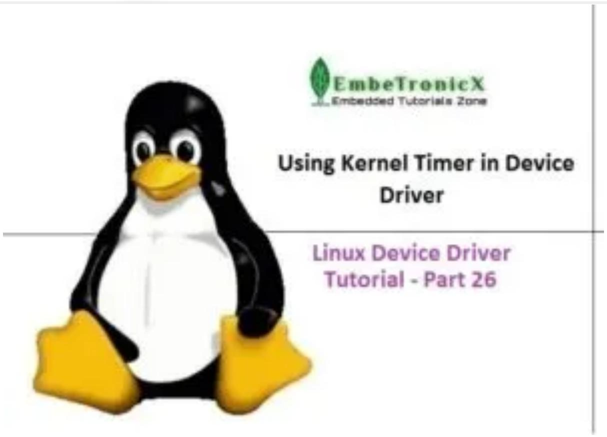
EmbeTronicX
Embedded Tutorials Zone

Sidebar

Home → Tutorials → Linux → Device Drivers → **Linux Device Driver Tutorial Part 26 – Using Kernel Timer In Linux Device Driver**

Device Drivers



Linux Device Driver Tutorial Part 26 – Using Kernel Timer In Linux Device Driver

2

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 26 – Using Kernel Timer In Linux

Device Driver. (Updated on 01-April-2020).

Post Contents [hide]

Timer

Introduction

What is a timer in general? According to Wikipedia, A timer is a specialized type of clock used for measuring specific time intervals. Timers can be categorized into two main types. A timer that counts upwards from zero for measuring elapsed time is often called a stopwatch, while a device that counts down from a specified time interval is more usually called a timer.

Timer in Linux Kernel

In Linux, kernel keeps track of the flow of time by means of timer interrupts. This timer interrupts are generated at regular timer intervals by using system's timing hardware. Every time a timer interrupt occurs, the value of an internal kernel counter is incremented. The counter is initialized to 0 at system boot, so it represents the number of clock ticks since last boot.

Kernel timer offers less precision but is more efficient in situations where the timer will probably be canceled before it fires. There are many places in the kernel where timers are used to detect when a device or a network peer has failed to respond within the expected time.

When you want to do some action after some time, then kernel timers are one of the options for you. These timers are used to schedule the execution of a function at a particular time in the future, based on the clock tick, and can be used for a variety of tasks.

Uses of Kernel Timers

- Polling a device by checking its state at regular intervals when the hardware can't fire interrupts.
- The user wants to send some messages to another device at regular intervals.
- Send error when some action didn't happen in a particular time period.
- Etc.

Kernel Timer API

Linux Kernel provides the driver to create timers that are not periodic by default, register the timers and delete the timers.

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

```
struct timer_list {
    /* ... */
    unsigned long expires;
    void (*function)(unsigned long);
    unsigned long data;
};
```

The **expires** field contains the expiration time of the timer (in jiffies). On expiration, **function()** will be called with the given **data** value.

Initialize / Setup Kernel Timer

There are multiple ways to Initialize / Setup Kernel Timer. We'll see one by one.

init_timer

```
void fastcall init_timer ( struct timer_list * timer);
```

This function is used to initialize the timer. **init_timer** must be done to a timer prior to calling any of the other timer functions. If you are using this function to initialize the timer, then you need to set the callback **function** and **data** of the **timer_list** structure manually.

Argument:

timer – the timer to be initialized

setup_timer

void setup_timer(timer, function, data);

Instead of initializing timer manually by calling **init_timer**, you can use this function to set **data** and **function** of **timer_list** structure and initialize the timer. *This is recommended to use*. This API will be available for the older kernel version. If you are using the newer kernel, then you have to use below API (**timer_setup**).

Argument:

timer – the timer to be initialized

function – Callback function to be called when the timer expires. In this callback function, the argument will be **unsigned long**.

data – data has to be given to the callback function

Example

```
/* setup your timer to call my_timer_callback */
setup_timer(&etx_timer, timer_callback, 0);

//Timer Callback function. This will be called when timer expires
void timer_callback(unsigned long data)
{

}
```

timer_setup

If you use a newer kernel version, then setup_timer won't work. So you need to use this **timer_setup** function.

2

void timer_setup(timer, function, data);

Instead of initializing timer manually by calling **init_timer**, you can use this function to set **data** and **function** of **timer_list** structure and initialize the timer. *This is recommended to use*.

Argument:

timer – the timer to be initialized

function – Callback function to be called when timer expires. In this callback function, the argument will be **struct timer_list ***.

data – data has to be given to the callback function

Example

```
/* setup your timer to call my_timer_callback */
timer_setup(&etx_timer, timer_callback, 0);

//Timer Callback function. This will be called when timer expires
void timer_callback(struct timer_list * data)
{

}
```

DEFINE_TIMER

DEFINE_TIMER(_name, _function, _expires, _data)

If we are using this method, then no need to create the **timer_list** structure on our side. The kernel will create the structure in the name of **_name** and initialize it.

