УДК 004.42

Self-configurable time source initialization for obtaining high-precision user-space timing

I. Fedotova, E. Siemens

In this paper, the algorithms and identification logics of a high performance user-space time source are described along with an experimental comparison of the accuracy and CPU costs of the available time sources under the Linux OS. This mechanism is presented within the unified user-space library *HighPerTimer* which allows identification of the most appropriate and accurate time source on PC-platforms as well as on ARM-architectures and obtaining a high-accurate down to nanosecond precision of time acquisition. This solution substitutes the accustomed way of getting time value through Linux system calls and provides much faster way for time acquisition.

Key words: timestamp precision, hardware timestamping, timekeeping, HPET, TSC, high-performance computing.

1. Introduction

Efficient and precise tracking and measurement of time intervals is a key feature of implementation of state machines and communication protocols in modern computers and embedded systems [1, 2]. Furthermore, time measurements in high-speed performance networks and the correct performance of proper algorithm implementation in telecommunication protocols require and depend on high performance and accurate timestamping. Regarding this point, the way of time acquisition through standard C library interface doesn't provide appropriate reliability, because system calls are invoked and some extra time is spent, which on some architecture can be up to several microseconds [3, 4]. However, interaction with time hardware directly can help to avoid wrapping system calls and therefore increase precision of timekeeping.

Though our investigations have been performed under the Linux OS, most of the issues are common to other wide-spread operating systems like Linux, BSD-derivates and MacOS. Firstly, depending on the hardware-environment and boot parameterization, the Linux kernel can use different hardware time sources. The most advanced are the Time Stamp Counter (TSC) [5, 6] and the High Precision Event Timer (HPET) [7, 8]. Their main characteristics such as reliability, stability and clock resolution heavily depend on the processor architecture. So the TSC can have non-monotonic characteristics, means that in some circumstances 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 time interval measurements. Moreover, there are some cases [9, 10] in which 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 with very low CPU costs of the calculations with these timers.

In this paper, a novel approach solving the initialization problems of high-efficient and precise timer is elaborated. The primary goal is the instantiation of time handling classes, which allow combining well-known timer sources to a single unified class interface, which automatically takes

upon itself the choice of most suitable source (TSC, HPET or another one) like Linux kernel does it at boot time. This work then seeks to identify the most reliable hardware and provides to the user fast access to it through the *HighPerTimer* library, performing accurate time interval measurement. In the context of this library, the most reliable time source is a source, which supplies the accurate time information and posses not only the fastest way for time acquisition, but the most secure and trustworthy on the particular processor. It means, that user should not investigate in advance which timer hardware is more suitable for his purposes, due to the fact that an assignment of the appropriate source to *HighPerTimer* source occurs at initialization stage in user space. Therefore, by targeting the nanosecond precision, this mechanism is focusing on the possibility of a universal, purely library-based solution for platforms running Linux OS. One more goal is also developing an advanced software design, making it using features of the new C++11 standard and achieving higher efficiency and code maintainability.

2. Time source definition

In general, the Linux kernel usually explicitly interacts with the Time Stamp Counter (TSC) and the High Precision Event Timer (HPET). Since the clock is the fundamental unit of time as seen by the processor, the TSC posses the fastest and the lowest possible overhead way of getting CPU time and provides the highest-resolution timing information available for that processor. It is set to 0 following a reset of the processor and since that the counter increments every clock cycle of a particular CPU. In modern SMP- and NUMA-systems each processor has its own TSC counter. The time value of TSC is derived from FSB buses cycles and during processor performance, TSC' values can be out of alignment. 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 [11, p. 4].

In newer processors the TSC may support an enhancement, referred to as an *Invariant TSC* feature, which is not tightly bound to processor cores and their cycles and, according to Intel documentation, has a constant rate [12, vol. 3B, 17-50]. For most cases, the presence of this flag is essential for accepting it as *HighPerTimer* source. However, not all processors families increment the Time-Stamp Counter in similar way. There are also some specific processors, which support the use of the TSC as a preferable timer even if the processor core changes its frequency. So *HighPerTimer* library investigates a particular model and family of given processor to identify the way of TSC implementation. At this point, our approach is – to check the stability of TSC and in case of an instable TSC just to switch to another more reliable clock source.

