Bernburg
Dessau
Köthen

Hochschule Anhalt
Anhalt University of Applied Sciences

emw
Fachbereich
Elektrotechnik, Maschinenbau
und Wirtschaftsingenieurwesen

Master Thesis

submitted in partial satisfaction of the requirements
for the degree of Master of Engineering (M. Eng.)

Irina Fedotova

Name Surname

Faculty of Electrical, Mechanical and
Industrial Engineering, 2012, 4055928

Program, Matriculation, Matriculation number

Implementation of a unified user-space time handling library under the Linux OS

Prof. Dr.-Ing. E. Siemens

1. Supervisor

Prof. Dr.-Ing. M. Enzmann

2. Supervisor

13.09.2012

Submission date

Table of Contents

List of Abbreviations

TSC	Time Stamp Counter
HPET	High Precision Event Timer
RTC	Real-time clock
PIT	Programmable Interval Timer
APIC	Advanced Programmable Interrupt Controller
PIC	Programmable Interrupt Controller
GP Timer	General Purpose Timer
RDTSC	Read Time Stamp Counter instruction
RDTSCP	Read Time Stamp Counter and Processor ID instruction
syscall	System Call
vsyscall	Virtual System Call
OOP	Object Oriented Programming

1 Intoduction

Accurate measurement of time domain in modern computer systems often plays a key role in high-performance computing applications and communication processes. Without that it is difficult to imagine the process of end-to-end performance monitoring and designing traffic control algorithms in computer networks [1, 2, 3]. The correct performance of the majority of services and proper algorithm implementation in telecommunication protocols depend on the accuracy of time domain measurements. Due to that fact the necessity for reliable and fast time sourcing is obvious.

However the high-precise and high-performance time interval measurement is facing some significant issues. Firstly, the Linux kernel uses different time sources. The most interesting are the Time Stamp Counter (TSC) [4, 5, 6] and the High Precision Event Timer (HPET) [7, 8]. Their main characteristics such as reliability and stability depend heavily on the processor architecture. So the time counter can have non-monotonic characteristics, it can even decrement ticks, or it may overflow and wrap back to zero. At this point the high priority task is to identify the most suitable and reliable time source for the appropriate measurements. Moreover, there are some cases [9, 10, 11] where it is not enough to have the accuracy of a timer up to microseconds. If it is meant the real high-accuracy performance, there must be a capability to handle nanoseconds and an ability to make appropriate operations with them.

In this thesis, a novel approach solving the problems of high-efficient and precise time measurements on PC-platforms as well as ARM-architectures is elaborated. As a basis some already existing classes *CTSCTimer* and *HPETTimer* have been used. They provided the access to TSC and HPET hardware timers, respectively. However, there were no any guarantee that used processor architecture possesses of these timers, and even if it does no guarantee, how stable TSC and HPET are. In other words, it means, that user should investigate in advance which timer hardware is more suitable for his purposes and after that take a decision, which classes to use. Furthermore, existing *CTSCTimer* and *HPETTimer* allow operations at most with the resolution of microseconds. Also their sleep state spinning entirely on the CPU – so called *busy-wait*, which is very inefficient. It should be taken into account that *CTSCTimer* was written about 10 years ago. Since that the world of C++ has undergone enormous change, and one of the main goals were developing better software design, making them correspond with the C++11 standards and achieving greater efficiency and code maintainability.

As a result, a new high performance timer class *HighPerTimer* and a corresponding library has been introduced. It combines the known time counters and automatically takes upon itself the choice of suitable source (TSC, HPET or OS timer) like Linux kernel does it at boot time. It should be emphasized that the setting source for *HighPerTimer* occurs in user space, not in kernel space. The latter feature makes the *HighPerTimer* library very usable in versatile program environments on different PC and embedded processors. *HighPerTimer* class also has the user ability to change default timer source to another one. It provides access to some advanced hardware features such as CPU information or some TSC and HPET features. *HighPerTimer* extends a scope and successfully runs on new and old models of Intel, AMD, VIA and ARM processors.

One of the most important achievements in this work is a new *HighPerTimer* combined sleep function. Actually, for sleeps standard C Library system sleep or busy-waiting mode (or so called spinning wait) is mostly used. System sleep runs not on CPU, but is a missing the target wake-up time for more than 50 microseconds in average, which is unacceptable for high-accuracy program sleep. Sleep in busy-waiting have a miss about 100 nanoseconds, but keeps the CPU busy and prevents other threads from using the given CPU. It means that its performance is limited by numbers of processor cores. *HighPerTimer* sleep combines these two ways of sleeping and has a minimum nanoseconds miss with minimum CPU utilization, which creates a big competitive advantage over its predecessor solutions.

2 Main System Timers

The kernel usually explicitly interacts with three kinds of hardware clocks: the Real Time Clock (RTC) [12], the Time Stamp Counter (TSC) and the High Precision Event Timer (HPET). The first two hardware devices allow the kernel to keep track of the current time of day, the latter device is programmed by the kernel so that it issues interrupts at a fixed, predefined frequency. However, there are always one more alternative – to rely on the operating system choice, which means to use some system functions for obtaining the time value. This system functions apply for the most stable clock source, which it is set during the boot by kernel. Accordingly, the *HighPerTimer* library posses the most usable and preferable time counters: TSC, HPET and alternative timer of operating system (OS Timer). For each of them, brief overview of their functionality, preferred use as well as disadvantages along with some performance aspects are described below.

2.1 Time-Stamp Counter Overview

The Time Stamp Counter is a hardware feature found on a number of contemporary processors (on the P6 family, Pentium, Pentium M, Pentium 4, Intel Xeon, Intel Core Solo and Intel Core Duo processors and later processors) [13 vol. 3B 17-49]. The TSC is a special register which is simply incremented every clock cycle of a particular CPU. Since the clock is the fundamental unit of time as seen by the processor, the TSC provides the highest-resolution timing information available for that processor.

2.1.1 Register Definition and General Capabilities

The Time Stamp Counter is a 64-bit per-CPU register available on processors (beginning with the Intel Pentium) that can be used to monitor and identify the relative time occurrence of processor events. It is set to 0 following a reset of the processor and since that the counter increments. TSC posses the fastest and the lowest possible overhead way of getting CPU timing information with comparison of obtaining time value from other timer hardware. The RDTSC and RDTSCP instructions read the time-stamp counter and return a monotonically increasing unique value whenever executed, except for a 64-bit counter wrap around. The RDTSC and RDTSCP instructions are normally available in user mode. Moreover, Intel guarantees that the Time-Stamp Counter will not wrap around within 10 years after being reset [13 vol. 3B 17-50].

The RDTSC instruction (unlike the RDTSCP instruction) is not serializing or ordered with other instructions and can be executed out-of-order with respect to other

instructions around. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDTSC instruction operation is performed [13 vol. 2B 4-461, 14 p. 411, 4 p. 3].

The RDTSCP instruction waits until all previous instructions have been executed before reading the counter, which called serializing feature. The advantage of using RDTSCP is its feature to vanish cashes from all waiting commands in queue before execution. It will guarantee that RDTSCP instruction will be executed in the order as user expects. According to the Table 2.1, average cost of setting TSC timer with RDTSC instruction is 10 nanoseconds and with RDTSC instruction it is about 17 nanoseconds. Due to this results, it can be concluded that reading RDTSCP instruction is slower for about 6.7 nanoseconds, but using this instruction is much more safety and reliable. The results were obtained on Intel ® Core ™ i7-2600 CPU processor, it was run 100 millions tests and results were not filtered.

The way of reading TSC timer	Mean, nsec	Standard Deviation, nsec	The final mean difference, nsec
With RDTSC instruction	10.2031	20.9812	6.7067
With RDTSCP instruction	16.9097	21.8443	

Table 2.1: Average cost of setting TSC timer with RDTSC and RDTSCP instructions

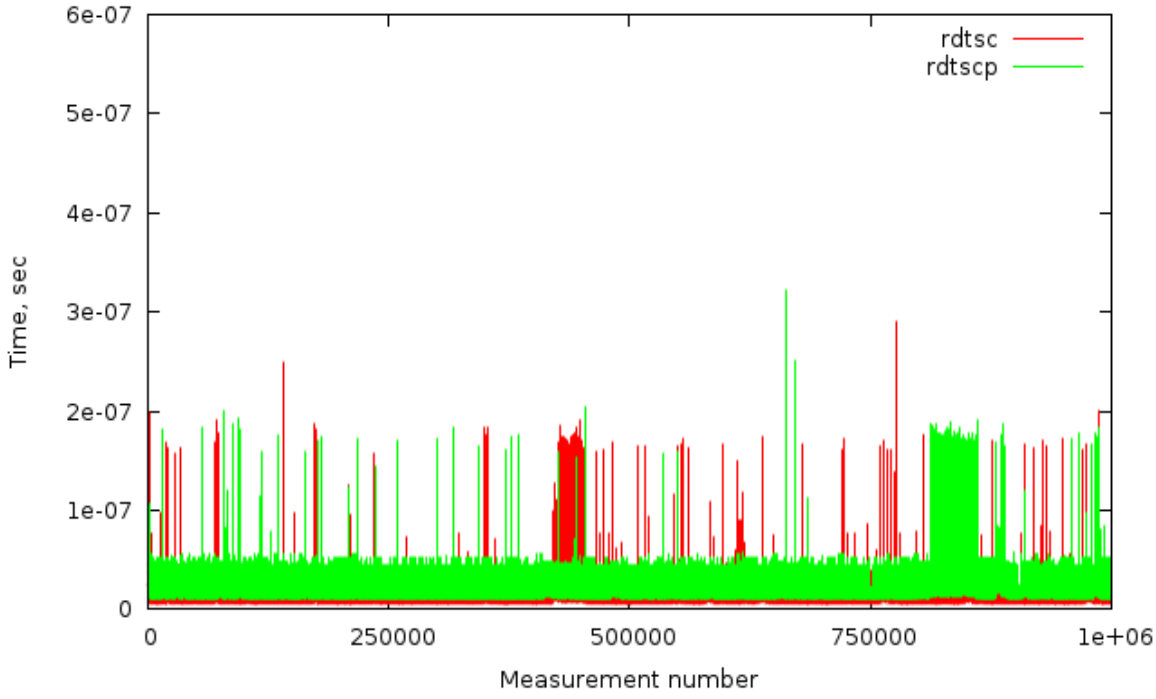

Figure 2.1: Comparison behavior of TSC with RDTSC and RDTSCP instructions

2.1.2 Identification of TSC Implementation Issues

The TSC is the finest grained, widest, and most convenient timer device to access. However, it also has several drawbacks. Since the advent of multi-core CPUs, systems with multiple CPUs, and "hibernating" operating systems, the TSC cannot be relied on providing accurate results unless great care is taken to correct the possible flaws: rate of tick and whether all cores have identical values in their time-keeping registers. With the introduction of these features, it can no longer be ensured that the Time Stamp Counters of multiple CPUs on a single motherboard are synchronized. Thus programmers can only obtain reliable results by locking their code to a single CPU. Even then, the CPU speed may change due to power-saving measures taken by the OS or BIOS, or the system may be hibernated and later resumed (resetting the Time Stamp Counter). Several forms of power management technology vary the processor's clock speed dynamically and thereby change the TSC rate with little or no notice [15, 16 p. 4]. In those latter cases, the counter must be periodically recalibrated to stay accurate.

Moreover, not all processors families increment the Time-Stamp Counter in similar way including Pentium 4 processors, old families of Intel Xeon processors, Intel Core Solo, old models of Intel Core Duo processors and so on [13 vol. 3B 17-49]. The specific processor configuration determines the behavior. Constant TSC behavior ensures that the duration of each clock tick is uniform and supports the use of the TSC as a wall clock timer even if the processor core changes frequency.

In newer processors the TSC may support an enhancement, referred to as Invariant TSC feature, which is not so tightly bound to processor cores and their cycles and, according to Intel documentation, has a constant rate. To determine whether a processor possesses Invariant TSC it is sufficient to inquire a processor ID handle and in case flag "*InvariantTSC*" equals 1, one can use TSC, if not, it would be better to look for other solutions [13 vol. 3B 17-50]. Accordingly, the TSC is considered stable if there are Invariant TSC flag or appropriate processor refers to some specific Intel-family processors.

2.1.3 TSC Performance Optimization Aspects

In the context of this thesis, initialization of the time counter means to ensure its reliability to provide accurate time information and make it is ready for use. The algorithm of the initialization process for Time-Stamp Counter consists of several steps and here is no way to avoid of using the CPUID opcode.

The CPUID opcode is a processor supplementary instruction (its name derived from CPU IDentification) for the x86 architecture. It was introduced by Intel in 1993 when it introduced the Pentium and SL-Enhanced 486 processors [17 p. 5]. By using the CPUID opcode, software can determine processor type and the presence of some specific features. In assembly language the CPUID instruction takes no parameters as CPUID implicitly uses the EAX register for parameterization. The EAX register should be loaded with a value specifying what kind of information to return. Below it is shown the example of implementation of the function for executing the CPUID instruction in a scope of *HighPerTimer*:

```
// InputEAX — value of EAX register and input data
// RegsCPUIDoutput — structure for saving the value of registers
TSCTimer::RegsCPUID TSCTimer::ExecuteCPUID ( uint32_t InputEAX )
{
    RegsCPUID RegsCPUIDoutput;
    asm volatile
    (
       "cpuid;"
       : "=a" ( RegsCPUIDoutput.EAXBuf ),
         "=b" ( RegsCPUIDoutput.EBXBuf ),
         "=c" ( RegsCPUIDoutput.ECXBuf ),
         "=d" ( RegsCPUIDoutput.EDXBuf )
       : "a" ( InputEAX )
    );
    return RegsCPUIDoutput;
}
```

Listing 2.1: Function executing CPUID instruction with driving EAX register value

CPUID should be called with EAX = 0 first, as this will return the highest calling parameter that the CPU supports. The highest basic calling parameter (largest value that EAX can be set to before calling CPUID) is returned in EAX [17 p. 17].

The second step is retrieving Vendor ID and Brand string. Calling CPUID with EAX = 0 returns the CPU's manufacturer ID string - a twelve character ASCII string stored in EBX, EDX, ECX - in that order [17 p. 20, 18 p. 10]. The following are some known processor manufacturer ID strings (in particular used in a scope of *HighPerTimer*):

- "GenuineIntel" — Intel
- "CentaurHauls" — Centaur
- "AuthenticAMD" — AMD

Calling CPUID with EAX = 80000002h, 80000003h, 80000004h returns processor Brand String [17 p. 45, 18 p. 11]. CPUID must be issued with each parameter in sequence to get the entire 48-byte null-terminated ASCII processor brand string. It is necessary to check whether the feature is supported by the CPU by issuing CPUID with EAX = 80000000h first and checking if the returned value is greater or equal to 80000004h.

The presence of the RDTSCP instruction is indicated by CPUID leaf 80000001h, *EDX* bit 27. If the bit is set to 1 then RDTSCP is present on the processor [17 p. 45]. Processor’s support for invariant TSC is indicated by CPUID leaf 80000007h, EDX bit 8. If this feature is presented, TSC doesn’t tightly bound to processor cores and has a stable rate. So TSC initialization function is succeed and returns true, thus *HighPerTimer* time source is assigned to the TSC timer.