Argument:

_name – name of the timer_list structure to be created

_function – Callback function to be called when the timer expires

_expires – the expiration time of the timer (in jiffies)

_data – data has to be given to the callback function

Start a Kernel Timer

add_timer

```
void add_timer(struct timer_list *timer);
```

This will start a timer.

Argument:

`timer` – the timer needs to be started

Modifying Kernel Timer’s timeout

mod_timer

```
int mod_timer (struct timer_list * timer, unsigned long expires);
```

This function is used to modify a timer’s timeout. This is a more efficient way to update the expire field of an active timer (if the timer is inactive it will be activated).

`mod_timer(timer, expires)` is equivalent to:

```
del_timer(timer); timer->expires = expires; add_timer(timer);
```

Argument:

`timer` – the timer needs to modify the timer period

`expires` – the updated expiration time of the timer (in jiffies)

Return:

The function returns whether it has modified a pending timer or not.

0 – `mod_timer` of an inactive timer

1 – `mod_timer` of an active timer

Stop a Kernel Timer

These below functions will be used to deactivate the kernel timers.

del_timer

```
int del_timer (struct timer_list * timer);
```

This will deactivate a timer. This works on both active and inactive timers.

Argument:

`timer` – the timer needs to be deactivated

Return:

The function returns whether it has deactivated a pending timer or not.

0 – `del_timer` of an inactive timer

1 – `del_timer` of an active timer

del_timer_sync

```
int del_timer_sync (struct timer_list * timer);
```

2

This will deactivate a timer and wait for the handler to finish. This works on both active and inactive timers.

Argument:

`timer` – the timer needs to be deactivated

Return:

The function returns whether it has deactivated a pending timer or not.

0 – `del_timer_sync`of an inactive timer

1 – `del_timer_sync` of an active timer

Note: *callers must prevent restarting of the timer, otherwise this function is meaningless. It must not be called from interrupt contexts. The caller must not hold locks that would prevent completion of the timer's handler. The timer's handler must not call add_timer_on. Upon exit, the timer is not queued and the handler is not running on any CPU.*

Check Kernel Timer status

timer_pending

```
int timer_pending(const struct timer_list * timer);
```

This will tell whether a given timer is currently pending, or not. Callers must ensure serialization wrt. other operations done to this timer, eg. interrupt contexts or other CPUs on SMP.

Argument:

`timer` – the timer needs to check the status

Return:

The function returns whether the timer is pending or not.

0 – timer is not pending

1 – timer is pending

Device Driver Source Code

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

1. Initialize the timer and set the time interval
2. After the timeout, a registered timer callback will be called.
3. In the timer callback function again we are re-enabling the timer. We have to do this step if we want a periodic timer. Otherwise, we can ignore this.
4. Once we have done, we can disable the timer.

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/timer.h>
#include <linux/jiffies.h>

//Timer Variable
#define TIMEOUT 5000    //milliseconds
static struct timer_list etx_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 * of
static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * o

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
void timer_callback(struct timer_list * data)
{
     /* do your timer stuff here */
    printk(KERN_INFO "Timer Callback function Called [%d]\n",count++);

    /*
       Re-enable timer. Because this function will be called only first time.
```

```
       If we re-enable this will work like periodic timer.
    */
    mod_timer(&etx_timer, jiffies + msecs_to_jiffies(TIMEOUT));
}

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 *of
{
    printk(KERN_INFO "Read Function\n");
    return 0;
}
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, lof
{
    printk(KERN_INFO "Write function\n");
    return 0;
}

static int __init etx_driver_init(void)
{
    /*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;
    }

    /* setup your timer to call my_timer_callback */
    timer_setup(&etx_timer, timer_callback, 0);       //If you face some issues and

    /* setup timer interval to based on TIMEOUT Macro */
    mod_timer(&etx_timer, jiffies + msecs_to_jiffies(TIMEOUT));
```

```

    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)
{
    /* remove kernel timer when unloading module */
    del_timer(&etx_timer);

    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 - Kernel Timer");
MODULE_VERSION("1.21");
```

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**)


```
linux@embetronicx-VirtualBox: dmesg

[ 2253.635127] Device Driver Insert...Done!!!
[ 2258.642048] Timer Callback function Called [0]
[ 2263.647050] Timer Callback function Called [1]
[ 2268.652684] Timer Callback function Called [2]
[ 2273.658274] Timer Callback function Called [3]
[ 2278.663885] Timer Callback function Called [4]
[ 2283.668997] Timer Callback function Called [5]
```

```
[ 2288.675109] Timer Callback function Called [6]
[ 2293.680160] Timer Callback function Called [7]
[ 2298.685771] Timer Callback function Called [8]
[ 2303.691392] Timer Callback function Called [9]
[ 2308.697013] Timer Callback function Called [10]
[ 2313.702033] Timer Callback function Called [11]
[ 2318.707772] Timer Callback function Called [12]
```

- 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 the 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.

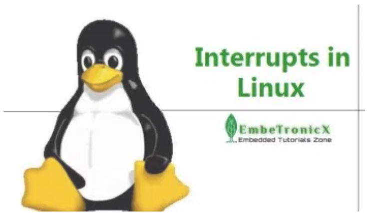
- