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 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>**:

```
structhrtimer {
structrb_node node;
ktime_t expires;
int (* function) (structhrtimer *);
structhrtimer_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

voidhrtimer_init(structhrtimer *timer, clockid_tclock_id, enumhrtimer_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

inthrtimer_start(structhrtimer *timer, ktime_t time, constenumhrtimer_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 able to stop the High Resolution Timer.

hrtimer_cancel

inthrtimer_cancel (structhrtimer * 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

inthrtimer_try_to_cancel (structhrtimer * 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 (structhrtimer * 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(structhrtimer *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_thrtimer_get_remaining (conststructhrtimer * 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

inthrtimer_callback_running(structhrtimer *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_thrtimer_cb_get_time(structhrtimer *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 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
staticstructhrtimeretx_hr_timer;
static unsigned int count = 0;

dev_tdev = 0;
staticstruct class *dev_class;
staticstructcdevetx_cdev;

staticint __initetx_driver_init(void);
```

```
static void __exit etx_driver_exit(void);
staticintetx_open(structinode *inode, struct file *file);
staticintetx_release(structinode *inode, struct file *file);
staticssize_tetx_read(struct file *filp, char __user *buf, size_tlen,loff_t * off);
staticssize_tetx_write(struct file *filp, const char *buf, size_tlen, loff_t * off);

staticstructfile_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
enumhrtimer_restarttimer_callback(structhrtimer *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;
}

staticintetx_open(structinode *inode, struct file *file)
{
printk(KERN_INFO "Device File Opened...!!!\n");
return 0;
}

staticintetx_release(structinode *inode, struct file *file)
{
printk(KERN_INFO "Device File Closed...!!!\n");
return 0;
}

staticssize_tetx_read(struct file *filp, char __user *buf, size_tlen, loff_t *off)
```

```
{
printk(KERN_INFO "Read Function\n");
return 0;
}
staticssize_tetx_write(struct file *filp, const char __user *buf, size_tlen, loff_t
*off)
{
printk(KERN_INFO "Write function\n");
return 0;
}

staticint __initetx_driver_init(void)
{
ktime_tktime;

   /*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));

   /*Creatingcdev 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");
gotor_class;
   }

   /*Creatingstruct class*/
if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
printk(KERN_INFO "Cannot create the struct class\n");
gotor_class;
   }
```

```
   /*Creating device*/
if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
printk(KERN_INFO "Cannot create the Device 1\n");
gotor_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 **sudoinsmoddriver.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 the timestamp. That callback function is executing every 5 seconds.
- Unload the module using **sudormmod 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