If processor doesn't support Invariant TSC, it is recommended to check for some specific processors. To retrieve processor model and family, it is called CPUID with EAX=1 [17 p. 21, 18 p. 10]. This returns the CPU's stepping, model, and family information in EAX (also called the CPU signature), feature flags in EDX and ECX, and additional feature information in EBX. The format of the information in EAX is as follows:

- `3:0 - Stepping`
- `7:4 - Model`
- `11:8 - Family`
- `13:12 - Processor Type`
- `19:16 - Extended Model`
- `27:20 - Extended Family`

Intel suggests applications to display the family of a CPU as the sum of the "Family" and the "Extended Family" fields shown above, and the model as the sum of the "Model"

and the 4-bit left-shifted "Extended Model" fields. AMD recommends the same only if "Family" is equal to 15 (i.e. all bits set to 1). If "Family" is lower than 15, only the "Family" and "Model" fields should be used while the "Extended Family" and "Extended Model" bits are reserved. If "Family" is set to 15, then "Extended Family" and the 4-bit left-shifted "Extended Model" should be added to the respective base values. For example, here is the check for TSC stability of some specific Intel processor:

```
if ( !memcmp ( VendorString, IntelVendor, 12 ) )
{
    // since family 0x0f and model 0x03 all has constant tsc
    // including Pentium 4, Intel Xeon processors
    if ((( SignCPUIDoutput.FamilyID + SignCPUIDoutput.ExtendedFamilyID)
== 0x0f) && ((SignCPUIDoutput.ExtendedModel << 4|SignCPUIDoutput.Model
) >= 0x03))
    {
       TSCTimer::HasConstantTSC = true;
       return true;
    }
}
```

Listing 2.2: The check of specific Intel processor for TSC stability

The final task of figuring out the actual frequency of TSC is accomplished next. In the context of *HighPerTimer* it is more useful to save the value of frequency as the number of cycles per one microsecond (*int64_t TicsPerUsec*) and reciprocal value as the number of nanoseconds within one cycle (*double NsecPerTic*). To figure out TSC frequency data returned from *gettimeofday()* system call and data retrieved from direct reading RDTSC/RDTSCP instruction are compared. This operation is performed in a loop of 5 steps with system sleep for 20 000 microseconds between two inquiries. At each step, it is compared data from both methods, saved intermediate result and finally calculated the mean value of frequency.

2.2 High Precision Event Timer Overview

The High Precision Event Timer is a hardware timer used in personal computers, formerly referred to by Intel as a *Multimedia Timer* [8 p. 4]. The term HPET was selected to avoid confusion with the multimedia timers software feature introduced in the MultiMedia Extensions to Windows 3.0.

2.2.1 Hardware Implementation and General Capabilities

The High Precision Event Timer was developed jointly by Intel and Microsoft and has been incorporated in PC chipsets since 2005 [19]. The main motivation for its creation was the necessity to replace slow and older Programmable Interval Timer (PIT) [20] whose frequency (1.19 MHz) did not meet the current requirements. The HPET circuit in modern PCs is integrated into the south bridge chip and consists of a 64-bit main counter counting at a frequency of at least 10 MHz and a set of timers that can be used by the operating system. These timers are 32- or 64-bit wide. Each timer can be configured to generate a separate interrupt. HPET specification allows for a block of 32 timers, with support for up to 8 blocks, for a total of 256 timers [8 p. 7]. However, specific implementations can include only a subset of these timers.

The timers are implemented as a single up-counter with a set of comparators. The counter increases monotonically. When software does two consecutive reads of the counter, the second read will never return a value that is less than the first read unless the counter has actually rolled over. Each timer includes a match register and a comparator. Each individual timer can generate an interrupt when the value in its match register equals the value of the free-running counter. Some of the timers can be enabled to generate a periodic interrupt.

The registers associated with these timers are mapped to memory space (almost like the I/O APIC). However, it is not implemented as a standard PCI function. Instead, the BIOS reports to the operating system the location of the memory-mapped register space consumed by the timers. The hardware can support re-locatable address decode space, however the BIOS will set this space prior to handing it over to the OS. It is not expected that the OS will move the location of these timers once it is set by the BIOS [8 p. 7].

2.2.2 Identification of HPET Implementation Issues

The main problem of using HPET from user space is that operating systems designed before HPET has been introduced, can’t access them, so they work only on hardware that has other timer facilities. Indeed most current south bridge chips have legacy-supporting instances of PIT [20], PIC [21, 22], APIC [23] and RTC [12] devices incorporated into their silicon whether or not they are used by the motherboard or the operating system, which is why even a very modern PC can still run older operating

systems. The following operating systems are known not to be able to use HPET: Windows XP, Windows Server 2003, and earlier Windows versions and Linux kernels prior to 2.6 [19]. The basic proof HPET availability is the existence of */dev/hpet* file.

The difficulties are exacerbated if the main counter drives in a 32-bit mode. In fact a 32-bit timer can be read directly using processors that are capable of 32-bit or 64-bit instructions (in other words, read for one step). However, if the HPET main counter register has frequency of 10 MHz in a 32-bit mode, an overflow arises every 7,16 minutes [24 p. 10]. It is very dangerous to use it and this case should be avoided. In average cost of obtaining time value from HPET devise can more than 1 microsecond [24, 25, 26].

Possible fails during the HPET initialization process can be occurred when the software attempts to open the device, to map it into memory and in case if a size of the main counter register has 32-bit width. In general, all possible errors are combined in enumeration class *HPETFail*, which is accessible for user through *AccessTimeHardware* class.

```
enum class HPETFail
{
   ACCESS,    )
   FAULT,     |
   NOENT,     |
   MFILE,     }  // predominantly for attempt to open file of the device
   AGAIN,     |
   BUSY,      )
   BADF,      )
   NODEV,     }  // for attempt to mmap device, including BUSY error
   AGAIN,     |
   NOMEM,     )
   MC32BIT,      // if main counter has 32-bit width
   UNKNOWN       // unknown error
};
```

Listing 2.3: Possible reasons for HPET fail

In general, these names of HPET fail reason are corresponding with errors from *errno.h*, header file indicating the kind of error and detailed description of each errors are available from Linux man, with the exception of *MC32BIT* and *UNKNOWN* [27, 28]. Moreover, in case of *open()* and *mmap()* this enumeration contains not the whole list of possible errors, only the most probable, which can occurred. The others errors, not mentioned in that list, are marked as *UNKNOWN*.

2.2.3 HPET Performance Optimization Aspects

As mentioned above, a case when HPET main counter register has 32-bit width, should be avoided. The size of the main counter register is indicated by the 13th bit [8 p. 11]. If this bit is 0, the main counter is 32 bits wide:

```
HPETTimer::HpetFd = ::open ( "/dev/hpet", O_RDONLY );
HPETTimer::HpetAdd_ptr = ( unsigned char * ) mmap ( NULL, 1024,
PROT_READ, MAP_SHARED, HPETTimer::HpetFd, 0 );

// 32-bit HPET main counter overruns every 7,16 minutes, so it is
denied using this source.
uint64_t HpetMcounterSize = * ( ( int64_t * ) ( HPETTimer::
HpetAdd_ptr ) ) >> 13;
if ( ! HpetMcounterSize )
{
    // saves the reason which hpet initialization failed
    HPETTimer::HPETFailReason = HPETFail::MC32BIT;
    return false;
}
```

Listing 2.4: Indication of the 32-bit HPET main counter register

In case of 64-bit mode of a timer, an overflow occurs only after 58,494 years, but a 32-bit processor may not be able to read a 64-bit timer directly. The Intel specification suggests several ways of reading the 64 bit counter using 32 bit reads [8 p. 26]. The most suitable in the scope of *HighPerTimer* is doing a multiple separate reads of the counter. It is read the high 32 bits, then the low 32 bits, then the high 32 bits again. If the high 32 bits have not changed between the two reads, then a rollover has not happened and the low 32 bits are valid. Here is a detailed description of getting HPET ticks:

```
// get hpet timer tics
// the case when the main counter has 64 bit width, but a 32 bit
// addresses, meaning reading the main counter must be performed
// within two memory accesses. Multiple reads of HPET main counter
// register avoiding an accuracy problem, which may be arised if after
// reading one half, the other half rolls over and changes the first
// half

 uint64_t Hpet_High_Order = * ( reinterpret_cast<int32_t*>
```

```
    ( HPETTimer::HpetAdd_ptr + MainCounterOffsetHigh ) );
 uint64_t Hpet_Low_Order  = * ( reinterpret_cast<int32_t*>
    ( HPETTimer::HpetAdd_ptr + MainCounterOffset ) );
 uint64_t _Hpet_High_Order = * ( reinterpret_cast<int32_t*>
    ( HPETTimer::HpetAdd_ptr + MainCounterOffsetHigh ) );

// if both read from the high 32 bits of the main counter register
// HPET are equal, clock cycles are current returned
if ( Hpet_High_Order == _Hpet_High_Order )
{
    return ( Hpet_High_Order << 32 | Hpet_Low_Order );
}
```

Listing 2.5: Extraction HPET time value from a 64-bit timer by a 32-bit processor

The *MainCounterOffset* variable is equal to 0x0f0 and *MainCounterOffsetHigh* to 0x0f4 and the mean offset value from the first mapped memory address for the main counter register [8 p. 15]. After successful timer initialization process, HPET frequency is calculated. It is read 4 bites from the HPET table adding period offset, which is equal to 0x004 in hexadecimal system.

2.3 Operating System Timer Source

Timers of operating system keeps the current time and date on the running system, so that they can be returned to user programs through the respective system calls and used by the kernel itself as timestamps for files and network packets. This section describes the system calls related to timing measurements and the corresponding service routines.

2.3.1 Basic Principles of Time-Related System Calls

A personal computer has a battery driven hardware clock. The battery ensures that the clock will work even if the rest of the computer is without electricity. The hardware clock can be set from the BIOS setup screen or from whatever operating system is running on the system. The Linux kernel keeps track of time independently of the hardware clock. During the OS boot, Linux sets its own clock to the same time as the hardware clock. After this, both clocks run independently. Linux maintains its own clock because looking at the hardware is slow and complicated.

However, to obtain the time value on the running system, it is necessary to interact with a hardware device. It is known that all the devices communicate through drivers which run in kernel space and have full access to the hardware. It is not possible to directly link user-space applications with kernel space code. For reasons of security and reliability, user-space applications must not be allowed to access memory regions, used by kernel. Instead, the kernel must provide a mechanism by which a user-space application can "signal" the kernel that it wishes to invoke a system call.

System calls (often named *syscalls*) are function invocations made from user space into the kernel (the core internals of the system) in order to request some service or resource from the operating system. Linux implements far fewer system calls than most other operating system kernels. For example, a count of the i386 architecture's system calls comes in at around 300, compared with the allegedly thousands of system calls on Microsoft Windows. In addition, system calls available on one hardware-architecture may differ from those available on another. Nonetheless, a very large subset of system calls—more than 90 percent—is implemented by all architectures [29 p. 18].

The application tells the kernel which system call to execute and with what parameters via machine registers. System calls are denoted by number, starting at 0. On the i386 architecture, for example, to request system call 5 (which happens to be *open()*), the user-space application stuffs 5 in register EAX before issuing the instruction. Parameter passing is handled in a similar manner. On i386 a register is used for each possible parameter—registers EBX, ECX, EDX, ESI, and EDI contain, in order, the first five parameters [29 p. 19]. In the rare event of a system call with more than five parameters, a single register is used to point to a buffer in user space where all of the parameters are kept. However, most system calls have only a couple of parameters. Other architectures handle system call invocation differently, although the general approach is the same.

A system call is processed in kernel mode, which is accomplished by changing the processor execution mode to a more privileged one, and a privilege context switch does occur [30, 31, 32]. Switching from one process to another requires up to 2 microseconds for doing the administration - saving and loading registers and memory maps, updating various tables and list etc.

2.3.2 Virtual System Call *gettimeofday()* and *clock_gettime()*

The virtual system call (*vsyscall*) is a mechanism on Linux used to reduce the overhead of system calls. Its basic function is providing fast access to functionality which does not need to run in kernel mode. Recently vsyscall has come to be seen as an enabler of security attacks, so some patches have been put together to phase it out [33].

The vsyscall area has been added as a way to execute specific system calls which do not need any real kernel-level of privilege to run. A good-example for a vsyscall is *gettimeofday()* and *clock_gettime()* - all they need to do is to read the kernel's data of the current time. The kernel allows at memory page containing the current time to be mapped read-only into user space; the page contains a fast *gettimeofday()* and *clock_gettime()* execution code. Using this virtual system call, the C library can provide a fast *gettimeofday()* which never actually has to switch into the kernel context. Accordingly, the result is a kernel system call emulating a virtual system call which was put there to avoid the kernel system call in the first place and prevent a context switch, respectively. This capability allows saving a fraction of a microsecond (more precise it is up to 2 microseconds) in comparison with a pure system call. The comparison of real system calls with virtual system calls is given in details in Bachelor Thesis for Anhalt University of Applied Sciences by H. Hu [24]. Vsyscall has some limitations, among others, there is only space for a handful of virtual system calls in an OS.

The function *gettimeofday*() can get and set the time as well as a *timezone*. Filling a structure timeval (as specified in <*sys/time.h*>), it gives the number of seconds and microseconds since the UNIX Epoch. The functions *clock_gettime()* has higher accuracy and retrieve the number of seconds and nanoseconds in a *timespec* structure. It sets the time of the specified clock *clk_id.*

The *clk_id* argument is the identifier of the particular clock on which to act. A clock may be system-wide and hence visible for all processes, or per-process if it measures time only within a single process. All implementations support the system-wide real-time clock, which is identified by *CLOCK_REALLTIME*. Initially for OS Timer source *CLOCK_REALTIME* was chosen, which represents the machine's best-guess as to the current wall-clock. *CLOCK_MONOTONIC* is a clock that cannot be set and represents monotonic time since some unspecified starting point [34]. *CLOCK_MONOTONIC* represents monotonic time, which never goes back and always growing, when in theory *CLOCK_ REALTIME* can jump forwards and backwards as the system time-of-day clock

is changed. Actually this behavior has never been noticed on the go of *HighPerTimer*, because most modern processors posse at least one constant timer source for system call of clock_gettime(). Comparing their mean values on the Table 2.2, it is concluded that there are no big difference, only about 3 nanoseconds, but however *CLOCK_ REALTIME* argument is more precise.

