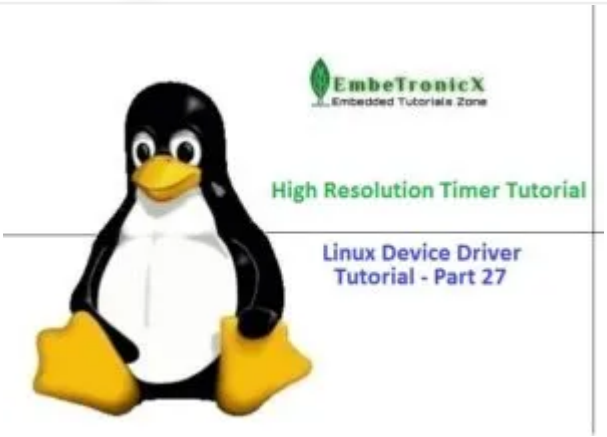
EmbeTronicX
Embedded Tutorials Zone

Sidebar

Home → Tutorials → Linux → Device Drivers → **Linux Device Driver Tutorial Part 27 – Using High Resolution Timer In Linux Device Driver**

Device Drivers



Linux Device Driver Tutorial Part 27 – Using High Resolution Timer In Linux Device Driver

This is the Series on Linux Device Driver. The aim of this series is to provide easy and practical examples that anyone can understand. This is the Linux Device Driver Tutorial Part 27 – Using High Resolution Timer In Linux Device Driver.

Post Contents [hide]

High Resolution Timer (HRT/hrtimer)

In our last tutorial we have seen kernel timer. Now we are talking about high resolution timer. Everyone might have some questions. Why the hell we need two timers? Why can they merge two timers into one? Can't able to integrate? Yes. They have tried to merge these two timers. But they have failed. Because **Cascading Timer Wheel (CTW)** is used in kernel timer. Cascading Timer Wheel (CTW) code is fundamentally not suitable for such an approach like merging these two timers. Because hrtimer is maintaining a time-ordered data structure of timers (timers are inserted in time order to minimize processing at activation time). The data structure used is a red-black tree, which is ideal for performance-focused applications (and happens to be available generically as a library within the kernel).

Kernel Timers are bound to **jiffies**. But this High Resolution Timer (HRT) is bound with 64-bit **nanoseconds** resolution.

With kernel version 2.6.21 onwards, high resolution timers (HRT) are available under Linux. For this, the kernel has to be compiled with the configuration parameter CONFIG_HIGH_RES_TIMERS enabled.

There are many ways to check whether high resolution timers are available,

- In the `/boot` directory, check the kernel config file. It should have a line like `CONFIG_HIGH_RES_TIMERS=y`.
- Check the contents of `/proc/timer_list`. For example, the `.resolution` entry showing 1 nanosecond and event_handler as hrtimer_interrupt in /proc/timer_list indicate that high resolution timers are available.
- Get the clock resolution using the `clock_getres` system call.

Users of High Resolution Timer

- The primary users of precision timers are user-space applications that utilize nanosleep, posix-timers and Interval Timer (itimer) interfaces.
- In-kernel users like drivers and subsystems which require precise timed events (e.g. multimedia).

High Resolution timer API

We need to include the **<linux/hrtimer.h>** (**#include <linux/hrtimer.h>**) in order to use kernel timers. Kernel timers are described by the **hrtimer** structure, defined in **<linux/hrtimer.h>**:

```
1 struct hrtimer {
2   struct rb_node node;
3   ktime_t expires;
4   int (* function) (struct hrtimer *);
5   struct hrtimer_base * base;
6 };
```

Where,

node – red black tree node for time ordered insertion

expires – the absolute expiry time in the hrtimers internal representation. The time is related to the clock on which the timer is based.

function – timer expiry callback function. This function has an integer return value, which should be either HRTIMER_NORESTART (for a one-shot timer which should not be started again) or HRTIMER_RESTART for a recurring timer. In the restart case, the callback must set a new expiration time before returning.

