреться к CLOCK_THREAD_CPUTIME_ID и CLOCK_PROCESS_CPUTIME_ID. В остальных случаях можно использовать CLOCK_MONOTONIC. RTC я бы не использовал.

Приложение:

1. Исходный код программы с измерением параметров системных таймеров в C++:

```
#include <iostream>
#include <algorithm>
#include <chrono>
#include <vector>
#include <cmath>

#define MAX_ITERATIONS (int) 2e8

using namespace std::chrono;
using std::vector;

vector<double> measure_time(auto clock_func){
    vector<double> intervals(MAX_ITERATIONS);
    int count = 0;
    auto previous_time = clock_func();

    for (int i = 0; i < MAX_ITERATIONS; ++i){
       auto current_time = clock_func();

       if (current_time != previous_time){
          auto delta = duration_cast<nanoseconds>(current_time -
previous_time).count();
          intervals[count++] = static_cast<double>(delta);
          previous_time = current_time;
       }
    }
    intervals.resize(count);
    return intervals;
}

void measure_deviations(const vector<double> &intervals){
    double sum = 0.0, mean = 0.0, variance = 0.0, mean_absolute_deviation =
0.0;

    // std::cout << *std::ranges::max_element(intervals) << std::endl;

    for (double interval: intervals){
       sum += interval;
    }
    mean = sum / intervals.size();

    for (double interval: intervals){
       mean_absolute_deviation += abs(interval - mean);
    }
    mean_absolute_deviation /= intervals.size();

    for (double interval: intervals){
       variance += pow(interval - mean, 2);
    }
    variance /= intervals.size();
    double standart_deviation = std::sqrt(variance);
```

```
    std::cout << "Resolution: " << mean << " ns\n";
    std::cout << "mean_absolute_deviation: " << mean_absolute_deviation << "
ns\n";
    std::cout << "standart_deviation: " << standart_deviation << " ns\n";
}

int main(){
    vector<double> intervals = measure_time(system_clock::now);
    measure_deviations(intervals);

    intervals = measure_time(steady_clock::now);
    measure_deviations(intervals);

    intervals = measure_time(high_resolution_clock::now);
    measure_deviations(intervals);
    return 0;
}
```