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 main motivation for its creation was the necessity to replace slow and older Programmable Interval Timer (PIT) [10] whose frequency (1.19 MHz) did not meet the current system and software requirements. The HPET circuit in modern PCs is integrated into the south bridge chip and consists of a 64-bit or 32-bit main counter register counting at a frequency of at least 10 MHz and a set of timers that can be used by the operating system. The problem using this timer in user-space is that operating systems, designed before HPET has been introduced, can't access it, so they work only on hardware that has other timer facilities.

Moreover, if the HPET main counter register has frequency of 10 MHz in a 32-bit mode, an overflow arises every 7.16 minutes [4 p. 10]. However, 10 MHz is explicitly the minimum required frequency of a HPET-compatible timer – there are well known chipsets with a HPET main counter frequency of 25 MHz, at which the register overflow in 32 bit mode within less than 3 minutes. Consequently, it is very dangerous it in 32-bit mode in the scope of *HighPerTimer*. So in this library the use of HPET timer with a 32-bit main counter is generally avoided.

Furthermore, there is always one more alternative clock source available - to rely on the operating system choice, which means to use some operating system functions for obtaining the time value. For instance, *clock_gettime()* system call, which is assumed as the OS timer in the context of *HighPerTimer* library. The issue and pitfall here is that, 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 [14, 15]. 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 lists, etc. So this way of relying on the OS time source has the lowest priority and is used only when other options are not available.

3. *HighPerTimer* library structure

The high priority *HighPerTimer* task is identifying the reliable time source among available on this processor and assigning of it to the high performance time source at the early stage of initialization. It should be emphasized that setting source for *HighPerTimer* occurs in user space, not in kernel space. So in most cases user doesn't need to know, with which timer hardware he works actually, he just need to rely on nanosecond precision timestamping through interacting with some kind of a unified timer software library (so called *High Performance Timer – HighPerTimer*).

Thereby, the *HighPerTimer* library supports the most usable and preferable time counters: TSC, HPET and alternative timer of operating system – OS Timer. For each of them corresponding classes *TSCTimer*, *HPETTimer* and *OSTimer* have been declared. On the Figure 1 the class diagram is showing a brief structure of the relationship between classes within *HighPerTimer* library. *TSCTimer*, *HPETTimer* and *OSTimer* classes have only private members and methods and their implementation is completely hidden from user. Through an assembly code, embedded into C++ methods, they provide a direct access to the timer hardware, initialize timer source, retrieve their time value and are at only *HighPerTimer* class's disposal. These classes have «friend» relationship with *HighPerTimer* class, which means that it places their private and protected methods and members at friend classes' disposal. For safety and security reasons, we protect the hardware access from use by application users and permit it only from special classes. A limited read-only access to some specific information regarding CPU and time hardware features, which is obtaining in a protected interface, is provided by *AccessTimeHardware* class from *HighPerTimer* file.

The principal mechanism for adaptation main characteristics of a unified timer, as well as the whole routine of handling time values, which is assumed for user interaction, is declared within class *HighPerTimer*. *HighPerTimer* class keeps a set of constructors which set an appropriate time value in seconds, nanosecond, in tics or in `timespec` or `timeval` structures as defined by all Unixes [16]. It has capabilities to retrieve the current time value in the sequel to perform appropriate operations with this value or measure the performance of some operations:

```
// declare HighPerTimer objects
HighPerTimer timer1, timer2;
HighPerTimer::Now (timer1);
// measuring operation
HighPerTimer::Now (timer2);
```

Further, timer object operations like comparison, arithmetical operations and assignments are also provided by the *HighPerTimer*, which provide a fast and effective way to handle time objects, since these operations are effectively deducted to respective operations on the signed 64-bit timer main counter which is counting the number of tics of the respective hardware clock frequency from the beginning of the Unix epoch., Seconds, microseconds and nanoseconds can be extracted from a timer object in a lazy initialization way, that means that calculation between the number of tics and seconds/nanoseconds is done only on demand, when these values are requested. However, seconds and nanoseconds are used rather rarely and mostly in a less time-critical part, so these recalculations don't hurt the high-performance time calculations. Also accessors to `timespec` or `timeval` structs are provided for bridging between the Unix timing-ecosystem and the *HighPerTimer* objects.