base – pointer to the timer base (per cpu and per clock)

The **hrtimer** structure must be initialized by init_hrtimer_#CLOCKTYPE.

ktime_set

There is a new type, **ktime_t**, which is used to store a time value in nanoseconds. On 64-bit systems, a **ktime_t** is really just a 64-bit integer value in nanoseconds. On 32-bit machines, however, it is a two-field structure: one 32-bit value holds the number of seconds, and the other

holds nanoseconds. The below function used to get the `ktime_t` from seconds and nanoseconds.

```
ktime_set(long secs, long nanosecs);
```

Arguments:

`secs` – seconds to set

`nsecs` – nanoseconds to set

Return:

The `ktime_t` representation of the value.

Initialize High Resolution Timer

hrtimer_init

```
void hrtimer_init( struct hrtimer *timer, clockid_t clock_id,
                   enum hrtimer_mode mode );
```

Arguments:

`timer` – the timer to be initialized

`clock_id` – the clock to be used

The clock to use is defined in `./include/linux/time.h` and represents the various clocks that the system supports (such as the real-time clock or a monotonic clock that simply represents the time from a starting point, such as system boot).

CLOCK_MONOTONIC: a clock that is guaranteed always to move forward in time, but which does not reflect “wall clock time” in any specific way. In the current implementation, *CLOCK_MONOTONIC* resembles the jiffies tick count in that it starts at zero when the system boots and increases monotonically from there.

CLOCK_REALTIME: which matches the current real-world time.

`mode` – timer mode absolute (HRTIMER_MODE_ABS) or relative (HRTIMER_MODE_REL)

Start High Resolution Timer

Once a timer has been initialized, it can be started with the below mentioned function.

hrtimer_start

```
int hrtimer_start(struct hrtimer *timer, ktime_t time, const enum
                        hrtimer_mode mode);
```

This call is used to (Re)start an hrtimer on the current CPU.

Arguments:

`timer` – the timer to be added

`time` – expiry time

`mode` – expiry mode: absolute (HRTIMER_MODE_ABS) or relative (HRTIMER_MODE_REL)

Returns:

0 on success 1 when the timer was active

Stop High Resolution Timer

Using the below function, we can able to stop the High Resolution Timer.

hrtimer_cancel

```
int hrtimer_cancel (struct hrtimer * timer);
```

This will cancel a timer and wait for the handler to finish.

Arguments:

`timer` – the timer to be canceled

Returns:

- 0 when the timer was not active
- 1 when the timer was active

hrtimer_try_to_cancel

```
int hrtimer_try_to_cancel (struct hrtimer * timer);
```

This will try to deactivate a timer.

Arguments:

`timer` – hrtimer to stop

Returns:

- 0 when the timer was not active
- 1 when the timer was active
- -1 when the timer is currently executing the callback function and cannot be stopped

Changing the High Resolution Timer's Timeout

If we are using this High Resolution Timer (hrtimer) as periodic timer, then the callback must set a new expiration time before returning. Usually, restarting timers are used by kernel subsystems which need a callback at a regular interval.

hrtimer_forward

```
u64 hrtimer_forward (struct hrtimer * timer, ktime_t now, ktime_t
                     interval);
```

This will forward the timer expiry so it will expire in the future by the given interval.

Arguments:

`timer` – hrtimer to forward

`now` – forward past this time

`interval` – the interval to forward

Returns:

Returns the number of overruns.

hrtimer_forward_now

```
u64 hrtimer_forward_now(struct hrtimer *timer, ktime_t interval);
```

This will forward the timer expiry so it will expire in the future from now by the given interval.

Arguments:

`timer` – hrtimer to forward

`interval` – the interval to forward

Returns:

Returns the number of overruns.

Check High Resolution Timer's status

The below explained functions are used to get the status and timings.

hrtimer_get_remaining

```
ktime_t hrtimer_get_remaining (const struct hrtimer * timer);
```

This is used to get the remaining time for the timer.

