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

[ <https://embetronicx.com/tutorials/linux/device-drivers/using-high-resolution-timer-in-linux-device-driver/> ]

This is the Linux Device Driver Tutorial Part 27 – Using High Resolution Timer In Linux Device Driver.

# High Resolution Timer (HRT/hrtimer)

In our [last tutorial](https://embetronicx.com/tutorials/linux/device-drivers/using-kernel-timer-in-linux-device-driver/) we have seen kernel timer. Now we are taking 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>**:

struct hrtimer {

struct rb\_node node;

ktime\_t expires;

int (\* function) (struct hrtimer \*);

struct hrtimer\_base \* base;

};

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 time from a starting point, such as system boot).

*CLOCK\_MONOTONIC*: a clock which 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 below function, we can 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 cancelled

**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 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](https://embetronicx.com/tutorials/linux/device-drivers/linux-device-driver-tutorial-programming/) 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 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 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:**

#include <linux/kernel.h>

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kdev\_t.h>

#include <linux/fs.h>

#include <linux/cdev.h>

#include <linux/device.h>

#include <linux/hrtimer.h>

#include <linux/ktime.h>

//Timer Variable

#define TIMEOUT 5000 \* 1000000L //nano seconds

static struct hrtimer etx\_hr\_timer;

static unsigned int count = 0;

dev\_t dev = 0;

static struct class \*dev\_class;

static struct cdev etx\_cdev;

static int \_\_init etx\_driver\_init(void);

static void \_\_exit etx\_driver\_exit(void);

static int etx\_open(struct inode \*inode, struct file \*file);

static int etx\_release(struct inode \*inode, struct file \*file);

static ssize\_t etx\_read(struct file \*filp, char \_\_user \*buf, size\_t len,loff\_t \* off);

static ssize\_t etx\_write(struct file \*filp, const char \*buf, size\_t len, loff\_t \* off);

static struct file\_operations fops =

{

.owner = THIS\_MODULE,

.read = etx\_read,

.write = etx\_write,

.open = etx\_open,

.release = etx\_release,

};

//Timer Callback function. This will be called when timer expires

enum hrtimer\_restart timer\_callback(struct hrtimer \*timer)

{

/\* do your timer stuff here \*/

printk(KERN\_INFO "Timer Callback function Called [%d]\n",count++);

hrtimer\_forward\_now(timer,ktime\_set(0,TIMEOUT));

return HRTIMER\_RESTART;

}

static int etx\_open(struct inode \*inode, struct file \*file)

{

printk(KERN\_INFO "Device File Opened...!!!\n");

return 0;

}

static int etx\_release(struct inode \*inode, struct file \*file)

{

printk(KERN\_INFO "Device File Closed...!!!\n");

return 0;

}

static ssize\_t etx\_read(struct file \*filp, char \_\_user \*buf, size\_t len, loff\_t \*off)

{

printk(KERN\_INFO "Read Function\n");

return 0;

}

static ssize\_t etx\_write(struct file \*filp, const char \_\_user \*buf, size\_t len, loff\_t \*off)

{

printk(KERN\_INFO "Write function\n");

return 0;

}

static int \_\_init etx\_driver\_init(void)

{

ktime\_t ktime;

/\*Allocating Major number\*/

if((alloc\_chrdev\_region(&dev, 0, 1, "etx\_Dev")) <0){

printk(KERN\_INFO "Cannot allocate major number\n");

return -1;

}

printk(KERN\_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

/\*Creating cdev structure\*/

cdev\_init(&etx\_cdev,&fops);

/\*Adding character device to the system\*/

if((cdev\_add(&etx\_cdev,dev,1)) < 0){

printk(KERN\_INFO "Cannot add the device to the system\n");

goto r\_class;

}

/\*Creating struct class\*/

if((dev\_class = class\_create(THIS\_MODULE,"etx\_class")) == NULL){

printk(KERN\_INFO "Cannot create the struct class\n");

goto r\_class;

}

/\*Creating device\*/

if((device\_create(dev\_class,NULL,dev,NULL,"etx\_device")) == NULL){

printk(KERN\_INFO "Cannot create the Device 1\n");

goto r\_device;

}

ktime = ktime\_set(0, TIMEOUT);

hrtimer\_init(&etx\_hr\_timer, CLOCK\_MONOTONIC, HRTIMER\_MODE\_REL);

etx\_hr\_timer.function = &timer\_callback;

hrtimer\_start( &etx\_hr\_timer, ktime, HRTIMER\_MODE\_REL);

printk(KERN\_INFO "Device Driver Insert...Done!!!\n");

return 0;

r\_device:

class\_destroy(dev\_class);

r\_class:

unregister\_chrdev\_region(dev,1);

return -1;

}

void \_\_exit etx\_driver\_exit(void)

{

//stop the timer

hrtimer\_cancel(&etx\_hr\_timer);

device\_destroy(dev\_class,dev);

class\_destroy(dev\_class);

cdev\_del(&etx\_cdev);

unregister\_chrdev\_region(dev, 1);

printk(KERN\_INFO "Device Driver Remove...Done!!!\n");

}

module\_init(etx\_driver\_init);

module\_exit(etx\_driver\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");

MODULE\_DESCRIPTION("A simple device driver - High Resolution Timer");

MODULE\_VERSION("1.22");

## Makefile:

obj-m += driver.o

KDIR = /lib/modules/$(shell uname -r)/build

all:

make -C $(KDIR) M=$(shell pwd) modules

clean:

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**)

*$ 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 the 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 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 interrupt context, 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