Inside the implementation of the default constructor of *HPTimerInitAndClean* class, the strict order of initialization process for *HighPerTimer* class is defined. It is accomplished by invocation of corresponding *HighPerTimer* initialization methods in a strict sequence. Thereby, a user interacts only with the high performance timer from *HighPerTimer* class and has an access through *AccessTimeHardware* class to some advanced hardware information, whereas interfaces of other classes are protected and hidden from him.

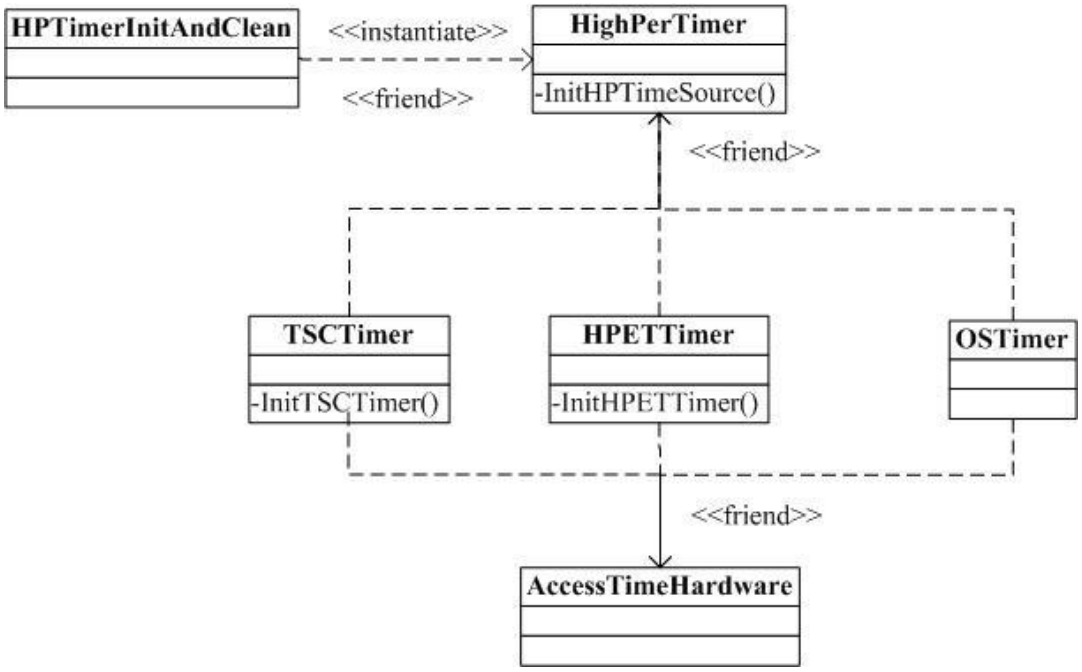

Figure 1. Simplified class diagram of *HighPerTimer* library

The *HighPerTimer* library is only partly thread-save. All generic operations on the timer object are by intention not-thread safe and it is assumed that interaction with *HighPerTimer* objects is performed from the one single thread. Otherwise, a user has to take care on protection of a competitive access by its own facilities. The rationale behind this is that thread-safety can't come without performance penalties, however high-resolution and high-performance timer objects are often used at very low levels of program logics, in which concurrent thread access to a particular timer object can be excluded from the program logics. However, some special capabilities of *HighPerTimer* library such as sleeping methods and mechanisms for resuming sleeping thread by an *Interrupt()* method are thread-save, since *Interrupt()* is by definition designed to be called from a thread apart from the sleeping object thread.

4. Change of the Time Source for *HighPerTimer*

The initializing routine of *HighPerTimer* implies an implementation of some central parameterization of the library, which has to be performed at the initialization time of the library. Primarily, it is the time source selection of the *HighPerTimer*, which is accomplished on the bases of the appropriate methods call from *TimeHardware* class. Especially, *InitHPTimeSource()* calls *InitTSCTimer()* and *InitHPETTimer()* methods, which attempt to initialize time hardware (see Fig. 1) Additionally, this concept also includes calculation of timer frequency, the value of shift against Unix Epoch, maximum and minimum values for *HighPerTimer* and determining HZ frequency of the kernel, where the value of HZ is defined as the system timer interrupt rate and varies across kernel versions and hardware platforms. These values are necessary for further operations with time values. 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.