The clk_id argument	Mean, nsec	Standard Deviation, nsec	The final mean difference, nsec
CLOCK_ REALTIME	31.232	45.056	2.887
CLOCK_MONOTONIC	34.119	53.927	

Table 2.2: Mean and standard deviation values of *clock_gettime()* with *CLOCK_ REALTIME* and *CLOCK_MONOTONIC* arguments on the Intel Core i7 processor

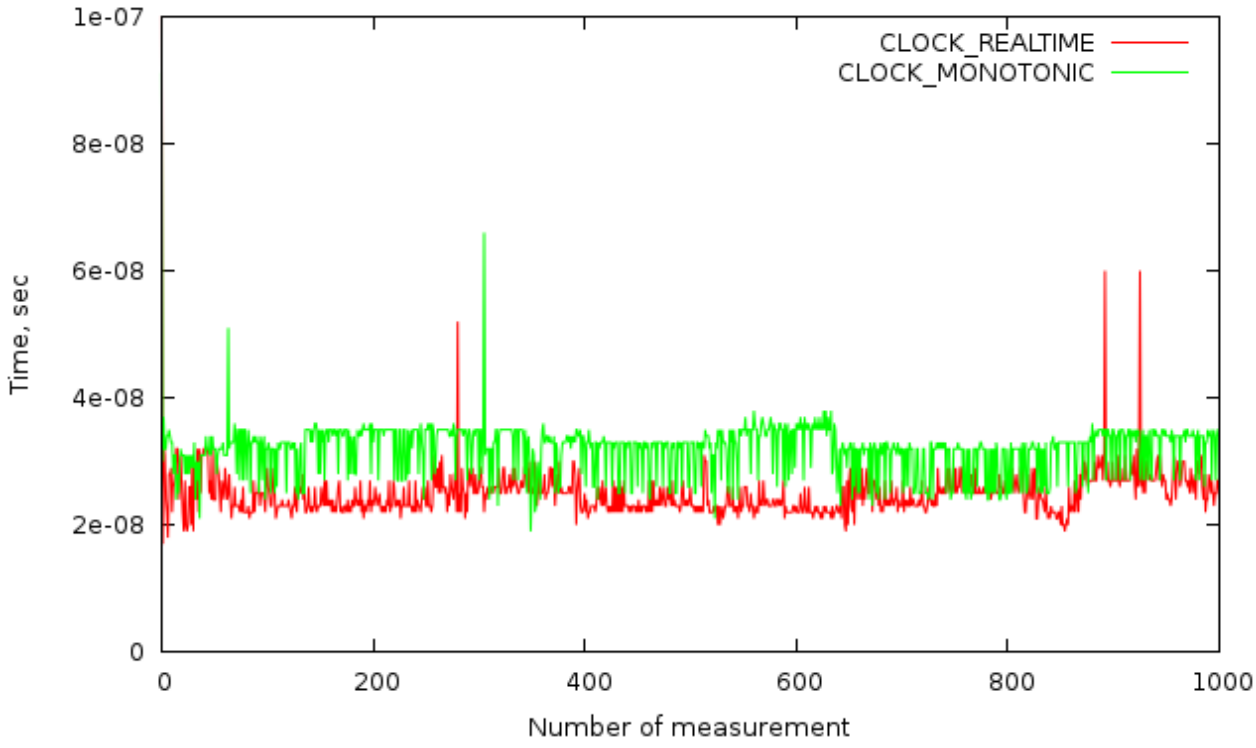

Figure 2.2: Measurements of *clock_gettime()* with *CLOCK_ REALTIME* and *CLOCK_MONOTONIC* arguments on the Intel Core i7 processor

For further work it was determined to switch clock id argument to more reliable *CLOCK_MONOTONIC*. By spending these couple of nanoseconds of CPU overhead, it is provided kind of assurance that no backward jumps in time will occur even in future implementations.

2.3.3 Performance Aspects of Choosing Clock Hardware Source

OS Timer is a good alternative source when it is impossible to rely on HPET or TSC timers, but it should be used with caution. Firstly, there is only one hardware timer which can be used as a clock source which is not known in advance. Secondly, there is no any guaranty whether a real or virtual system call is issued by *clock_gettime()*. Below it is shown two examples with current HPET timer on AMD Athlon processor and TSC timer on Intel Core i7 processor and it is demonstrated the essential difference between current clock source on each processor and system call of OS Timer.

Processor AMD Athlon has explicitly unstable TSC and uses HPET as a clock source. Comparing HPET behavior with *clock_gettime()* call (Figure 2.2), it is suggested that *clock_gettime()* invokes HPET device. According to Table 2.2, the difference in their mean values is about 17 nanoseconds, the values of standard deviation are also very close to each other. It provides a basis to suggest that in this particular case OS Timer invokes HPET hardware. However, relatively to the mean values of 1 microsecond, 17 nanoseconds for system call are not very essential. More detailed figure on this processor including TSC measurements is shown on the Figure 3.2.

Main features of tested processor:

Processor (CPU): AMD Athlon ™ X2 Dual Core Processor BE-2350
Speed: 1000 MHz
Cores: 2
RDTSCP instruction available
HPET Frequency: 25 MHz
cache size: 512 Kb

Timer source	Mean, usec	Standard Deviation, usec	The final mean difference, nsec
HPET	1.1125	0.3073	16.9241
OS	1.1294	0.3582	

Table 2.3: Mean and standard deviation values of HPET and *clock_gettime()* call on the AMD Athlon processor

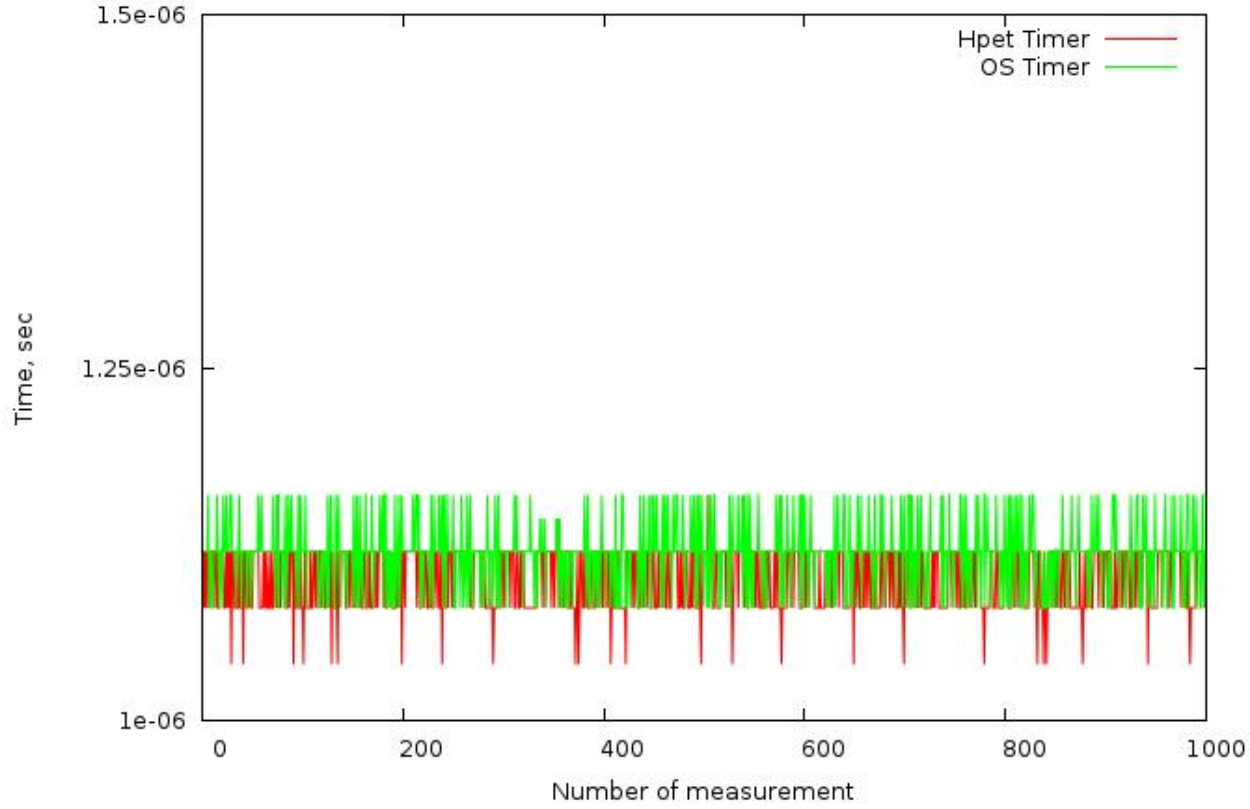

Figure 2.3: Measurements of *clock_gettime()* call with comparison of HPET on the AMD Athlon processor

Processor Intel Core i7 has Invariant TSC with constant rate and so uses it as a clock source. On the Figure 2.3 it is shown measurements of TSC timer with comparison *of clock_gettime()* call. Analyzing mean and standard deviation on the Table 2.3, it is concluded that OS Timer invoke TSC timer in this case. Average mean difference is about 14 nanoseconds. Presumably, this difference is the time which is spent for virtual system call of *clock_gettime().* With comparison of 16 nanoseconds of mean value of TSC invoking, 14 nanoseconds for system call can play an essential role.

Main features of tested processor:

Processor (CPU): Intel ® Core ™ i7-2600 CPU @ 3.40GHz

Speed: 1600 MHz

Cores: 8

RDTSCP instruction available

cache size: 8192 Kb

Timer source	Mean, nsec	Standard Deviation, nsec	The final mean difference, nsec
TSC	16.9091	33.1484	14.323
OS	31.2322	45.0567	

Table 2.4: Mean and standard deviation values of TSC and *clock_gettime()* call on the Intel Core i7 processor

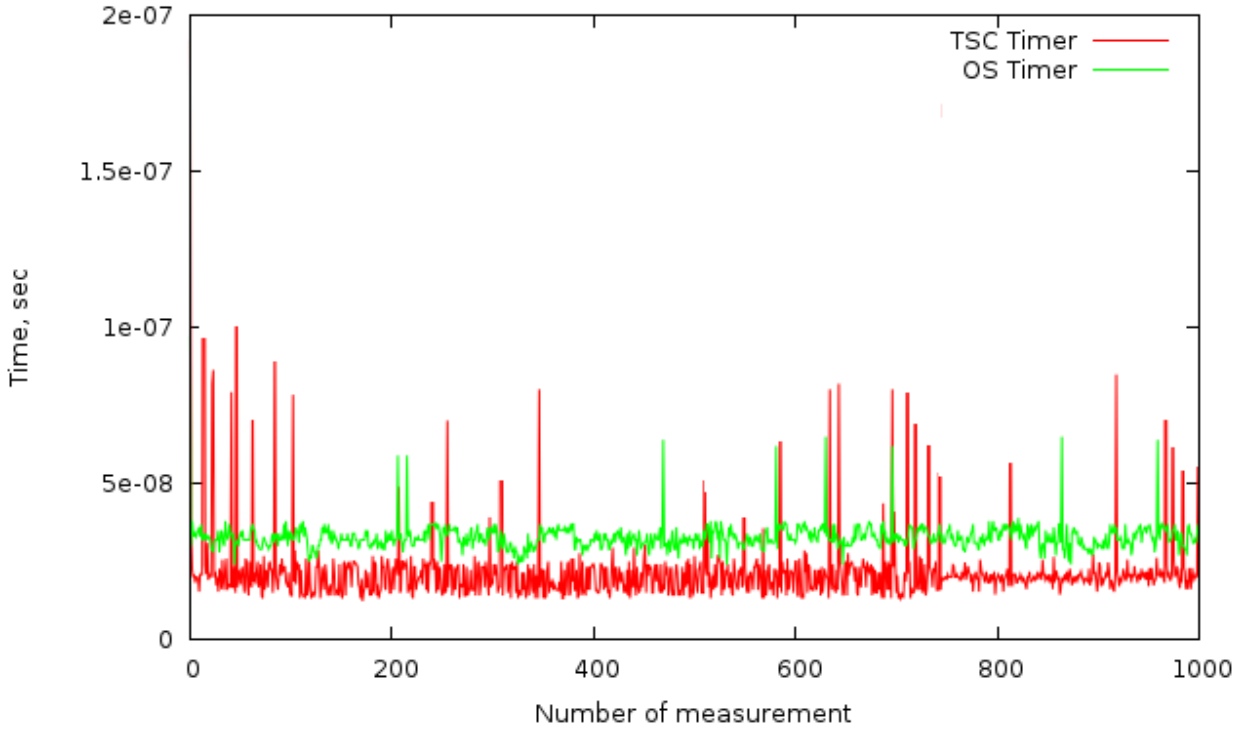

Figure 2.4: Measurements of *clock_gettime()* call with comparison of TSC on the Intel Core i7 processor

Concluding, the difference between current clock source and system time-related call varies depending on the tested processor and it is difficult to say in advance how much time is taken by real or virtual system call. The more important option is that the cost of system call and invocation current timer source is directly dependant on whether TSC or HPET is real time source. In case of HPET the mean value is more than 1 microsecond and the difference with OS Timer is only 17 nanoseconds, which is not always very significant. For case when TSC is the current clock source even 14 nanosecond can play a key role against 17 nanoseconds of mean value. Accordingly, it makes applying OS Timer for high accuracy measurement quite unreliable.

3 Logics of Timer Source Initialization

In this section the experimental comparison of available timer sources are described along with the algorithm of the decision for an appropriate time source for *HighPerTimer*.

3.1 *HighPerTimer* Source Definition

When the computer system is initially booted the Linux kernel sets its own clock source. One of ways to determine, which source using by kernel is checking the message buffer of the kernel with *dmesg* command. The example below shows a passage of *dmesg* output:

```
[  0.000000] hpet clockevent registered
[  0.867251] hpet0: 8 comparators, 64-bit 14.318180 MHz counter
[  0.869264] Switching to clocksource hpet
...
[  0.000000] Fast TSC calibration using PIT
[  2.420990] Refined TSC clocksource calibration: 3392.292 MHz.
[  2.420995] Switching to clocksource tsc
```

In the beginning, kernel is indicating hpet device and after about 2 seconds it is switching to TSC clock source. On the most Linux distributions (including OpenSuse, CentOS, Ubuntu distributions and so on) it can also be find out by checking the */sys/devices/system/clocksource/clocksource0/current_clocksource* file, where the current clock source is written in plain text.

