

# High-Resolution Timers

## A Prime Feature of Real-Time Applications

An overview of how to use high-resolution timers in real-time Linux systems.



Operating systems are resource managers. Their job is to interact with the underlying hardware and unleash its services on behalf of applications or users. In general, we are only concerned with the correct output and ignore negligible delays in processing time. However, for some tasks, we need to get an exact result within a set period of time. If the task is not completed within that time frame, it might create undesirable results.

Wikipedia summarises the concept of

real-time systems as: “A system is said to be real-time if the correctness of an operation depends not only upon the logical correctness of the operation but also upon the time at which it is performed. The classical conception is that in a hard or immediate real-time system, the completion of an operation after its deadline is considered useless—ultimately, this may lead to a critical failure of the complete system. A soft real-time system, on the other hand, will tolerate such lateness, and may respond with decreased service quality (e.g., dropping frames while displaying a video).” Therefore, for real-time tasks, our operating system should be able to provide the output within the set deadline.

Operating systems can be broadly classified into two categories. You can either go for real-time operating systems (RTOS) to run real-time applications like RTLinux, VxWorks, Nucleus etc, or you can use general-purpose operating systems (GPOS), which are usually run on desktops and servers.

Even though Linux is one of the GPOS, it still supports soft real-time tasks. So when we write a real-time program, we have to avail the real-time features of Linux—such as real-time priority, real-time scheduling policy, high-resolution timers, etc. Likewise, for any kind of real-time application, a timer is one of the most important features, as its purpose is to notify an event’s completion or, at least, account for a specified time.

High-resolution timers are proposed by many universities/groups, but it is always

advisable to follow the POSIX standards. POSIX high-resolution timers were standardised by IEEE 1003.1b in the early 1990s. They can be created in any process and used for various purposes. The following list shows the available APIs related to high-resolution clocks and POSIX high-resolution timers.

- `clock_settime`—to set time for a particular HRT clock
- `clock_gettime`—to know the HRT clock's remaining time
- `clock_getres`—to know the resolution/precision of the HRT clock
- `timer_create`—used to create a 'per process' high resolution timer
- `timer_settime`—to set time in seconds and nanoseconds
- `timer_gettime`—to know the remaining time
- `timer_getoverrun`—to know the count of timer overruns.
- `timer_delete`—to delete the created timer in the calling process.

In this article we will restrict ourselves to the `timer_create`, `timer_settime`, `timer_gettime` and `timer_delete` functions.

```
timer_create ( ) and timer_delete ( )
```

The syntax of the `timer_create ( )` and `timer_delete ( )` are as follows:

```
int timer_create(clockid_t clockid, struct sigevent *restrict
evp, timer_t *restrict timerid);
int timer_delete(timer_t timerid);
```

`timer_create` accepts three arguments—clock type (`clockid`), `sigevent` structure that holds the signal related information and a variable of type `timerid`. The above syntax explains the usage of the `timer_create` function.

Now, considering the first argument, 'clock type' indicates which clock to use. If the timer has to count according to a wall clock, the clock type should be `CLOCK_REALTIME`. This particular clock type is not system dependent. The second type of clock can be `CLOCK_MONOTONIC`. We can use it only if the system supports monotonic clocks. The third type of clock is `CLOCK_PROCESS_CPUTIME_ID`. This timer performs increment/decrement operations only when the CPU itself executes the calling process. It is system dependent. The fourth clock type is `CLOCK_THREAD_CPUTIME_ID`. Here, the timer performs increment/decrement operations only when the calling thread is with the CPU for execution. This, too, is system dependent.

The second argument in the syntax must be a variable of type `struct sigevent` (in the example program variable 's'). The `sigevent` structure contains many members, but only `sigev_notify` and `sigev_signo` is of interest to us. `sigev_notify` is used for specifying how the timer

expiration should be notified to the calling process — either by issuing a signal, creating a thread or indicating if any other notification method needs to be followed. However, Linux only supports the first two methods. The `sigev_signo` member should be assigned with any signal name or signal number. An advantage of using these timers is that any signal can be configured. Thus, when the timer expires, the signal will be delivered to the calling process.

The third argument is a variable for identifying the timer that has been created. So we need to declare a variable of type `timer_t` (in the example program, variable 'tid'). As these timers are per-process timers, we can delete the created timer using the `timer_delete ( )` function by passing `timer id` as the argument.

```
// Program 1 : Illustration of timer_create API
#include<stdio.h>
#include<time.h>
#include<signal.h>
int main()
{
    timer_t tid;
    struct sigevent s;
    s.sigev_signo = SIGUSR1;
    s.sigev_notify = SIGEV_SIGNAL;
    if(timer_create(CLOCK_REALTIME, &s, &tid) ==0)
        printf("\ntimer_create is success\n");
    timer_delete(tid);
    return 0;
}
```

## timer\_settime ( )

The syntax of the `timer_settime ( )` is as follows:

```
int timer_settime(timer_t timerid, int flags, const struct
itimerspec *restrict value, struct itimerspec *restrict ovalue);
```

As you can see, the function accepts four arguments. The first one is `timerid`. The second one is a flag value that can be either 0 or `TIMER_ABSTIME`. If the flag value is 0, then the timer expires according to the `it_value`, a member for the `struct itimerspec` (which we will explain shortly). However, if the flag value is `TIMER_ABSTIME`, then the timer will expire within an amount of time that is the difference between the absolute time specified by the `it_value` member and the current value of the clock associated with the timer.