Nevertheless, sometimes there are some cases, when user would prefer some particular time 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 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.

5. Identification of unified high performance time source

The *TSCTimer* and *HPETTimer* classes contain respective initialization routines and appropriate *InitTSCTimer()* and *InitHPETTimer()* methods, which return true for success and false for failure. Success is meant the verifiable time source is stable and it can be used for accurate counting, the failure means that it should not be relied on this time source. As it was mentioned above, the TSC is considered as a more preferred timer, so it is always checked first.

For example, processor VIA Nano X2 has a constant TSC rate, its initialization routine returns true and, for this case, the check of HPET device is not necessary and does not occur. Though, HPET and OS Timer can also be used by *HighPerTimer* special capability to change source by the library user after automatic initialization is finished. In Figure 2, the behavior of all available time sources is shown in detail. The tests focus on measurements of the cost of setting timer and were performed in a loop consisting of 100 million steps. The costs of getting timer tics mean here the time setting timer and extracting its time value is required.

The main features of given processor are:

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

However, during the experiments it was observed some rarely occurring peak values, which can obviously destroy mean and especially standard deviation values. This behavior can be traced with

all time sources, among all tested processors. They are noticeable from the Figure 2 and the Figure 3 and presumably, can be caused by an interruption of the time measurement loop (like by Interrupt Service Routines of by the process scheduler of the kernel). According to (1), which is taken for calculation standard deviation value *s*, on the initial range, sum of x^2 (sum of squares) and the mean value can be greatly increased under the influence of only one peak.

$$s = \sqrt{\frac{1}{n-1}\left[\left(\sum_{i=1}^{n} x_i^2\right) - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2\right]}. \quad (1)$$

So, in the course of given research, for calculating mean and standard deviation value (see Tab. 1, Tab. 2) filtering of range is necessary, as these phenomena of peaks are out of scope of our investigations. The point here is that with unfiltered range it is obtained not inaccurate results, but meaningless, since we are here interested in measurement of time distance between two consecutive time fetches only. So filtering out of peaks allows us to prevent calculation of the bias, caused by process interruptions and hence helps to obtain more physically meaningful results.

Table 1. Mean and standard deviation values of HPET, TSC and OS Timer costs on the VIA Nano X2 processor

Timer source	Mean, nsec	Standard deviation, nsec
TSC Timer	38.231	0.3134
HPET Timer	598.72	76.015
OS Timer	102.20	0.5253

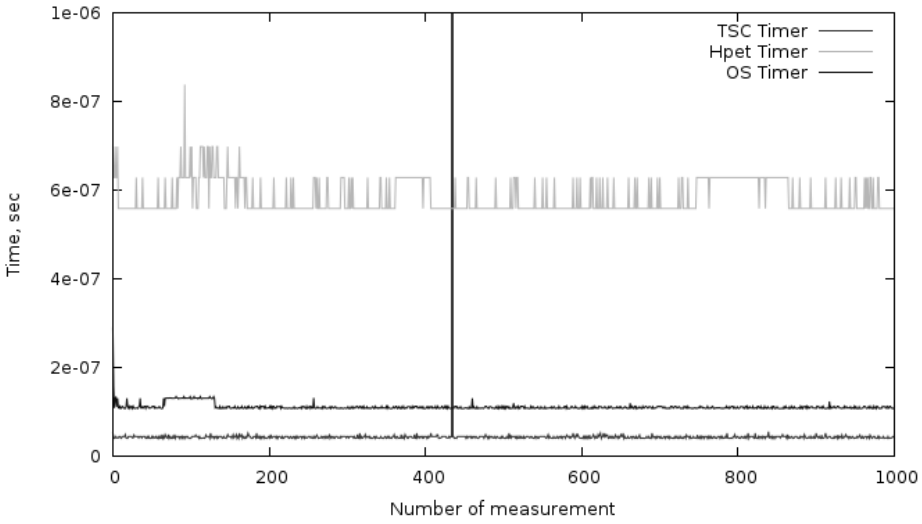

Figure 2. Measurements of TSC, HPET and OS Timer costs on the VIA Nano X2 processor