Arguments:

`timer` – hrtimer to get the remaining time

Returns:

Returns the remaining time.

hrtimer_callback_running

```
int hrtimer_callback_running(struct hrtimer *timer);
```

This is the helper function to check, whether the timer is running the callback function.

Arguments:

`timer` – hrtimer to check

Returns:

- 0 when the timer's callback function is not running
- 1 when the timer's callback function is running

hrtimer_cb_get_time

```
ktime_t hrtimer_cb_get_time(struct hrtimer *timer);
```

This function used to get the current time of the given timer.

Arguments:

`timer` – hrtimer to get the time

Returns:

Returns the time.

Using High Resolution Timer In Linux Device Driver

In this example, we took the basic driver source code from this tutorial. On top of that code, we have added the high resolution timer. The steps are mentioned below.

1. Initialize and start the timer in init function
2. After the timeout, registered timer callback will be called.
3. In the timer callback function again we are forwarding the time period and return **HRTIMER_RESTART**. We have to do this step if we want a periodic timer. Otherwise, we can ignore that time forwarding and return **HRTIMER_NORESTART**.
4. Once we are done, we can disable the timer.

Driver Source Code

driver.c:

```
1  #include <linux/kernel.h>
2  #include <linux/init.h>
3  #include <linux/module.h>
4  #include <linux/kdev_t.h>
5  #include <linux/fs.h>
6  #include <linux/cdev.h>
7  #include <linux/device.h>
8  #include <linux/hrtimer.h>
9  #include <linux/ktime.h>
10
11 //Timer Variable
12 #define TIMEOUT   5000 * 1000000L  //nano seconds
13 static struct hrtimer etx_hr_timer;
14 static unsigned int count = 0;
15
16 dev_t dev = 0;
17 static struct class *dev_class;
18 static struct cdev etx_cdev;
19
20 static int __init etx_driver_init(void);
21 static void __exit etx_driver_exit(void);
22 static int etx_open(struct inode *inode, struct file *file);
23 static int etx_release(struct inode *inode, struct file *file);
24 static ssize_t etx_read(struct file *filp, char __user *buf, size_t len,loff_t * of
25 static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * o
26
27 static struct file_operations fops =
28 {
29         .owner          = THIS_MODULE,
30         .read           = etx_read,
31         .write          = etx_write,
32         .open           = etx_open,
33         .release        = etx_release,
34 };
35
36 //Timer Callback function. This will be called when timer expires
37 enum hrtimer_restart timer_callback(struct hrtimer *timer)
38 {
39       /* do your timer stuff here */
40     printk(KERN_INFO "Timer Callback function Called [%d]\n",count++);
41     hrtimer_forward_now(timer,ktime_set(0,TIMEOUT));
42     return HRTIMER_RESTART;
43 }
44
45 static int etx_open(struct inode *inode, struct file *file)
```

```
46  {
47      printk(KERN_INFO "Device File Opened...!!!\n");
48      return 0;
49  }
50
51  static int etx_release(struct inode *inode, struct file *file)
52  {
53      printk(KERN_INFO "Device File Closed...!!!\n");
54      return 0;
55  }
56
57  static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *of
58  {
59      printk(KERN_INFO "Read Function\n");
60      return 0;
61  }
62  static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, lof
63  {
64      printk(KERN_INFO "Write function\n");
65      return 0;
66  }
67
68  static int __init etx_driver_init(void)
69  {
70       ktime_t ktime;
71
72      /*Allocating Major number*/
73      if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
74              printk(KERN_INFO "Cannot allocate major number\n");
75              return -1;
76      }
77      printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
78
79      /*Creating cdev structure*/
80      cdev_init(&etx_cdev,&fops);
81
82      /*Adding character device to the system*/
83      if((cdev_add(&etx_cdev,dev,1)) < 0){
84          printk(KERN_INFO "Cannot add the device to the system\n");
85          goto r_class;
86      }
87
88      /*Creating struct class*/
89      if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
90          printk(KERN_INFO "Cannot create the struct class\n");
91          goto r_class;
92      }
93
94      /*Creating device*/
95      if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
96          printk(KERN_INFO "Cannot create the Device 1\n");
97          goto r_device;
98      }
99
100     ktime = ktime_set(0, TIMEOUT);
101     hrtimer_init(&etx_hr_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
102     etx_hr_timer.function = &timer_callback;
103     hrtimer_start( &etx_hr_timer, ktime, HRTIMER_MODE_REL);
104
105     printk(KERN_INFO "Device Driver Insert...Done!!!\n");
106     return 0;
107 r_device:
108     class_destroy(dev_class);
```

```
109 r_class:
110     unregister_chrdev_region(dev,1);
111     return -1;
112 }
113
114 void __exit etx_driver_exit(void)
115 {
116     //stop the timer
117     hrtimer_cancel(&etx_hr_timer);
118     device_destroy(dev_class,dev);
119     class_destroy(dev_class);
120     cdev_del(&etx_cdev);
121     unregister_chrdev_region(dev, 1);
122     printk(KERN_INFO "Device Driver Remove...Done!!!\n");
123 }
124
125 module_init(etx_driver_init);
126 module_exit(etx_driver_exit);
127
128 MODULE_LICENSE("GPL");
129 MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
130 MODULE_DESCRIPTION("A simple device driver - High Resolution Timer");
131 MODULE_VERSION("1.22");
```

Makefile:

```
1 obj-m += driver.o
2
3 KDIR = /lib/modules/$(shell uname -r)/build
4
5 all:
6     make -C $(KDIR) M=$(shell pwd) modules
7
8 clean:
9     make -C $(KDIR) M=$(shell pwd) clean
```

Building and Testing Driver

- Build the driver by using Makefile (**sudo make**)
- Load the driver using **sudo insmod driver.ko**
- Now see the Dmesg (**dmesg**)

```
linux@embetronicx-VirtualBox: dmesg

[ 2643.773119] Device Driver Insert...Done!!!
[ 2648.773546] Timer Callback function Called [0]
[ 2653.773609] Timer Callback function Called [1]
[ 2658.774170] Timer Callback function Called [2]
[ 2663.773271] Timer Callback function Called [3]
[ 2668.773388] Timer Callback function Called [4]
```

- See timestamp. That callback function is executing every 5 seconds.
- Unload the module using **sudo rmmod driver**

Points to remember

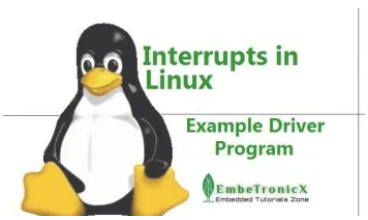
This timer callback function will be executed from the interrupt context. If you want to check that, you can use function **`in_ interrupt( )`**, which takes no parameters and returns nonzero if the processor is currently running in interrupt context, either hardware interrupt or software interrupt. Since it is running in an interrupt context, the user cannot perform some actions inside the callback function mentioned below.

- Go to sleep or relinquish the processor
- Acquire a mutex
- Perform time-consuming tasks
- Access user space virtual memory

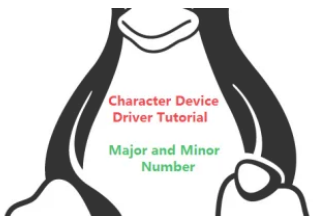
In our next tutorial, we will discuss completion in the Linux device dirver.