The high priority *HighPerTimer* task is identifying the reliable time source at the early stage of initialization. Likewise the kernel, *HighPerTimer* has also the ability to choise the most suitable source. The function *HighPerTimer::InitTimerSource()* is responsible for this. In header file *TimeHardware.h* three classes with respective initialization routines has been declared - *TSCTimer*, *HPETTimer* and *OSTimer*:

- TSCTimer class ⟶ TSCTimer::InitTSCTimer()
- HPETTimer class ⟶ HPETTimer::InitHPETTimer()
- OSTimer class ——— no init routine

TSCTimer and *HPETTimer* contain the initialization functions which return true for success and false for failure. Success is meant the verifiable timer source is stable and it can be used for accurate counting, the failure means that it is should not rely on this

timer source. So the preferred timer is TSC, so that's way it is checked first:

```
void HighPerTimer::InitTimerSource()
{
    if ( TSCTimer::InitTSCTimer() )
    {
        HighPerTimer::HPTimerSource =  TimeSource::TSC;
        return;
    }
}
```

Listing 3.1: Initialization *HighPerTimer* source by TSC Timer

For example, processor VIA Nano X2 has a constant TSC rate and hence *InitTimerSource()* returns true immediately after the TSC check without *HPETTimer::InitHPETTimer()* call. However, HPET and OS Timer can also be used, but TSC timer is the most stable and the fastest one. In Figure 3.1 it is described behavior of all available timer sources in detail:

Main features of tested processor:

Processor (CPU): VIA Nano X2 U4025 @ 1.2 GHz
Speed: 1067 MHz
Cores: 2
Constant TSC rate
HPET Frequency: 14 MHz
cache size: 1024 Kb

Timer source	Mean, usec	Standard deviation, usec
TSC Timer	0.0441	0.3711
HPET Timer	0.6042	3.0271
OS Timer	0.1122	0.3755

Table 3.1: Mean and standard deviation values of HPET, TSC and OS Timer

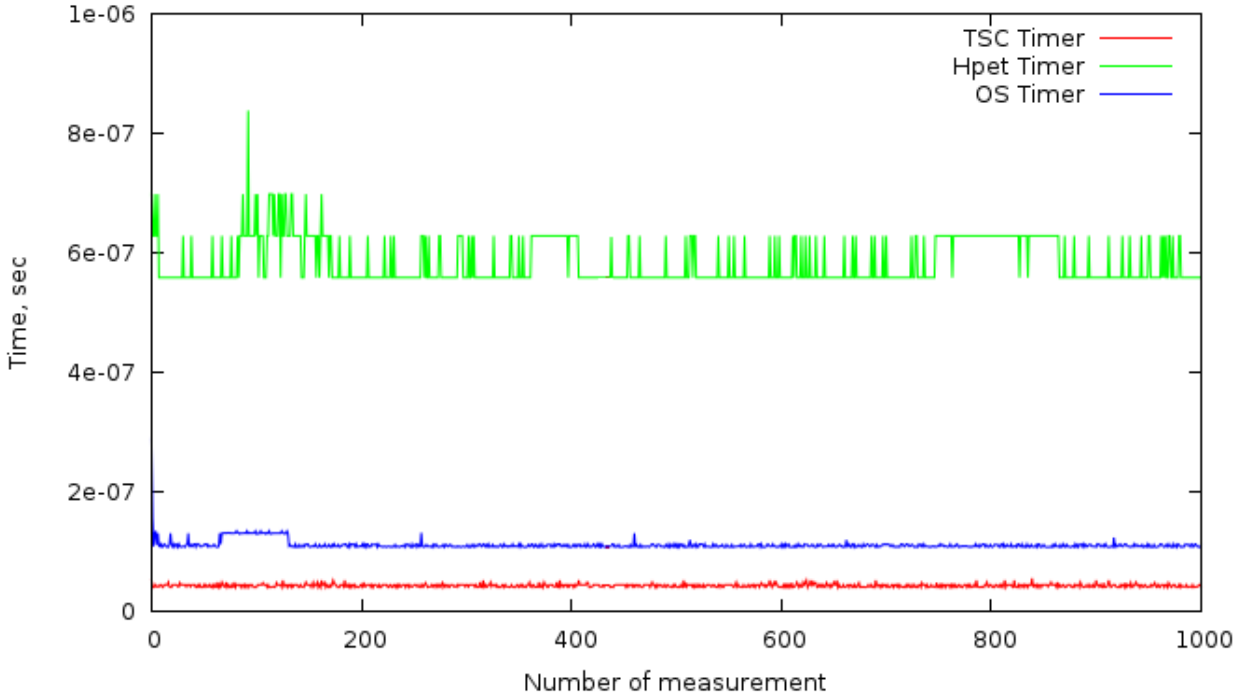

Figure 3.1: Measurements of TSC, HPET and OS Timer on the VIA Nano X2 processor

If *InitTSCTimer*() returns false, it means that TSC is unstable and can't be a time source for the library. In that case there are two more options – HPET or OS Timer. However, there is no guarantee which hardware timer is used by *clock_gettime()* call and regarding this point it is necessary to check the mean value of getting ticks cost of both timers. The cost of getting timer tics means how long it is spent to set timer to actuall value and extract its time value. If mean value of timers differ by less than 25 %, there is a sense to estimate timers by standard deviation value:

```
// mean values
double MeanHpet ( 0 ), MeanOs ( 0 );
// standard deviation values
double StDevHpet ( 0 ), StDevOs ( 0 );
// percents value means a limit which is compared with
double Limit (25.0);

// calculate percentage of means
if ( MeanHpet < MeanOs )
{
    Percentage = 100 - ( MeanHpet / MeanOs * 100 );
}
else
```

```
{
      Percentage = 100 - ( MeanOs / MeanHpet * 100 );
}
// when the mean values are similar (the difference is no more than
25%), compare deviation values of timers
if ( Percentage < Limit )
{
   if ( StDevHpet < StDevOs)
   {
       HighPerTimer::HPTimerSource =  TimeSource::HPET;
       return;
   }
   else
   {
      HighPerTimer::HPTimerSource =  TimeSource::OS;
      return;
   }
}
```

Listing 3.2: Estimate of mean and deviation values of HPET and OS Timers

As an example of unstable TSC source, a processor AMD Athlon X2 Dual Core can be examined. The RDTSCP instruction is available, but the system has neither Invariant TSC flag nor constant TSC, so *TSCTimer::InitTSCTimer()* returns false. HPET device with frequency of 25 MHz is also accessible. According to Figure 3.2, OS Timer and to be more preciously, *clock_gettime()* invokes HPET timer as a source. Mean values of this both timers are close to each other and they differ by about 2%. In this case it is necessary to compare standard deviation o their values also and ultimately for this processor default *HPTimerSource* is equal to HPET Timer.

Main features of tested processor:

Processor (CPU): AMD Athlon ™ X2 Dual Core Processor BE-2350
Speed: 1000 MHz
Cores: 2
RDTSCP instruction available
HPET Frequency: 25 MHz
cache size: 512 Kb

Timer source	Mean, usec	Standard deviation, usec
TSC Timer	0.0275	0.0756
HPET Timer	1.1179	0.3764
OS Timer	1.1423	0.5292

Table 3.2: Mean and standard deviation values of HPET, TSC and OS Timer on the AMD Athlon processor

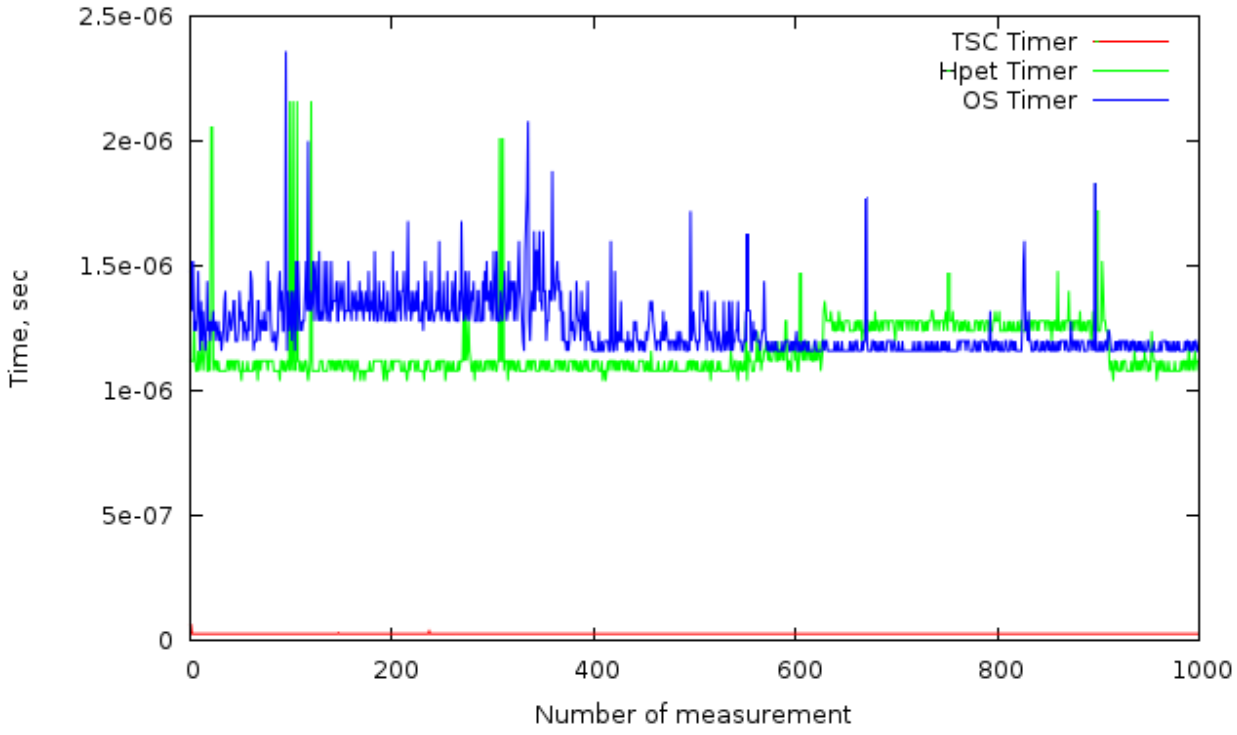

Figure 3.2: Measurements of TSC, HPET and OS Timer on the AMD Athlon processor

3.2 Change of the Time Source for *HighPerTimer*

Initializing *HighPerTimer* means not only to define its timer source. This concept also includes calculating of timer frequency, the value of shift against Unix Epoch, Max and Min values for *HighPerTimer* and determining HZ frequency of the kernel. These values are necessary for further operations with time value and overall design is described in Chapter 5. In fact, these variables must be defined primarily, in a strict order before and have unchanged values globally across the entire library scope. In other words, they should be allocated statically. The listing below shows the order of *HighPerTimer* initialization:

```
HighPerTimer::InitTimerSource();
HighPerTimer::InitHPFrequency();
```

```
HighPerTimer::InitUnixZeroShift();
HighPerTimer::InitMaxMinHPTimer();
HighPerTimer::InitSecPerJiffy();
```

Listing 3.3: Order of *HighPerTimer* initialization

All these functions are static and determine appropriate parameters. *InitHPFrequency*() sets the time counter frequency and calculates the value of *NsecPerTic* - number of nanoseconds within one counter period. *InitUnixZeroShift()* sets the *HighPerTimer* counter offset against the Unix zero time - 1 January 1970. *InitMaxMinHPTimer()* initializes max and min *HighPerTimer* values which save values in seconds, nanoseconds and in timer tics. Both they are public, accessible for user from main routine and saved as instances of *HighPerTimer* objects: *static HighPerTimer HPTimer_MAX* and *static HighPerTimer HPTimer_MIN*. The biggest possible value in tics is always equal to the biggest value for 64-bit integer type. To calculate max value in seconds, it is divided to timer frequency or multiplied to reciprocal value of *NSecPerTic*. In case of HPET the minimum possible frequency is 10 MHz, so the max possible value of *NSecPerTic* is 100 and it can be caused a possible overflow and a loss of accuracy calculating seconds and nanoseconds max/min parts of timer. So in HPET case it is more reliable to decrease appropriate limits for max and min *HighPerTimer* values. Being more precise, these limits are decreased on 1 femptosecond (1e-15 sec), which is equal to 1e-6 nanosecond. In case of TSC or OS timer, the *NSecPerTic* value is always less than zero. So it is still safe to values within max and min values of 64-bit integer type. *InitSecPerJiffy()* obtains 1/HZ value, which needs for correct sleep performance. The number of ticks since the system started is recorded in a variable called jiffies in the kernel, which size is determined by the value of the kernel constant HZ.

Sometimes there are some cases, when user would prefer some particular timer source which doesn't coincide with already initialized timer source. For that it is provided a special ability to change default timer - *HighPerTimer::SetTimerSource (const TimeSource UserSource).* This feature should be used with caution only at system initialization time, and in any case before instantiation of the first *HighPerTimer* object. This is very important, because, as it was described above, a number of global parameters are directly dependent upon this source. So when a change of timer source occurs, recalculation of most parameters also occurs, leading to invalidation of all the already existing timer objects within the executed program.

4 Software Design of The Unified Time Handling

Software design is a process of problem solving and planning for a software solution. When the purpose and specifications of software are determined, a plan for a solution has to be developed. It includes low-level components and algorithm implementation issues as well as the architectural view. There are many aspects to consider in the design of a piece of software. The importance of each should reflect the goals the software is trying to achieve. The most significant guideline about designing interfaces of any kind: that they should be easy to use correctly and hard to use incorrectly. That sets the stage for a number of more specific guidelines addressing a wide range of topics, including correctness, efficiency, encapsulation, maintainability, extensibility, and conformance to convention [35 p. 79].

1. Platform-independent model

In general, a software design may be platform-independent or platform-specific, depending on the availability of the technology called for by the design. In fact, *HighPerTimer* tool depends of a specific technological platform. But due to its implementation it is allowed to perform it on different 64-bit and 32-bit processors of Intel, AMD, VIA and ARM processors, considering their features and specialities. The most special case relates in this case to the ARM processor. ARM's embedded 32-bit processor architecture is found within cell phones, automotive systems, industrial computers, and other devices [36]. Comparing with familiar Intel and AMD processors, the ARM architecture specifies on other instructions set, other registers and different content of CPUID coprocessor register (although ARM architectures have a CPUID coprocessor register for the same purpose) [37 B3-713]. Furthermore, it posses neither TSC timer nor HPET device and to avoid possible compilation errors, it is used preprocessor commands like *#ifdef ARM.*

2. Encapsulation and Control Hierarchy

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding. Data encapsulation is a mechanism of bundling the data, and the functions that use them and data abstraction is a mechanism of exposing only the interfaces and hiding the implementation details from the user. C++ supports the properties of

encapsulation and data hiding through the creation of user-defined types, called classes [38 p. 52].

On the Figure 4.1, it is shown a structure that represents the organization of program components and implies a hierarchy of control. *HighPerTimer* tool includes two header files and two implementation files called *HighPerTimer* and *TimeHardware*. Each of them contains three classes. *TimeHardware* module is designed so, that information contained within it, is inaccessible to user. There is an access only through *AccessTimeHardware* class from *HighPerTimer* module.

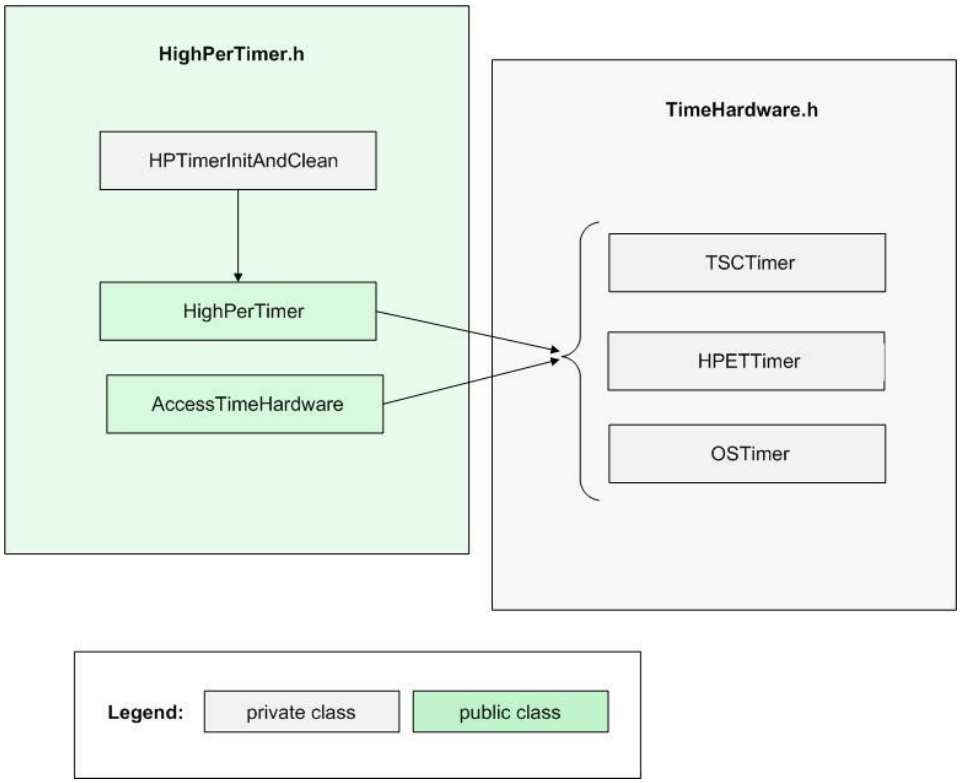

Figure 4.1: *HighPerTimer* control hierarchy

HPTimerInitAndClean is an also closed class, it defines a strict order of *HighPerTimer* initialization to control it and have the one instance in implementation *HighPerTimer* process. *HPTimerInitAndClean* class performs one more important operation. It is handling with HPET descriptor. When it is invoked the process of initialization HPET device, the descriptor is saved in a static variable. Accidentally terminating the program, HPET device can stay opened and *HPTimerInitAndClean* destructor provides correct closure. The correct closure of file is guaranteed because destructor runs every time when an object is destroyed.

3. Correctness and error handling

Error handling is fundamental. Any good program makes use of a language's exception handling mechanisms. It provides correctness of performance and allows avoiding a lot of runtime problems. An exception is a situation in which a program has an unexpected circumstance that the section of code containing the problem is not explicitly designed to handle. In C++, exception handling is useful, because it makes it easy to separate the error handling code from the code written to handle the chores of the program. Doing so makes reading and writing the code easier.

The most common potential problem for *HighPerTimer* is an type overflow of an integer variable and memory allocation error. Mostly it is related to storing time values in seconds (or nanoseconds) and in counter tics together. It's well-known, that counter tics are equal to seconds plus nanoseconds and multiplied to timer frequency or divided by the reciprocal value, accordingly to listing 4.1. In this situation, there is a high hazard of overflow *int64_t mHPTics* variable if adjusted value of *mSeconds* and *Nseconds* are too big. It is important to consider it creating an *HighPerTimer* object and working with any arithmetic operations and exceptions provide a way to react to this exceptional circumstances. This case is the most common for *HighPerTimer* and a lot of source code was devoted to throw an exception reporting an out-of-range error:

```
// check for possible overflow according to max and min value of
HighPerTimer
if ( ( mSeconds * ONE_BILLION + mNSeconds ) > ( HPTimer_MAX.Seconds()
* ONE_BILLION + HPTimer_MAX.NSeconds() ) )
{
   throw std::out_of_range ( "HPTimer overflow" );
}
mHPTics = ( mSeconds * ONE_BILLION + mNSeconds ) / ( NsecPerTic );
```

Listing 4.1: Example of throwing out of range exception

There are no any error outputs in log files in a scope of *HighPerTimer* library at all, because the record process can take time up to milliseconds, which is cruel for operations with high-precision timer.

4. Conformance to convention

C++11 is the most recent iteration of the C++ programming language. It was approved by ISO on 12 August 2011, replacing C++03 [39]. C++11 includes several additions to

the core language and extends the C++ standard library, incorporating most of the C++ Technical Report 1 (TR1) libraries — with the exception of the library of mathematical special functions. C++11 was published as *ISO/IEC 14882:2011* in September 2011. For *HighPerTimer* implementation the performance benefit, either of memory or of computational speed are primarily essential and it is provided by C++11 core language runtime performance enhancements.

- Move semantics

C++ has always supported copying object state. In contrary, moving semantics is a key new idea of C++11. A move constructor, like a copy constructor, takes an instance of an object as its argument and creates a new instance based on the original object. However, the move constructor can avoid memory reallocation, because it has been provided a temporary object, so instead of coping the fields of the object, it move them. Move semantics allows avoiding unnecessary copies when working with temporary objects that are about to evaporate, and which resources can safely be taken from that temporary object and used by another. In fact, a move operation needs to do three things: get rid of the destination's current value, move the source's value to the destination, and leave the source in a valid state [40 p. 249]. *HighPerTimer* class posses a move constructor and a move assignment operator.

```
HighPerTimer & HighPerTimer::operator= ( HighPerTimer &&  Timer )
{
    mHPTics = Timer.mHPTics;
    mNormalized = false;
    return *this;
}
```

Listing 4.2: Move assignment of *HighPerTimer* class

- Generalized constant expressions

C++ has always had the concept of constant expressions. Constant expressions are optimization opportunities for compilers, and compilers frequently execute them at compile time and hardcode the results in the program. Also, there are a number of places where the C++ specification requires the use of constant expressions. Defining an array requires a constant expression, and enumerator values must be constant expressions. However, constant expressions have always ended whenever

a function call or object constructor was encountered. C++11 introduced the keyword *constexpr*, which allows the user to guarantee that a function or object constructor is a compile-time constant [39]. Such data variables are implicitly const, and must have an initializer which must be a constant expression. The example can be written as follows:

```
// constant string for Cenatur vendor "CentaurHauls"
constexpr char CentaurVendor[12] = {'C', 'e', 'n', 't','a','u', 'r',
'H', 'a', 'u', 'l', 's'};
// constant string for Intel vendor "GenuineIntel"
constexpr char IntelVendor[12] = {'G', 'e', 'n', 'u', 'i', 'n', 'e',
'I', 'n', 't', 'e', 'l'};
```

Listing 4.3: Example of using constant expressions

Prior to C++11, the values of variables could be used in constant expressions only if the variables are declared const, have an initializer which is a constant expression, and are of integral or enumeration type. C++11 removes the restriction that the variables must be of integral or enumeration type if they are defined with the *constexpr* keyword.

- Object construction improvement

In C++03, constructors of a class are not allowed to call other constructors of that class; each constructor must construct all of its class members itself or call a common member function. Constructors for base classes cannot be directly exposed to derived classes; each derived class must implement constructors even if a base class constructor would be appropriate. Non-constant data members of classes cannot be initialized at the site of the declaration of those members. They can be initialized only in a constructor. C++11 provides solutions to all of these problems. C++11 allows constructors to call other peer constructors (known as delegation). This allows constructors to utilize another constructor's behavior with a minimum of added code [40 p. 301]. A constructor that delegates to another constructor may do not anything else on its member initialization list. This example of syntax in a scope of *HighPerTimer* is as follows. It is removed this redundancy in the forthcoming code using delegating constructors:

```
// ctor
// @param tv is the timevalue struct
HighPerTimer::HighPerTimer ( const timeval & TV ) :
    // delegating constructors
    HighPerTimer ( timespec { TV.tv_sec, TV.tv_usec * 1000 } )
{
}

// ctor
// @param ts is the timespec struct
HighPerTimer::HighPerTimer ( const timespec & TS ) :
     mSeconds ( TS.tv_sec ),
     mNSeconds ( TS.tv_nsec ),
     mSign ( false ),
      mNormalized ( true )
{
//check for possible overflow according to max and min value HPTimer
    if ( ( mSeconds * ONE_BILLION + mNSeconds ) > (
HighPerTimer::HPTimer_MAX.Seconds() * ONE_BILLION +
HighPerTimer::HPTimer_MAX.NSeconds() ) )
    {
        throw  std::out_of_range ( "HPTimer overflow" );
    }
    mHPTics = static_cast<int64_t> ( ( mSeconds * ONE_BILLION +
mNSeconds ) / ( HighPerTimer::NsecPerTic + (1/ONE_QUADRILLION ) ) );
}
```

Listing 4.4: Example of using delegating constructors

- Strongly typed enumerations

In C++03 and before, enumerations were not type-safe. They were effectively integers, even when the enumeration types are distinct. This allowed comparison between two *enum* values of different enumeration types. The only safety that C++03 provided was that an integer or a value of one *enum* type does not convert implicitly to another *enum* type. Additionally, the underlying integral type is implementation-defined; code that depends on the size of the enumeration is therefore non-portable. Lastly, enumeration values are scoped to the enclosing scope. Thus, it is not possible for two separate enumerations to have matching member names. C++11 allows a

special classification of enumeration that has none of these issues. This is expressed using the *enum* class (*enum* structure is also accepted as a synonym) declaration. *HighPerTimer* makes use of several enumeration classes in its scope. One of them is shown below:

```
//source of timer: TSC Timer, HPET Timer or
// the timer, provided by OS
enum class TimeSource
{
    TSC, HPET, OS
};
```

Listing 4.5: Example of using enumeration class

This enumeration is type-safe. *Enum* class values are not implicitly converted to integers; therefore, they cannot be compared to integers either (the expression *Enumeration::OS == 2* gives a compiler error).

5 Precise Sleep Timer Aspects

To make process sleep it is mostly used two ways: standard C Library system sleep or sleep in busy-waiting loop. In this section it is given brief description of advantages and pitfalls of each method with comparison of newly implemented *HighPerTimer* sleep.

5.1 System Sleeps Implementation

C Library provides possibilities to suspend thread execution for the specified number of seconds, microseconds or nanoseconds. Declared in *unistd.h,* the function *sleep()* (analogically *usleep*() and *nanosleep*()) gives a simple way to make the program wait for a short interval. *sleep*() makes the calling thread sleep until seconds have elapsed or a signal arrives which is not ignored. It returns zero if the requested time has elapsed, or the number of seconds left to sleep, if the call was interrupted by a signal handler. If the call is interrupted by a signal handler or encounters an error, then it returns -1, with *errno* set to indicate the error [41, 42, 43].

Linux kernel keeps waiting threads in a special queue which afford to leave CPU free. It means that CPU usage during system sleep is close to zero. This is the main advantage of system sleeping. However, comparing the deviation value (or so called miss), it can be seen a big pitfall (Table 5.1). It was implemented system sleep from 1 second until 1 microsecond. Measurements are performed with *HighPerTimer* on the Intel Core -i7 processor with HZ = 1000.

Obviously, miss value depends some way on the sleep time value. After some point its values are aligned and in average are equal to 53 microseconds, which unacceptably for high performance measurements. Nevertheless, at this point it is important to understand, how system sleep works and where it is spent so much time. On the Figure 5.1, it is shown a simplified scheme of executing processes (in particular sleeping process) for a general overview.

Sleep time, sec	Miss, usec
1	161,135
0,5	112,908
0,25	92,139
0,125	100,193
0,0625	74,393
0,03125	70,981
0,015625	68,225
0,007812	85,311
0,003906	66,612
0,001953	64,948
0,000976	59,379
0,000488	64,359
0,000244	55,422
0,000122	53,916
0,000061	53,436
0.000030	53,439
0,000015	53,318
0,000007	53,269
0,000003	53.250
0,000001	53,267

Table 5.1: Miss measurements of system sleep, performed with TSC on the Intel Core i7 processor, HZ = 1000

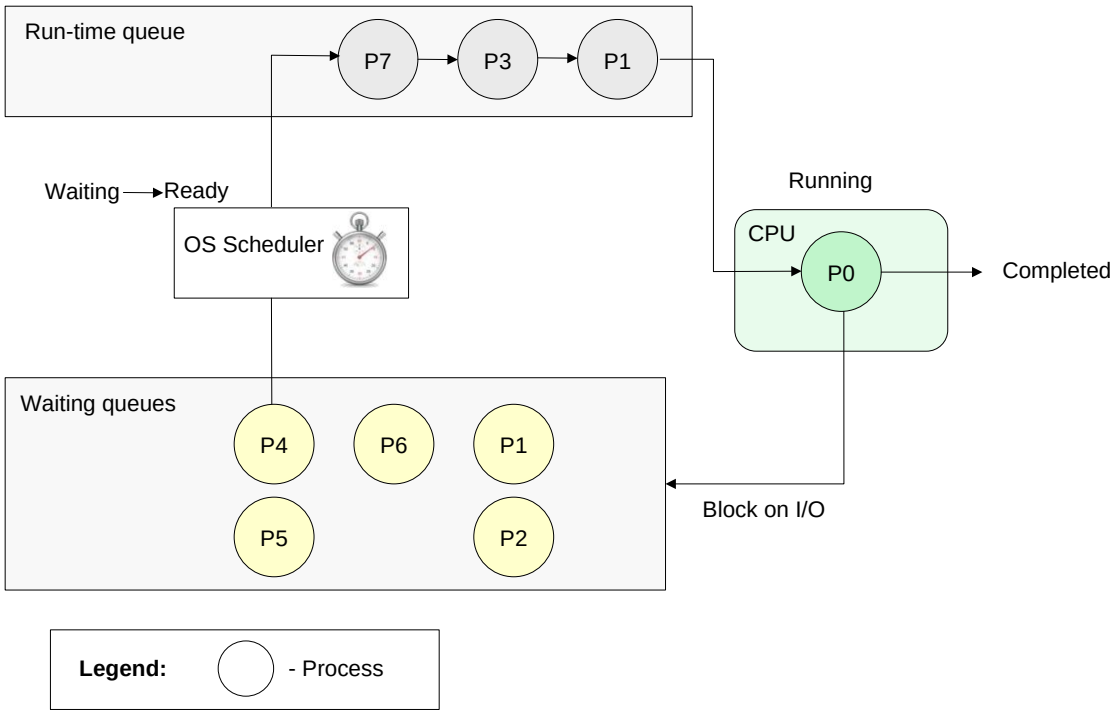

Figure 5.1: Simplified scheme of executing processes in Linux kernel

The *Run-time queue*, also known as a *Ready list*, in the operating system keeps a list of all processes that are ready to run and not blocked on some I/O or other system request, such as a semaphore. The entries in this list are pointers to the process control block, which stores all information and state about a process. When some process execution is suspended, the first place where it is going is the *Run-time queue.* After that, according to its priority, it is getting to the *Wait queue*, passing the CPU. The key point is that process is waiting for adjusted time not on CPU, but on the *Wait queue.* After sleep time is elapsed, process moves from the *waiting* state to the *ready* state, gets placed on the *Ready list* and completes its execution [44, 45]. This whole process is controlled by the *process scheduler.*

The *process scheduler* is the component of the operating system that is responsible for deciding whether the currently running process should continue running and, if not, which process should run next, including a responsibility to alarm sleeping process [46 Chapter 10.2]. It runs with a frequency of HZ, it means that it is able to check state of suspended process every 1/HZ seconds.

HZ (hertz) is the frequency with which the system's timer hardware is programmed to interrupt the kernel and defined as the number of cycles per second. Much of the kernel's internal housekeeping, including process accounting, scheduler time slice accounting, and internal time management, is done in the timer interrupt handler. Thus, the frequency of the timer interrupt affects a number of things; in particular, it puts an upper bound on the resolution of timers used with the kernel. The value of HZ is configurable at kernel compilation time. The actual used HZ value can be extracted from and defined in */proc/config.gz.* The value of HZ varies across kernel versions and hardware platforms and can be equal to 1000, 300, 250 or 100. If HZ is 1000 (the i386 default for 2.6 kernels through 2.6.12), then timers will have a best-case resolution of 1ms. If, instead, HZ is 100 (the 2.4 and prior default), that resolution is 10 milliseconds. The 250 HZ default in 2.6.13 gives a maximum timer resolution of 4ms and since kernel 2.6.20, a further frequency is 300 Hz and respectively resolution is 3 milliseconds.