The third argument of the function is a structure of type `struct itimerspec`. The declaration of this structure is shown below:

```
struct itimerspec {
    struct timespec it_interval;
    struct timespec it_value;
};
```

The declaration of *struct timespec* is as follows:

```
struct timespec {
    __time_t    tv_sec;
    long int    tv_nsec;
};
```

As you can see, before we can call the *timer\_settime* API, we need to assign the values for *tv\_sec* and *tv\_nsec*. *it\_value* holds the value of time that the timer counts. The *it\_interval* member holds the next re-loadable value for *it\_value*, which is the next interval for the timer. If *it\_interval* is assigned values, then the timer will act as a periodic timer; otherwise it will act as a one-shot timer.

The final argument is again a variable of type *struct itimerspec*, but this variable can hold the old value of the timer.

In Program 2, the configured timer is acting as a one-shot timer. Programs 2 and 3 use a signal API, which is used to register an incoming signal. (For simplicity, we have considered the signal API instead of a sigaction API).

```
//Program 2 : Illustration of timer_settime API
#include<stdio.h>
#include<time.h>
#include<signal.h>
void sig_fn()
{
    printf("SIGUSR1 handled successfully\n");
}
int main()
{
    timer_t tid;
    struct sigevent s;
    struct itimerspec it;
    struct timespec req;
    s.sigev_signo=SIGUSR1;
    s.sigev_notify=SIGEV_SIGNAL;
    it.it_value.tv_sec=5;
    it.it_value.tv_nsec=1000;
    req.tv_sec=8;
    req.tv_nsec=1000000;
    signal(SIGUSR1,sig_fn);
    timer_create(CLOCK_REALTIME,&s,&tid);
    if(timer_settime(tid,0,&it,0) ==0)
        printf("timer_settime is success\n");
    nanosleep(&req,0);
    timer_delete(tid);
    return 0; }
```

## timer\_gettime ( )

The syntax of *timer\_gettime* ( ) is as follows:

```
int timer_gettime(timer_t timerid, struct itimerspec *value);
```

The *timer\_gettime* ( ) function is required whenever we need to fetch information about the remaining time in an existing timer. *timer\_gettime* accepts two arguments. The first argument is the *timerid* and the second argument is a variable of type *struct itimerspec* that should be empty as it is acting as an in parameter. The *it\_value* member of the second argument holds the remaining time before the timer expires and the *it\_interval* member contains the reloaded value that gets set by the previous *timer\_settime* function.

Programs 2 and 3 also use another function called *nanosleep*, which is used to delay the program execution. The syntax of *nanosleep* is as follows:

```
int nanosleep(const struct timespec *req, struct timespec *rem);
```

One of its primary advantages is that it lets one set the time in seconds, or even nanoseconds. *nanosleep* accepts two arguments, where both are of type *struct timespec*. The first argument contains the actual seconds and nanoseconds by which the process execution has to be delayed. The second argument holds the remaining time. This comes in handy if the process execution gets stopped due to a signal or an interrupt, as the function will hold the remaining time set with the first argument. The remaining time can be utilised by calling the *nanosleep* function again.

```
//Program 3 : Illustration of timer_gettime API
#include<stdio.h>
#include<time.h>
#include<signal.h>
void sig_fn()
{
    printf("SIGUSR1 handled successfully\n");
}
int main()
{
    timer_t tid;
    struct sigevent s;
    struct itimerspec it,it_get;
    struct timespec req;
    s.sigev_signo=SIGUSR1;
    s.sigev_notify=SIGEV_SIGNAL;
    it.it_value.tv_sec=10;
    it.it_value.tv_nsec=1000;
    req.tv_sec=5;
    req.tv_nsec=1000000;
    signal(SIGUSR1,sig_fn);
    timer_create(CLOCK_REALTIME,&s,&tid);
    timer_settime(tid,0,&it,0);
    nanosleep(&req,0);
    if(timer_gettime(tid,&it_get)==0)
        printf("Remaining secs=%d\t\tnsecs=%d\n",it_get.it_value.tv_sec,it_get.it_value.tv_nsec);
```

```
timer_delete(tid);  
return 0; }
```

Finally, there is an important point about the compilation process: one needs to use the “-lrt” option along with other required options. For example:


```
# gcc -lrt <filename>
```

Now, with the definitions of some of the high-resolution timer functions out of our way, it's time to look at some of the products that utilise these timers.

1. In the SUSE Linux Enterprise Real Time operating system, high-resolution timers are used to support enhanced scheduling and it is possible to obtain the resolutions at the nanosecond-level.
2. TimeSys Linux operating system from TimeSys Corporation uses high-resolution timers, which significantly improves the performance of critical control applications.
3. NaradaBrokering, a distributed messaging infrastructure, uses high-resolution timers in their substrate to ensure that messages are time-stamped as

accurately as possible.

4. KURT is Kansas University's real-time Linux project in the Information and Telecommunications Technology Centre. The current KURT release uses high-resolution timers to improve its performance.
5. High-resolution timers are implemented in Linux kernel version 2.6 onwards, to schedule any task periodically at accurate time intervals.
6. pSOSystem 3 uses high-resolution timers to manage many asynchronous events.
7. RT Linux has also implemented high-resolution timers because of the features offered by the POSIX timers.

We hope we were able to give you an idea about high-resolution timers and their usage through timer-related functions. These timers can be used in an appropriate real-time application in a very similar way.  **END**

**By: Parimala S. and Dr B. Thangaraju.** The authors are working with Talent Transformation, Wipro Technologies, Bangalore. They can be reached at [parimala.sathyapramodha@wipro.com](mailto:parimala.sathyapramodha@wipro.com) and [balat.raju@wipro.com](mailto:balat.raju@wipro.com)