2. Исходный код программы с измерением параметров системных таймеров в C:

```
#include <algorithm>
#include <iostream>
#include <cstdio>
#include <ctime>
#include <cmath>
#include <vector>

#define MAX_ITERATIONS (int) 1e7

using std::vector;

vector<double> measure_time(const int &clock){
    struct timespec previous_time, current_time;
    vector<double> intervals(MAX_ITERATIONS);
    int count = 0;

    clock_gettime(clock, &previous_time);

    for (int i = 0; i < MAX_ITERATIONS; ++i){
       clock_gettime(clock, &current_time);

       if (current_time.tv_sec != previous_time.tv_sec ||
current_time.tv_nsec != previous_time.tv_nsec){
          double delta = (current_time.tv_sec - previous_time.tv_sec) * 1e9
+
                         (current_time.tv_nsec - previous_time.tv_nsec);
          intervals[count++] = delta;
          previous_time = current_time;
       }
    }
    intervals.resize(count);
    return intervals;
}

void measure_deviations(const vector<double> &intervals){
```

```
    double sum = 0.0, mean = 0.0, variance = 0.0, mean_absolute_deviation =
0.0;

    for (const double &interval: intervals){
       sum += interval;
    }
    mean = sum / intervals.size();

    for (const double &interval: intervals){
       mean_absolute_deviation += fabs(interval - mean);
    }
    mean_absolute_deviation /= intervals.size();

    for (const double &interval: intervals){
       variance += pow(interval - mean, 2);
    }
    variance /= intervals.size();
    double standard_deviation = sqrt(variance);

    printf("Resolution: %.2f ns\n", mean);
    printf("Mean Absolute Deviation: %.2f ns\n", mean_absolute_deviation);
    printf("Standard Deviation: %.2f ns\n", standard_deviation);
}

void get_clock_resolution(const int &clock){
    struct timespec resolution;

    if (clock_getres(clock, &resolution) == 0){
       std::cout << "Resolution of " << clock << ": "
             << resolution.tv_sec << " seconds, "
             << resolution.tv_nsec << " nanoseconds\n";
    }
}

void measure_initialization_and_return(const int &clock1, const int
&clock2){
    struct timespec start_time, end_time;
    double total_time11 = 0.0, total_time22 = 0.0, total_time12 = 0.0,
total_time21 = 0.0;
    double base_diff = 0.0;

    if (clock1 == 0 and clock2 == 1){
       struct timespec clock1_base, clock2_base;
       clock_gettime(clock1, &clock1_base);
       clock_gettime(clock2, &clock2_base);
       base_diff = (clock2_base.tv_sec - clock1_base.tv_sec) * 1e9 +
(clock2_base.tv_nsec - clock1_base.tv_nsec);
    }

    for (int i = 0; i < MAX_ITERATIONS; ++i){
       clock_gettime(clock1, &start_time);
       clock_gettime(clock1, &end_time);
       total_time11 += (end_time.tv_sec - start_time.tv_sec) * 1e9 +
(end_time.tv_nsec - start_time.tv_nsec);
```

```
        clock_gettime(clock2, &start_time);
        clock_gettime(clock2, &end_time);
        total_time22 += (end_time.tv_sec - start_time.tv_sec) * 1e9 +
(end_time.tv_nsec - start_time.tv_nsec);

        clock_gettime(clock1, &start_time);
        clock_gettime(clock2, &end_time);
        total_time12 += (end_time.tv_sec - start_time.tv_sec) * 1e9 +
(end_time.tv_nsec - start_time.tv_nsec) -
                base_diff;

        clock_gettime(clock2, &start_time);
        clock_gettime(clock1, &end_time);
        total_time21 += (end_time.tv_sec - start_time.tv_sec) * 1e9 +
(end_time.tv_nsec - start_time.tv_nsec) +
                base_diff;
    }

    double avg_time11 = fabs(total_time11 / MAX_ITERATIONS);
    double avg_time22 = fabs(total_time22 / MAX_ITERATIONS);
    double avg_time12 = fabs(total_time12 / MAX_ITERATIONS);
    double avg_time21 = fabs(total_time21 / MAX_ITERATIONS);

    std::cout << "Clock " << clock1 << " and " << clock2 << ":\n";

    std::cout << "A1 + A2 = " << avg_time11 << " ns\n";
    std::cout << "B1 + B2 = " << avg_time22 << " ns\n";
    std::cout << "A1 + B2 = " << avg_time21 << " ns\n";
    std::cout << "B1 + A2 = " << avg_time12 << " ns\n";
}

void measure_all_clocks_res_and_deviations(){
    for (int clock: {
            CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_THREAD_CPUTIME_ID,
CLOCK_PROCESS_CPUTIME_ID, CLOCK_REALTIME_COARSE
         }){
        const vector<double> intervals = measure_time(clock);
        measure_deviations(intervals);
        get_clock_resolution(clock);
    }
}

void measure_all_clocks_initialization_and_return(){
    int CLOCKS_EVEN_NUMBER = 6;
    int clocks[CLOCKS_EVEN_NUMBER] = {
        CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_THREAD_CPUTIME_ID,
CLOCK_PROCESS_CPUTIME_ID, CLOCK_REALTIME_COARSE,
        CLOCK_REALTIME
    };

    for (int i = 0; i < CLOCKS_EVEN_NUMBER; i += 2){
        measure_initialization_and_return(clocks[i], clocks[i + 1]);
    }
}
```

```
int main(){
    // measure_all_clocks_res_and_deviations();
    measure_all_clocks_initialization_and_return();
    return 0;
}
```