Executing system sleep is managed by scheduler and it means is directly dependent upon the HZ tick rate. It has a kind of limit and when sleep time is less than 1/HZ, system sleep miss is always about 53 microseconds. For accurate precise sleep this way is unsatisfactory.

5.2 Implementation Sleep in Busy-wait Loop

Alternative way for sleeping, which was used by old *CTSCTimer*, is sleeping in the so called mode of *busy-waiting*. This approach defines the way of waiting for an event by spinning through a tight loop or timed-delay loop, that polls for the event on each pass, as opposed to setting up an interrupt handler and continuing execution on another part of the task. This is a wasteful technique which hogs the processor.

There are several ways to improve the performance of spin-wait loops or, in other words, make processor do nothing. In the Intel specification, it is recommended to use the *pause* instruction, which can be executed on a Pentium 4 or Intel Xeon processor. On a Pentium 4 processor, it provides the added benefit of reducing processor power consumption while executing a spin-wait loop [13 vol. 1 11-18]. On older processors, this instruction operates as a *nop*. The *nop* instruction performs no operation and usually it is used with *rep* (repeat the next instruction). "*rep; nop"* is indeed the same as the *pause* instruction. It might be used for assemblers which don't support the *pause* instruction. On previous processors, this was simply did nothing, just like *nop* but in two bytes. For example, ARM architecture doesn't support either *rep* and *pause* instructions. On new processors which support hyper threading, "*rep; nop"* is used as a hint to the processor that a spin loop is executed to increase performance. The example of busy-waiting implementation is shown below:

```
// TargetTics means current tics and given sleep time
while ( HighPerTimer::GetTimerTics() < TargetTics )
{
    asm volatile
    (
        "rep;nop"
     );
}
```

Listing 5.1: Busy-waiting implementation

Such sleep hogs 100% of CPU performance, but according to Table 5.2 has the minimum of possible miss value. Measurement of Table 5.2 was performed on the platform with HZ = 1000. It can be noticed that the miss value is aligned after sleep time of 0.001953 seconds and are equal to about 50 nanoseconds, although it isn't evident on this table, but actually miss value is directly dependent upon 1/HZ value.

Sleep time, sec	Miss, nsec
1	334
0,5	243
0,25	159
0,125	169
0,0625	105
0,03125	98
0,015625	91
0,007812	67
0,003906	62
0,001953	79
0,000976	56
0,000488	46
0,000244	53
0,000122	61
0,000061	38
0.000030	60
0,000015	55
0,000007	50
0,000003	52
0,000001	44

Table 5.2: Miss measurements of sleep in busy waiting loop, performed with TSC on the Intel Core i7 processor, HZ = 1000

In case described above, when sleep time is more than 1/HZ, the miss value is directly proportional to sleep time. When sleep time is less than 1/HZ, the miss value is approximately equal to 50 nanoseconds, which is quite good for high-accuracy sleeping.

5.3 *HighPerTimer* Sleeps Realization

5.3.1 Main Issues of *HighPerTimer* Sleeps

HighPerTimer sleep is a new combined sleep mode which unifies system sleep and sleep in busy-waiting loop. The purpose is combining all advantages from each way of sleep, matching minimum CPU usage and minimum miss of nanoseconds. The main idea is dividing total sleep time in two parts, subtracting 1/HZ value like it is shown on the Figure 5.2

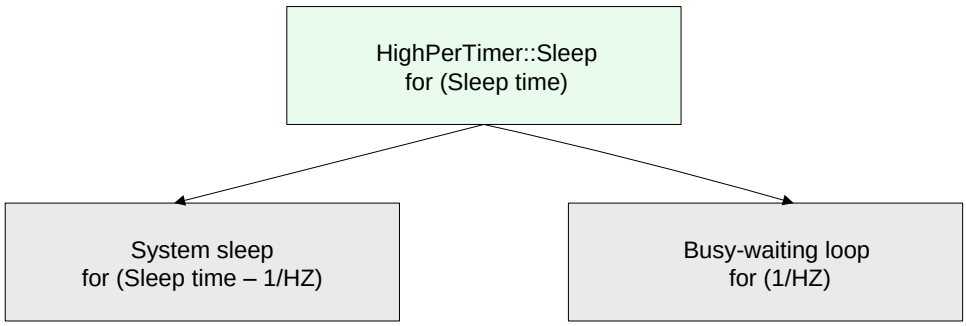

Figure 5.2: Division of sleep time for *HighPerTimer* sleep

Before subtracting appropriate value of 1/HZ, it should be determined in a scope of *HighPerTimer*. It is preferred not to read */proc/config.gz*, but find more flexible solution, not dependent on the file existence, moreover since some kernel configurations hide this information from the user. So the best way is calculating CPU usage during busy-waiting loop with different value of sleep time.

For obtaining resource usage, it was chosen to use *getrusage()* function specified in <*sys/time.h*> and in <*sys/resource.h*> [47]. The example of getting CPU time is shown below on the Listing 5.2. *RUSAGE_ THREAD* argument specifies to return resource usage statistics for the calling thread. Usages are returned in the struct *rusage* pointed by *ru* in this example. *ru_utime* field keeps user CPU time used and *ru_stime* – system CPU time used. The difference is whether the time is spent in user space or in kernel space. User CPU time is time spent on the process running program's code in user mode. System CPU time is the time spent running code in the operating system kernel on behalf of a program (via a system call).

```
struct rusage ru;
struct timeval tim;
double CpuTime;

getrusage(RUSAGE_THREAD, &ru);
tim = ru.ru_utime;
CpuTime = tim.tv_sec + (double)tim.tv_usec / ONE_MILLION;
tim = ru.ru_stime;
CpuTime += tim.tv_sec + tim.tv_usec / ONE_MILLION;
```

Listing 5.2: Obtaining CPU time

Analyzing how kernel measures CPU usage with different platform, it was found out some dependencies, which are quite straight forward. Each hardware timer has its own resolution (more precise this resolution is equal to 1/HZ). For example, platform with

HZ=1000 has resolution of 0.001 seconds and measure CPU usage well when sleep time is 0.01, 0.02, 0.03 and so on. If sleep time is, for example, 0.0025, the total CPU usage time will be predominantly equal to zero predominantly and sometimes 0.01. It means that values below or above resolution are inappreciable. On the Figure 5.3 it is demonstrated measurements of CPU usage during sleeping in busy-wait loop for appropriate sleep time for different kind of platforms. In general they represent quite precise value, but there are still some deviations.

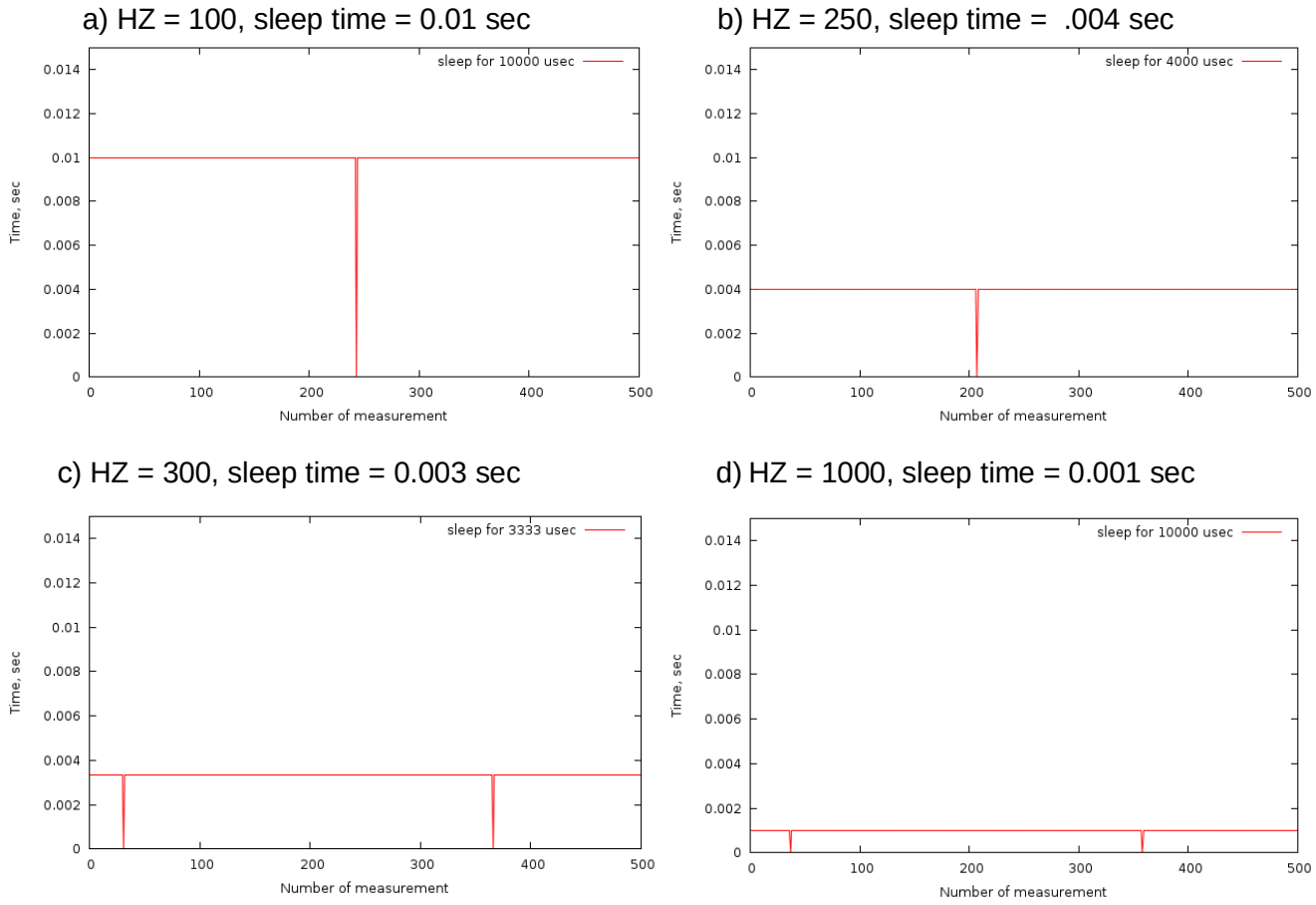

Figure 5.3: Measurements of CPU usage during busy-waiting on different platforms with different sleep time value

To simplify and speed up determination of HZ in a scope of *HighPerTimer*, it was chosen trade-off value of sleep time - 14500 microseconds for all kind of platforms. With this value it is possible to distinguish and identify HZ faster. On the Figure 5.4, it is shown that CPU usage values are not uniform, but each line has one stable value which dominates. For HZ = 100, this dominate value is 10 milliseconds (unlikely 20); for HZ = 250 it is gotten 16 millisecond (unlikely 12); for HZ = 300, it is 13.3 milliseconds (unlikely 16); for HZ = 1000, it is 14 milliseconds (unlikely 15).

HZ value	CPU usage, msec
100	10
250	16
300	13.3
1000	14

Table 5.3: The most probable values of CPU usage during sleeping in busy-wait loop for 14500 microseconds.

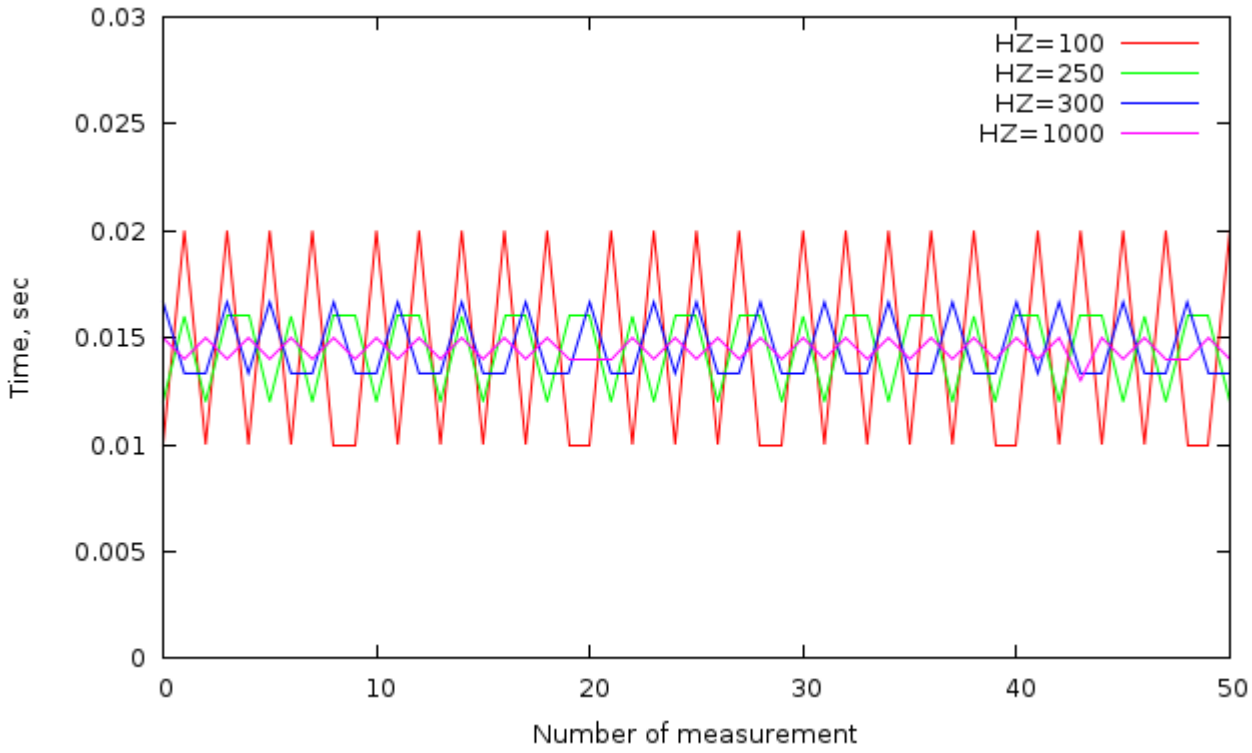

Figure 5.4: Measurements of CPU usage during busy-waiting for 14500 microseconds sleep time in different platforms

5.3.2 Performance- and Accuracy-optimization of HighPerTimer Sleeps

HighPerTimer posses several different ways to make thread sleep:

- sleep for the amount of corresponding time in microseconds;
- sleep the amount of corresponding time in nanoseconds;
- sleep the amount of time in tics which the corresponding timer is set to;

Moreover, *HighPerTimer* provides functions passing parameters of sleep time in different ways. The next example lists the *HighPerTimer::USecSleep()* function:

```
void HighPerTimer::USecSleep ( const uint64_t USeconds ) const
{
    int64_t TargetTics (  HighPerTimer::GetTimerTics() + ( USeconds *
1000 / ( HighPerTimer::NsecPerTic ) ) );
    // calculate 1/HZ in microseconds for system sleep, HPJiffies is
equal to 1/HZ in seconds
    uint64_t BusyUSeconds ( static_cast<uint64_t>
(HighPerTimer::HPJiffies * 1000000  ));
    if ( USeconds >=  BusyUSeconds )
    {
        usleep( USeconds - BusyUSeconds );
    }
// the rest time is spending in busy-wait loop
    while ( HighPerTimer::GetTimerTics() < Target )
    {
          RepNop();
    }
    return;
}
```

Listing 5.3: *HighPerTimer* sleep for given amount of microseconds

This operation is determined according to Figure 5.2. At first it is calculating 1/HZ value in microseconds (*BusyUSeconds*), and then it is subtracted from given initial sleep time (*USeconds*) for system sleep. After system sleep, the busy-wait loop is beginning, which duration is calculated due to initially value of *TargetTics,* which calculated in the very beginning of function. Here it is implied that real time spent in busy-waiting loop should be equal not only 1/HZ (as it was specified in Figure 5.2), but 1/HZ subtracted redundant time of system sleep or, in other words, 1/HZ minus system sleep miss. After successful implementation, *HighPerTimer* sleep is evaluated by two parameters – average miss value and CPU utilization:

1. At first, new combined sleep is evaluated by the miss value. The results are shown on the Table 5.4. For better representation it is shown two kinds of them – *Run 1* and *Run 2,* which perform with the same condition. After sleep time is less than *1/HZ,* the miss value varies from 50 until 100 nanoseconds constantly. To demonstrate this behavior *Run 1* and *Run 2* columns were added.