If the TSC initialization function returns false, it means that the TSC is unstable and can't be used as a time source. In that case there are two more options for the clock source – HPET or OS Timers. However, there is no guarantee which hardware timer is used by *clock_gettime()* call, issued by the OS Timer and, regarding to this point, it is necessary to check the mean value of getting ticks cost of both timers. If mean values are similar (the difference is no more than 25*%)*, there is a sense to estimate timers by standard deviation values.

The first point here is that though the TSC mean value can be lower than the HPET mean value, but Linux kernel, as well as our *HighPerTimer* library, indicates the TSC as a non-stable, unreliable timer and actually its frequency can change periodically. Regarding to the next point, the OS Timer is implemented through *clock_gettime()* function, which means that it always invokes a system call

to obtain time value from the time hardware. For the case, when the HPET is a current Linux time source, the OS Timer, in fact, interacts with the HPET, but spends some extra time because of wrapping system calls. Being more precise, according to the Table 2, this extra timer or, in other words, the difference of mean values of the HPET and OS Timer, is about 0.0541 usec and the similarity of their behaviors is noticeable from the Figure 3 against to the situation is described above (see Fig. 2) when the OS Timer uses the TSC. However, to provide appropriate level of reliability, we also evaluate numbers by some threshold for deviation difference and in fact, can give a precedence to the OS Timer, which has higher mean value, but lower deviation, so is more reliable.

As an example with unstable TSC source, a processor AMD Athlon X2 Dual Core can be examined. Its main features are:

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

Given processor has unstable TSC rate, so *InitTSCTimer()* returns false. HPET device with frequency of 25 MHz is accessible. According to Figure 3, OS Timer and, being more precisely, *clock_gettime()* based on the HPET timer. Their lines are very close to each other and have similar behavior. More precisely, mean values of this both timers differ by about 2% (Table 2). In this case it is necessary to compare standard deviation of their values, all the more so both lines are vary widely. Uniquely, this figure is established and explained the propriety of the additional deviation check.

Table 2. Mean and standard deviation values of HPET, TSC and OS Timer on the AMD Athlon processor

Timer source	Mean, usec	Standard deviation, usec
TSC Timer	0.0251	0.0015
HPET Timer	1.0633	0.2079
OS Timer	1.1174	0.3743

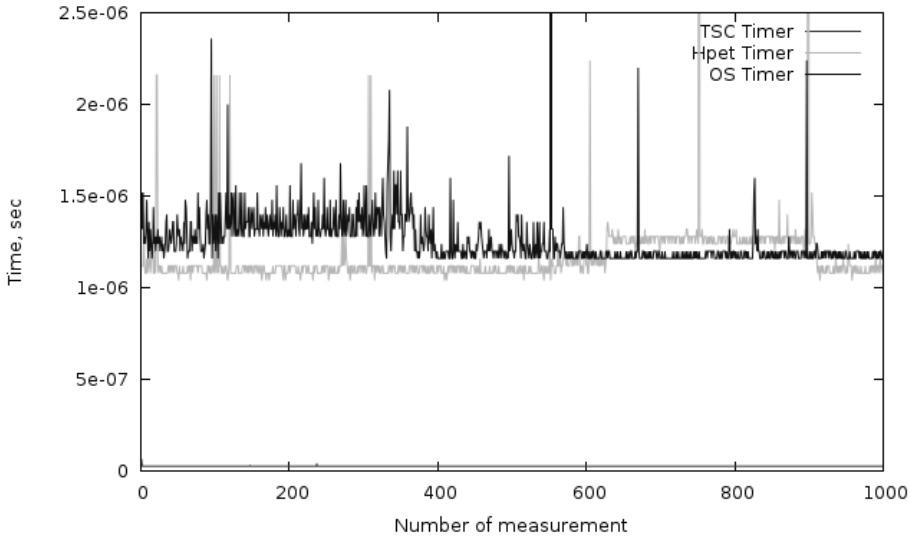

Figure 3. Measurements of TSC, HPET and OS Timer on the AMD Athlon processor

6. Conclusion

According to the requirements of advanced high speed data networks, the timestamp precision of network measurement applications must be increased against state-of-the-art methods. In this paper, we have succeeded in finding a solution, which is autoconfiguring itself on many systems, and providing access to the most reliable timer, which are much faster than standard system calls. *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.