Related

Linux Device Driver Tutorial Part 26 – Using



Linux Device Driver Tutorial Part 13 –

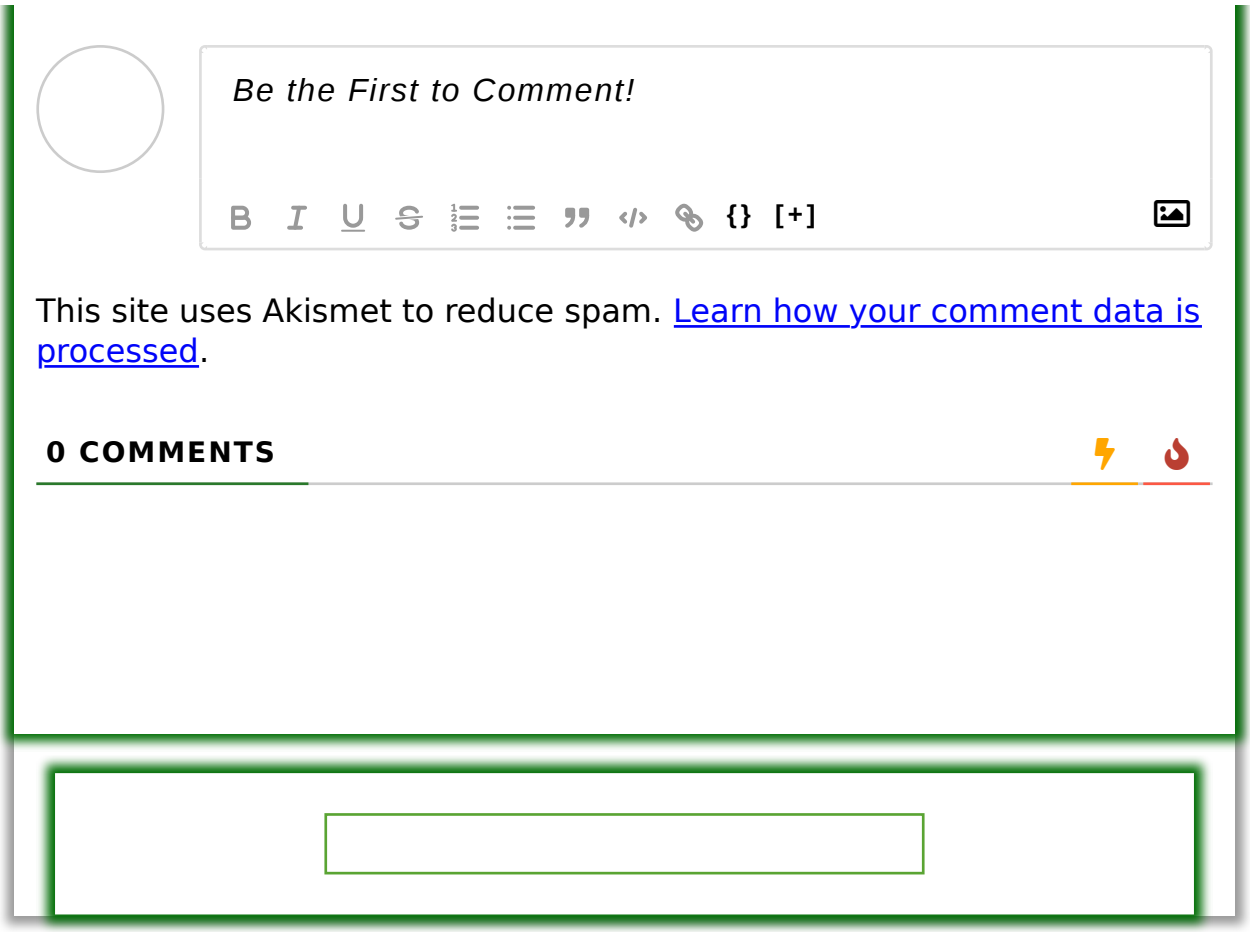


Linux Device Driver Tutorial Part 4 –

Kernel Timer In Linux Device Driver
In "Device Drivers"

Interrupts Example Program in Linux Kernel
In "Device Drivers"

Character Device Driver
In "Device Drivers"


Be the First to Comment!
{} [+]
This site uses Akismet to reduce spam. Learn how your comment data is processed.
0 COMMENTS