Sleep time, sec	Run 1	Run 2
	Miss, nsec	Miss, nsec
1	253	397
0,5	304	266
0,25	268	290
0,125	285	265
0,0625	239	281
0,03125	254	242
0,015625	208	230
0,007812	284	226
0,003906	311	236
0,001953	258	182
0,000976	113	161
0,000488	48	128
0,000244	48	114
0,000122	42	89
0,000061	48	96
0,00003	46	85
0,000015	46	100
0,000007	41	83
0,000003	38	114
0,000001	49	112

Table 5.4: Miss measurements in combined sleep, performed with TSC on the Intel Core –i7 processor, HZ = 1000

When sleep time is more than 1/HZ (according to the Table 5.4, the interval from 1 seconds to 1953 microseconds of sleep time), some tendency is observed. To demonstrate it more clearly one more measurements have been performed (Figure 5.5). Current results show behavior of miss during sleeping from 10 seconds until 1 microsecond. When sleep time is more than 1/HZ = 0.001 sec, miss value is increasing by some tendency. This dependency is also observed during sleep in busy-wait loop and in system sleep, but for better understanding the mean value has been calculated (Table 5.5).

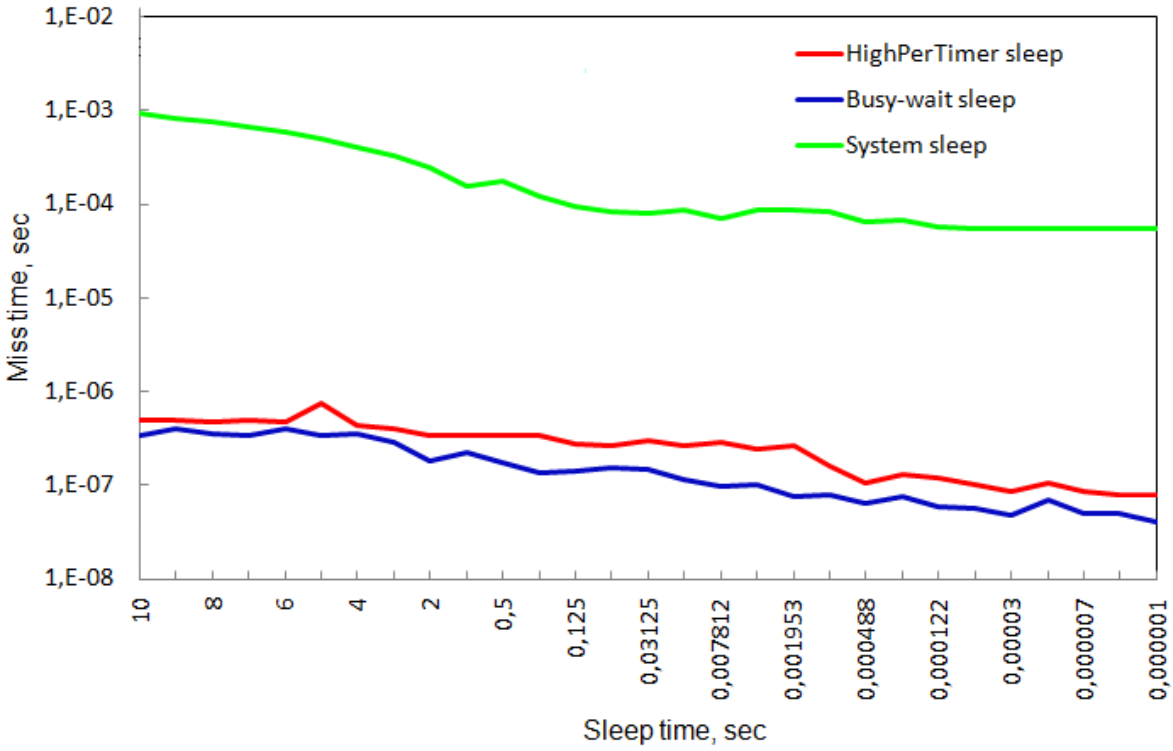

Figure 5.5: Dependency of miss on sleep time, performed with TSC on the Intel Core –i7 processor, HZ = 1000

On the Table 5.5 it is shown results of average miss value for each way of sleeping (in particular system sleep and sleep in busy-wait loop). The loop consists of 10000 steps and total performance time of measurements is equal about 830 minutes.

	Sleep time >= 1/HZ	**Sleep time < 1/HZ**	
	Mean miss, usec	**Mean miss, usec**	**Standard Deviation, usec**
System sleep	61.985	50.879	11.412
Busy-waiting loop	0.160	0.070	0.0400
HighPerTimer sleep	0.258	0,095	0.0404

Table 5.5: Miss value measurements of system sleep, busy-waiting sleep and *HighPerTimer* sleep, performed with TSC on the Intel Core –i7 processor, HZ = 1000

The mean value of *HighPerTimer* sleep miss with sleep time more than 1/HZ is 258 nanoseconds, which is quite bigger than the mean value of pure busy-waiting loop with 160 nanoseconds in the same interval. There is no sense to calculate standard deviation for case, when sleep time is more than 1/HZ, because miss value is dependent on sleep time by tendency according to Figure 5.5.

When sleep time is less than 1/HZ, the miss is also bigger and the difference is

about 25 nanoseconds. The deviation value for this interval is about 400 nanoseconds. In fact, miss of *HighPerTimer* sleep is really vary from 100 nanoseconds to 50 nanoseconds (like it is shown on the Table 5.4), but in busy-waiting sleep this value is more monotonic and stable. This behavior can be observed on the Figure 5.6, where it is compared miss value of *HPsleep* (red line) and miss after sleep in busy-waiting loop (green line). It is performed only with sleep time less than 1/HZ. The results were obtained on Intel ® Core ™ i7-2600 CPU processor:

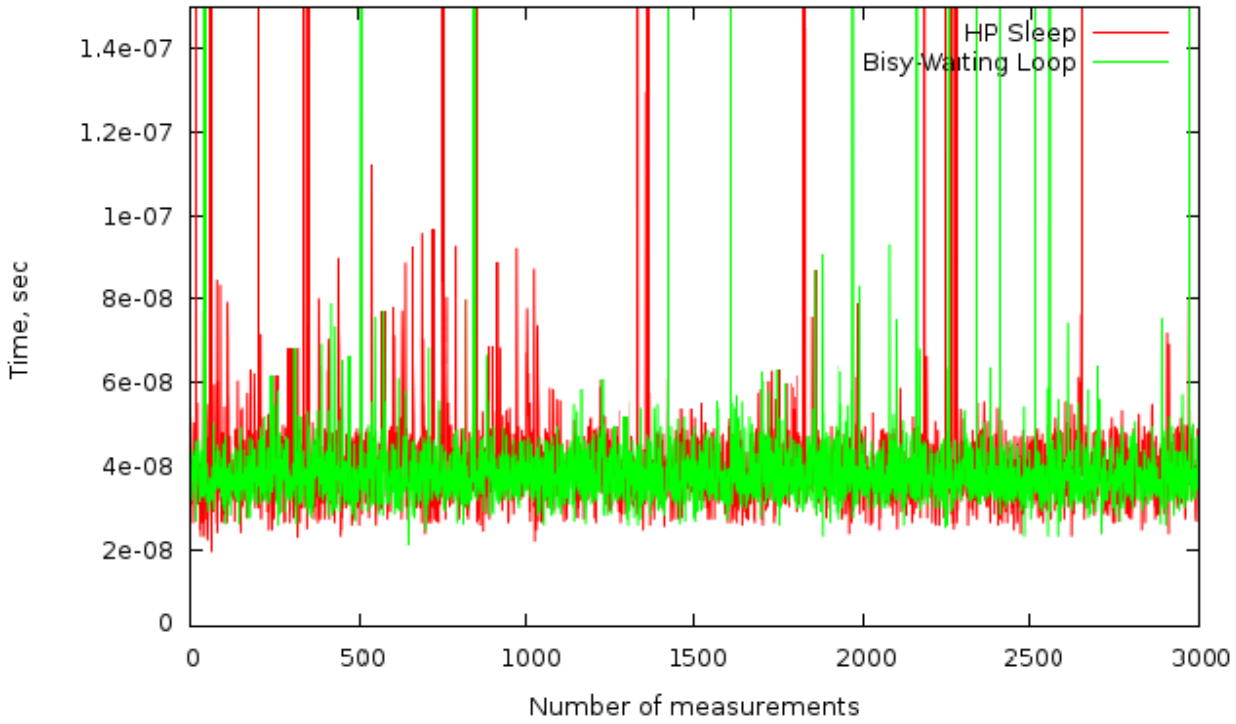

Figure 5.6: Comparison of miss during *HPSleep* and during sleep in busy-waiting loop when sleep time is less than 1/HZ

According to Figure 5.6, on the interval of measurements from 1000 to 2000, both miss values have approximately the same stable behavior and are about 50 nanoseconds. However on the interval from 0 to 1000 of measurements, miss of *HPSleep* is noticeably higher than busy-waiting sleep, about 90 nanoseconds.

To identify place in source code, where it is lost the accuracy, 4 measuring points have been set:

- T1 - the first point is set before invoking of *USecSleep()* in main routine;
- T2 - the second point is set in the initial source code of *USecSleep()* (Listing 5.3) in implementation file *HighPerTimer.cpp* after the Target calculating;

- T3 - the third point is set in the initial source code of *USecSleep()* (Listing 5.3) before the exit of the function (being more precise before keyword *return*);
- T4 - the fourth point is set after invoking of *USecSleep()* in main routine;

Structural these measuring points can be represented the way like it is shown on Figure 5.7. To set measuring points it is used method of getting current tics *HighPerTimer::Now():*

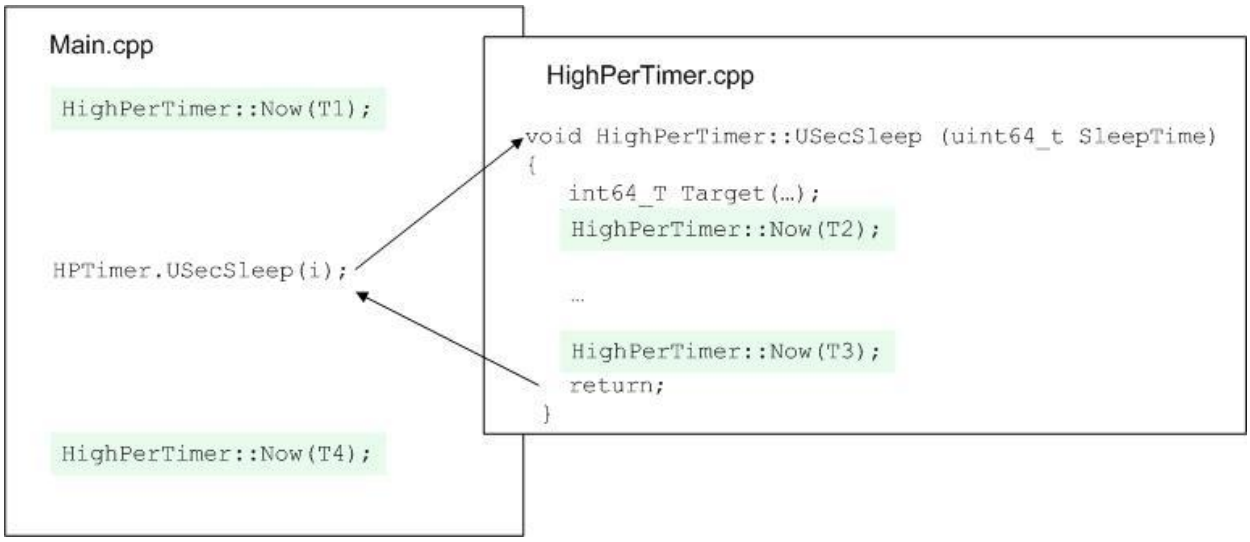

Figure 5.7: Measuring points invoking *USecSleep()*

On the Table 5.6, it is shown results of measuring time domain between T1 and T2 points, time domain between T3 and T4 points and between T2 and T3.

Sleep time, sec	T4 – T1, nsec	T3 – T2, nsec	T2 - T1, nsec	T4 - T3, nsec
1	391	16	143	273
0,5	389	15	133	241
0,25	328	12	75	241
0,125	353	31	130	213
0,0625	115	-14	78	52
0,03125	159	21	36	103
0,015625	210	10	78	123
0,007812	216	16	78	123
0,003906	205	8	75	123
0,001953	223	22	78	123
0,000976	126	21	75	30
0,000488	86	23	42	20
0,000244	79	21	40	18
0,000122	87	34	35	18
0,000061	66	13	35	18
0,00003	70	18	35	18
0,000015	67	17	35	15
0,000007	71	14	40	18
0,000003	81	29	35	18
0,000001	67	14	38	15

Table 5.6: Miss measurements of system sleep, busy-waiting sleep and *HighPerTimer* sleep, performed with TSC on the Intel Core –i7 processor, HZ = 1000

Decrypting Table 5.6, it is implied that the second column "*T4 – T1"* means real miss value, which it is considered before, but it should be considered that these values are increased on two extra operations of obtaining tics. The third column "*T3 – T2"* means miss, obtained inside a function, which ideally is equal to zero, but in fact can vary around zero and sometimes be negative. This column does not show a huge deviation. The values there are quite smooth, so it means that, in a whole, function works correctly. Nevertheless, the last two columns induce suspicion. "*T2 – T1"* column includes the entry to function and calculating tics for *TargetTics* variable in particular. "*T4 – T3"* is just an exit from *USecSleep().* Moreover these two columns give a hint that probably entrance and exit from the function can be dependent upon 1/HZ. Presumably calculating tics also has this dependency. To check these assumptions is necessary to investigate assembler code, which is out of scope of this thesis.