3. Исходный код программы с измерением параметров RTC:

```
#include <iostream>
#include <fstream>
#include <fcntl.h>
#include <unistd.h>
#include <linux/rtc.h>
#include <sys/ioctl.h>
#include <cmath>
#include <vector>

#define MAX_ITERATIONS (int) 1e5

using std::vector;

void measure_rtc_resolution(){
    int fd = open("/dev/rtc", O_RDONLY);
    if (fd == -1){
       std::cerr << "Failed to open /dev/rtc: " << std::endl;
       return;
    }

    struct rtc_time previous_time, current_time;
    vector<double> intervals;


    if (ioctl(fd, RTC_RD_TIME, &previous_time) == -1){
       std::cerr << "Failed to read initial RTC time: " << std::endl;
       close(fd);
       return;
    }

    for (int i = 0; i < MAX_ITERATIONS; ++i){
       if (ioctl(fd, RTC_RD_TIME, &current_time) == -1){
          std::cerr << "Failed to read RTC time: " << std::endl;
          close(fd);
          return;
       }


       if (current_time.tm_sec != previous_time.tm_sec ||
           current_time.tm_min != previous_time.tm_min ||
           current_time.tm_hour != previous_time.tm_hour){
          double delta = (current_time.tm_hour - previous_time.tm_hour) *
3600 +
                         (current_time.tm_min - previous_time.tm_min) * 60 +
                         (current_time.tm_sec - previous_time.tm_sec);

          intervals.push_back(delta);
          previous_time = current_time;
```

```
        }
    }

    close(fd);


    double sum = 0.0, mean = 0.0, variance = 0.0, mean_absolute_deviation =
0.0;

    for (double interval: intervals){
        sum += interval;
    }
    mean = sum / intervals.size();

    for (double interval: intervals){
        mean_absolute_deviation += std::fabs(interval - mean);
        variance += std::pow(interval - mean, 2);
    }
    mean_absolute_deviation /= intervals.size();
    variance /= intervals.size();
    double standard_deviation = std::sqrt(variance);


    std::cout << "Resolution (average interval): " << mean << " seconds\n";
    std::cout << "Mean Absolute Deviation: " << mean_absolute_deviation << "
seconds\n";
    std::cout << "Standard Deviation: " << standard_deviation << "
seconds\n";
}

int main(){
    measure_rtc_resolution();
    return 0;
}
```

4. Исходный код программы с измерением параметров `__rdtsc()`:

```
#include <iostream>
#include <vector>
#include <cmath>
#include <intrin.h>
#include <thread>
#include <chrono>

#define MAX_ITERATIONS (int) 1e8

using std::vector;

double get_tsc_frequency(){
    unsigned long long start_tsc = __rdtsc();
    auto start = std::chrono::high_resolution_clock::now();
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    unsigned long long end_tsc = __rdtsc();
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    return (end_tsc - start_tsc) / elapsed.count(); // TSC в Гц
```

```
}

void measure_rdtsc_resolution(){
    vector<double> intervals(MAX_ITERATIONS);
    int count = 0;
    unsigned long long previous_tsc = __rdtsc();

    for (int i = 0; i < MAX_ITERATIONS; ++i){
       unsigned long long current_tsc = __rdtsc();

       if (current_tsc != previous_tsc){
          double delta = static_cast<double>(current_tsc - previous_tsc);
          intervals[count++] = delta;
          previous_tsc = current_tsc;
       }
    }

    intervals.resize(count);

    double sum = 0.0, mean = 0.0, variance = 0.0, mean_absolute_deviation =
0.0;

    for (double interval: intervals){
       sum += interval;
    }
    mean = sum / intervals.size();

    for (double interval: intervals){
       mean_absolute_deviation += std::fabs(interval - mean);
       variance += std::pow(interval - mean, 2);
    }
    mean_absolute_deviation /= intervals.size();
    variance /= intervals.size();
    double standard_deviation = std::sqrt(variance);
    double tsc_frequency = get_tsc_frequency();

    std::cout << "Resolution (average interval): " << mean / tsc_frequency *
1e9 << " ns\n";
    std::cout << "Mean Absolute Deviation: " << mean_absolute_deviation /
tsc_frequency * 1e9 << " ns\n";
    std::cout << "Standard Deviation: " << standard_deviation /
tsc_frequency * 1e9 << " ns\n";
    std::cout << "tsc_frequency: " << tsc_frequency / 1e9 << " GHz\n";
}

int main(){
    measure_rdtsc_resolution();
    return 0;
}
```