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 develop better supporting the ARM processor system timer. Since ARM posses neither HPET nor TSC, the only way to run on the 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 [17, B3-744] (or it is also called GP Timer in 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. *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.
2. *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.
3. *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.
4. *H. Hu*. Untersuchung und prototypische Implementierung von Methoden zur hochperformanten Zeitmessung unter Linux, (German), Bachelor Thesis, Anhalt University of Applied Sciences, Koethen. November 2011.
3. Performance monitoring with the RDTSC instruction. URL: `http://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf`. Issue: June 2011.
6. *E. Corell, P. Saxholm, D. Veitch*. A user friendly TSC clock // in Proc. of PAM, Adelaide. Australia, March, 2006.
7. *S. Siddha, V. Pallipadi, D. Ven*. Getting maximum mileage out of tickles // in Proc. of the 2007 Linux Symposium, 2007.
8. Intel IA-PC HPET (High Precision Event Timers) Specification. URL: `http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf`. Issue: October 2004.
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. Enabling Timekeeping Function and Prolonging Battery Life in Low Power Systems, NXP Semiconductors. URL: `http://www.digikey.com/us/en/techzone/microcontroller/resources/articles/enabling-timekeeping-function.html`. 2011.
11. Intel SpeedStep Technology for the Intel Pentium M Processor, URL: `http://download.intel.com/design/network/papers/30117401.pdf`. Issue: March 2012

12. Intel 64 and IA-32 Architectures, Software Developer‘s Manual. URL: http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-vol-1-2a-2b-3a-3b-manual.pdf. Issue: March 2012.
13. *J. Ala-Paavola*. Software interrupt based real time clock source code project for PIC microcontroller. URL: http://users.tkk.fi/~jalapaav/Electronics/Pic/clock/index.html. August 2007.
14. *J. Dike*, A user-mode port of the Linux kernel // USENIX Association Berkeley, California, USA, 2000.
15. *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.
16. GNU Operating System Manual, «Elapsed Time». URL: http://www.gnu.org/software/libc/manual/html_node/Elapsed-Time.html. Issue: November, 2012.
17. ARM v7-M Architecture Reference Manual, URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0395b/CIHCAGHH.html. Issue: November 2010.

Paper was received for editing 29.10.2012;
Reprocessed version — 20.11.2012

Fedotova Irina
Master student of MG 11 group, SibSUTI, Siberian State University of Telecommunications and Informatics, e-mail: fis.irina@gmail.com.

Siemens Eduard
Communications Technology Chair, Anhalt University of Applied Sciences (Bernburger Str. 57, 06366 Koethen, Germany) Tel.: +49 3496 67 2327, e-mail: e.siemens@emw.hs-anhalt.de.

Автоматическая инициализация источника времени для получения высокоточных временны́х измерений в пространстве пользователя

И. С. Федотова, Э. Сименс

В статье описываются алгоритмы и логики идентификации высокопроизводительного источника времени в пространстве пользователя, наряду с экспериментальным сравнением точности и стоимости выполнения на ЦПУ доступных временны́х источников ОС Linux. Данный механизм представлен в рамках унифицированной библиотеки *HighPerTimer*, которая позволяет идентифицировать наиболее подходящий и наиболее точный источник как на ПК-платформе, так и на архитектуре ARM, и получать высокоточные временные интервалы с разрешением до наносекунд. Данное решение заменяет хорошо известный способ получения значения времени средствами системных вызовов ОС Linux и обеспечивает намного более быстрый подход для сбора временны́х меток.

Ключевые слова: точность временны́х меток, аппаратные временны́е метки, регистрация времени, HPET, TSC, высокопроизводительные вычисления.