Additionally, it was also checked *HighPerTimer* sleep performance with *inline* keyword. In theory, inline function is a function upon which the compiler has been requested to perform inline expansion. In other words, the compiler inserts the complete body of the function in every place that the function is called, rather than generating code to call the function in the one place it is defined. If the real reason of increased miss (it means with the comparison of sleep in busy-waiting loop) is entrance and exit from the function, inline expansion could help to avoid it. But in fact, against expectations, it doesn't give a significant enhancement. On the Table 5.7 it is compared two ways of implementations:

	Miss value of HighPerTimer sleep, nsec	
	Not inline expansion	**Inline expansion**
Sleep time >= 1/HZ	0.258	0.250
Sleep time < 1/HZ	0.095	0.095

Table 5.7: Miss value of *HighPerTimer* sleep with comparison of different performance

So at this stage average miss of *HighPerTimer* sleep corresponds with Table 5.5 – when sleep time is more than 1/HZ, it is about 258 nanoseconds and when sleep time is less than 1/HZ, it is about 95 nanoseconds.

2. Secondly, *HighPerTimer* sleep is estimated with the CPU usage, which is shown on the Table 5.8 and with the comparison of CPU usage during standard system sleep and sleep in busy-waiting loop on the Table 5.8. Measurements are performed with *HighPerTimer* on the Intel Core -i7 processor with HZ = 1000:

	System sleep	**Busy-waiting**
Real time	835 min 49.698 sec	833 min 18.146 sec
User CPU usage	3.619 sec	830 min 36.564 sec
System CPU usage	4.560 sec	2.115 sec
User + System CPU usage	8.179 sec	830 min 38.679 sec
Ratio of CPU usage to real time	0.016%	99.681%

Table 5.8: Average CPU usage in system sleep and in busy-waiting sleep

Test results for Table 5.8 and Table 5.9 were obtained by performing a set of sleep execution like in Table 5.1 and Table 5.2 with sleep time from 0.25 seconds to 1 microsecond. The loop consisted from 10000 steps and total performance time (raw "*Real time"*) is equal about 835-833 minutes. As it was mentioned above the

difference between *User CPU usage* and *System CPU usage* is whether the time is spent in user space or in kernel space. The last row is a total percentage. In case of sleep in busy-waiting this value is almost 100%, in case of system sleep it tends to zero.

In Table 5.9, it is added rows for executing sleep in pure busy-waiting mode (when sleep time is less than 1/HZ seconds) and for executing sleep in combined mode with involvement of system sleep (when sleep time is more than 1/HZ seconds). Accordingly, sleep in real combined mode is performed for about 830 minutes and spent about 12 minutes on CPU, which is 1,5 %. Actually this percentage is quite bigger with comparison of 0.016% (Table 5.7), presumably the reason is an inaccuracy during scheduler performance. Sleep in pure busy-waiting mode is performed only for about 3 minutes and spent all this time on CPU, which is 98%. In general total time of sleep is about 833 minutes and percentage of CPU usage relatively to real time is about 1.89%, which can be estimated as satisfactory.

	Combined sleep
Real time	833 min 18.675 sec
Real time when sleep time >= 1/HZ	830 min 0.078 sec
Real time when sleep time < 1/HZ	3 min 18.597 sec
User CPU usage	15 min 42.393 sec
System CPU usage	3.613 sec
User + System CPU usage	15 min 46.006 sec
CPU usage when sleep time >= 1/HZ	12 min 29.425 sec
CPU usage when sleep time < 1/HZ	3 min 16.581 sec
Ratio of CPU usage to real time when sleep time >= 1/HZ	1.504 %
Ratio of CPU usage to real time when sleep time < 1/HZ	98.32 %
The final ratio of CPU usage to real time	1.892 %

Table 5.9: Average CPU usage in combined sleep

Analyzing all results, presented above, it can be concluded that *HighPerTimer* sleep is appropriate for high accuracy performance. It has the miss value of about 95 nanoseconds for sleep time less than 1/HZ. Moreover, predominantly the performance of *HighPerTimer* sleep is not dependent upon processor cores and has minimum CPU utilization, being more precise only 1.89%.

5.3.3 Handling Interruption of *HighPerTimer* Sleeps

Since *HighPerTimer* library posses the mechanism for accurate sleeping, it is required the appropriate mechanism for an interruption this sleeping state. To interrupt sleeping process means terminate it at appropriate time moment with maximum possible accuracy.

Actually, the implementation of the interrupt of sleep in busy-waiting loop is trivial and was borrowed from *CTSCTimer* class. The interruption can only be called from a different thread created in main routine, accessing the same object. Though it is intended to be used in a different thread from sleeps, for performance reason it is not thread save. So the iterrupt shall be considered as an effort to wake up prematurely, but gives no guarantee on this. The interruption is provided by function *HighPerTimer::Interrupt*() which set *true* value for corresponding member *volatile bool mInterrupted.* Below listing shows this implementation:

```
void HighPerTimer::Interrupt()
{
    mInterrupted = true;
    return;
}
```

Listing 5.4: Interruption function called from a different thread

Meanwhile, sleeping process is spinning in a loop, which allows putting some conditional operator inside a loop. More precise, the loop contains the check whether *mInterrupted* member was changed to "*true"* value. Avoiding too often access to *mInterrupted* member, it is created local *variable long Counter.* It increments on each step and allows to make the check only on every 16th steps, since the *mInterrupted* is always on the stack. Below example is a passage from *HighPerTimer* sleep demonstrating an interruption process during sleep:

```
long Counter ( 0 );
mInterrupted = false;

while ( HighPerTimer::GetTimerTics() < Target )
{
     // provides access to mInterrupted on every 16th step
```

```
        if ( 0 == ( ++Counter & 0x0F ) && mInterrupted )
        {
            return;
        }
        RepNop();
}
```

Listing 5.5: Check for interrupting state during *HighPerTimer* sleep

At this stage *HighPerTimer* provides an interruption only when process sleeps in busy-wait loop. The task of interruption during system sleeping relates to further work and now it is difficult to predict which pitfalls can be faced. However, it's known that system *sleep()* function may be implemented using *SIGALRM* [41]. Presumably, to terminate sleep it would necessary to send a signal and use some thread synchronization methods, for instance, *mutex* which allows accessing the resource only one thread at a time. Only when a process goes to the signaled state are the other resources allowed to access.

Conclusions

The primary goal of this thesis was the creation of time handling classes on the basis of separately existing *CTSCTimer* and *HPETTimer* classes, which allows combining well-known timer sources to a single unified *HighPerTimer*. With the presented modifications and additions to the original code, it was expected to achieve greater efficiency, expand a scope of applicable systems and enhance resolution up to nanoseconds. As a result, *HighPerTimer* successfully runs on older and newer versions of Intel and AMD processors, including VIA and embeded-plattforms with ARM processors. *HighPerTimer* supports three kinds of timer sources, automatically identifies and chooses the most stable and reliable source and has a capability for user to change default timer to another one. It also corresponded with the last standard of programming language, which provides better readability of source code and incorporates a resilient fault tolerant solution due to handling error. The important achievement is the creation of a new improved *HighPerTimer* sleep with minimum nanoseconds loss and CPU utilization, which creates a big competitive advantage for the new *HighPerTimer*.

HighPerTimer has the potential for a wide applicability, predominantly in a scope of telecommunication technologies. With *HighPerTimer* it is possible to improve an estimation of high-precision network performance, to increase accuracy of measurements of bandwidth capacity, maximum sustained throughput, packet delay and do in-depth analysis of many other derivative network parameters. *HighPerTimer* also has the potential to have real-time multimedia applications in the Internet and be applicable in video mixing.

However, there is also space for developing and improving the created timing classes. One of the next steps is testing the accuracy and support on the virtual machines. For *HighPerTimer*, the possibility to run under conditions of virtuality will be a great advantage. Now it is unknown, which explicit difficulties can be faced and this case should be checked. It is also planned to better support the ARM processor system timer. Since ARM posses neither HPET nor TSC, the only way to support ARM at this stage is to select OS Timer. In the documentation it is said that an ARM implementation must include a system timer, SysTick [37 B3-744] (or it is also called GP Timer the ARM-kernel tree). Presumably, an invocation of the initial ARM system timer can afford to save several additional microseconds and improve the time accuracy.

Bibliography

[1] A. Phanishayee, E. Krevat, V. Vasudevan, D.G. Andersen, G.R. Ganger, G.A. Gibson, S. Seshan. *Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems.* USENIX FAST, February 2008

[2] G. Iannaccone, C. Diot, I. Graham, N. McKeown. *Monitoring very high speed links*. Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement, San Francisco, USA, November 2001

[3] R. Takano, T. Kudoh, Y. Kodama, F. Okazaki. *High-resolution Timer-based Packet Pacing Mechanism on the Linux Operating System*. IEICE Transactions on Communication, November 2011

[4] Performance monitoring with the RDTSC instruction. Issue: June 2011
URL: http://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf

[5] E. Corell, P. Saxholm, D. Veitch. *A user friendly TSC clock*. in Proc. PAM, Adelaide. Australia, March, 2006

[6] F. Proctor, W. Shackleford. *Real-time Operating System Timing Jitter and its Impact on Motor Contro.* Proceedings of SPIE, 2001

[7] S. Siddha, V. Pallipadi, D. Ven. *Getting maximum mileage out of tickless*. in Proc. of the 2007 Linux Symposium, 2007

[8] Intel IA-PC HPET (High Precision Event Timers) Specification, Issue: October 2004
URL: http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf

[9] A. Bakharev, E.Siemens, V.Shuvalov. *Methodology of High-accuracy Measurements of Delay in Modern Computer Systems, (Russian).* 6th Industrial Scientific Conference “Information Society Technologies”. Moscow, 2012

[10] J. Micheel, S. Donnelly and I. Graham, *Precision time stamping of network packets*, Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement, San Francisco, California, USA. November 2001

[11] P.Orosz, T.Skopko. *Performance Evaluation of a High Precision Software-based Timestamping Solution.* International Journal on Advances in Software, 2011

[12] *Enabling Timekeeping Function and Prolonging Battery Life in Low Power Systems*, NXP Semiconductors, 2011
URL: http://www.digikey.com/us/en/techzone/microcontroller/resources/articles/enabling-timekeeping-function.html

[13] Intel 64 and IA-32 Architectures, Software Developer‘s Manual, Issue: March 2012
URL: http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-vol-1-2a-2b-3a-3b-manual.pdf

[14] AMD64 Archutecture Programmer’s Manual, Issue: December 2005
URL: http://support.amd.com/us/Processor_TechDocs/24593_APM_v2.pdf

[15] *Timekeeping in VMware Virtual Machines,* 2010
URL: http://www.vmware.com/files/pdf/Timekeeping-In-VirtualMachines.pdf

[16] Intel SpeedStep Technology for the Intel Pentium M Processor, Issue: March 2012
URL: http://download.intel.com/design/network/papers/30117401.pdf

[17] Intel Processor Identification and the CPUID Instruction, Issue: May 2012
URL: http://www.intel.com/content/www/us/en/processors/processor-identification-cpuid-instruction-note.html

[18] AMD CPUID Specification, Issue: September 2012
URL: http://support.amd.com/us/Embedded_TechDocs/25481.pdf

[19] Wikipedia, *High Precision Event Timer*. Version: 24.07.2012
URL: http://en.wikipedia.org/wiki/High_Precision_Event_Timer

[20] OSDev wiki, Online community of operating system developers. *Programmable Interval Timer*. Version: 26.06.2012
URL: http://wiki.osdev.org/Programmable_Interval_Timer

[21] OSDev wiki, Online community of operating system developers. *8259 PIC.* Version: 04.04.2012
URL: http://wiki.osdev.org/PIC

[22] J. Ala-Paavola. *Software interrupt based real time clock source code project for PIC microcontroller.* August 2007
URL: http://users.tkk.fi/~jalapaav/Electronics/Pic/Clock/index.html

[23] OSDev wiki, Online community of operating system developers. *APIC timer.* Version: 12.05.2012
URL: http://wiki.osdev.org/APIC_timer

[24] H. Hu. *Untersuchung und prototypische Implementierung von Methoden zur hochperformanten Zeitmessung unter Linux,* (German), Bachelor Thesis, Anhalt University of Applied Sciences, Koethen. November 2011

[25] Y. Artyukh, V. Bespal'ko, E. Boole. *Nonlinearity errors of high-precision event timing.* Automatic Control and Computer Sciences Conference. Latvia, 2008

[26] D. Kachan, E.Siemens, H.Hu. *Tools for the high-accuracy time measurement in computer systems*, (Russian). 6th Industrial Scientific Conference “Information Society Technologies”. Moscow, 2012

[27] Online documentation. *open(2) - Linux Manual Page*. 2011
http://linux.die.net/man/2/open

[28] Online documentation. *mmap(2) - Linux Manual Page*. 2011
http://linux.die.net/man/2/mmap

[29] R. Love. *Linux System Programming.* O'Reilly Media. First edition, 2007

[30] J. Dike, *A user-mode port of the Linux kernel*. USENIX Association Berkeley, CA, USA, 2000

[31] K. Jain, R. Sekar. *User-Level Infrastructure for System Call Interposition: A Platform for Intrustion Detection and Confinement.* In Proceedings of the ISOC Symposium on Network and Distributed System Security, February 2000

[32] T. Garfinkel. *Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In* Proceedings of the ISOC Symposium on Network and Distributed System Security*, 2003*

[33] J. Corbet. *On vsyscalls and the vDSO. Kernel development news.* Linux news site LWN. June, 2011
URL: http://lwn.net/Articles/446125/

[34] Online documentation*. clock_getres(3) - Linux Manual Page*. 2011
http://linux.die.net/man/3/clock_getres

[35] S. Meyers. *Effective C++. 55 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley Professional. Third edition, 2011

[36] M. Hachman. *ARM Cores Climb Into 3G Territory.* Technology weblog ExtremeTech. 2002.
URL: http://www.extremetech.com/extreme/52180-arm-cores-climb-into-3g-territory

[37] ARM v7-M Architecture Reference Manual, Issue: November 2010
URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0395b/CIHCAGHH.html

[38] Dr. Durgesh Pant, Mahesh Kr. Sharma, K.S. Vaisla. *Object-Oriented Systems in C++.* Firewall Media. First edition, 2007

[39] Wikipedia. *C++11*. Version: 12.08.2012
URL: http://en.wikipedia.org/wiki/C%2B%2B11

[40] S. Meyers. *Overview of the New C++ (C++11).* Fifth edition, 2012

[41] Online documentation. *sleep(3) - Linux Manual Page*. 2011
http://linux.die.net/man/3/sleep

[42] Online documentation. *usleep(3) - Linux Manual Page*. 2011
http://linux.die.net/man/3/usleep

[43] Online documentation. *nanosleep(2) - Linux Manual Page*. 2011
http://linux.die.net/man/2/nanosleep

[44] M. Kravetz, H. Franke, S. Nagar, R. Ravindran, *Enhancing Linux Scheduler Scalability.* Proceedings of the Ottawa Linux Symposium, Ottawa, Canada, July 2001

[45] Steven A. Hofmeyr, Stephanie Forrest, Anil Somayaji. *Intrusion Detection Using Sequences of System Calls*. Journal of Computer Security, 1998

[46] D. Bovet, M. Cesati. *Understanding Linux Kernel.* O'Reilly. Second edition, 2003

[47] Online documentation. *getrusage (2) - Linux Manual Page*. 2011
http://linux.die.net/man/2/getrusage