5. Исходный код программы с измерением параметров QPC:

```
#include <iostream>
#include <vector>
#include <cmath>
#include <windows.h>
```

```cpp
#define MAX_ITERATIONS (int) 1e8

void measure_qpc_resolution(){
    LARGE_INTEGER frequency;
    QueryPerformanceFrequency(&frequency);

    LARGE_INTEGER previous_time, current_time;
    std::vector<double> intervals(MAX_ITERATIONS);
    int count = 0;

    QueryPerformanceCounter(&previous_time);

    for (int i = 0; i < MAX_ITERATIONS; ++i){
       QueryPerformanceCounter(&current_time);

       if (current_time.QuadPart != previous_time.QuadPart){
          double delta = static_cast<double>(current_time.QuadPart -
previous_time.QuadPart);
          intervals[count++] = delta;
          previous_time = current_time;
       }
    }

    intervals.resize(count);

    double sum = 0.0, mean = 0.0, variance = 0.0, mean_absolute_deviation =
0.0;

    for (double interval: intervals){
       sum += interval;
    }
    mean = sum / intervals.size();

    for (double interval: intervals){
       mean_absolute_deviation += std::fabs(interval - mean);
       variance += std::pow(interval - mean, 2);
    }
    mean_absolute_deviation /= intervals.size();
    variance /= intervals.size();
    double standard_deviation = std::sqrt(variance);

    std::cout << "Resolution (average interval): " << mean /
frequency.QuadPart * 1e9 << " ns\n";
    std::cout << "Mean Absolute Deviation: " << mean_absolute_deviation /
frequency.QuadPart * 1e9 << " ns\n";
    std::cout << "Standard Deviation: " << standard_deviation /
frequency.QuadPart * 1e9 << " ns\n";
}

int main(){
    measure_qpc_resolution();
    return 0;
}
```

6. Вывод измерения времени инициализации и возврата ответа:

```
Clock 0 and 1:
A1 + A2 = 32.7302 ns
B1 + B2 = 29.3478 ns
A1 + B2 = 332.348 ns
B1 + A2 = 255.815 ns
Clock 3 and 2:
A1 + A2 = 106.25 ns
B1 + B2 = 143.75 ns
A1 + B2 = 134.375 ns
B1 + A2 = 125 ns
Clock 4 and 0:
A1 + A2 = 0.20001 ns
B1 + B2 = 33.0242 ns
A1 + B2 = 500292 ns
B1 + A2 = 500333 ns
```

7. Пример использования `__rdtsc()`:

```cpp
#include <iostream>
#include <intrin.h> // Для __rdtsc()
int main() {
    unsigned long long timestamp = __rdtsc();
    std::cout << "Timestamp: " << timestamp << "\n";
    return 0;
}
```

8. Пример вычисления коэффициента TSC и замеров времени выполнения с помощью RDTSC:

```cpp
#include <iostream>
#include <intrin.h>
#include <thread>
#include <chrono>
double get_tsc_frequency(){
    unsigned long long start_tsc = __rdtsc();
    auto start = std::chrono::high_resolution_clock::now();
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    unsigned long long end_tsc = __rdtsc();
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    return (end_tsc - start_tsc) / elapsed.count(); // TSC в Гц
}
void measure_time(double tsc_freq){
    unsigned long long start_tsc = __rdtsc();
    auto start = std::chrono::high_resolution_clock::now();
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    unsigned long long end_tsc = __rdtsc();
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    unsigned long long elapsed_tsc = end_tsc - start_tsc;
    double elapsed_seconds = elapsed_tsc / tsc_freq;
    std::cout << "Measured Sleep Time: " << elapsed.count() << " seconds
(chrono)" << std::endl;
    std::cout << "Measured Sleep Time: " << elapsed_seconds << " seconds
(TSC)" << std::endl;
```

```
}
int main(){
    double tsc_freq = get_tsc_frequency();
    std::cout << "TSC Frequency: " << tsc_freq / 1e9 << " GHz" << std::endl;
    measure_time(tsc_freq);
    return 0;
}
```

9. Пример использования QPC:

```
#include <chrono>
#include <iostream>
#include <thread>
#include <windows.h>
int main(){
    LARGE_INTEGER StartingTime, EndingTime, ElapsedMicroseconds;
    LARGE_INTEGER Frequency;
    QueryPerformanceFrequency(&Frequency);
    QueryPerformanceCounter(&StartingTime);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    QueryPerformanceCounter(&EndingTime);
    ElapsedMicroseconds.QuadPart = EndingTime.QuadPart -
StartingTime.QuadPart;
    double elapsed_time = ElapsedMicroseconds.QuadPart;
    elapsed_time /= Frequency.QuadPart;
    std::cout << elapsed_time << std::endl;
    return 0;
}
```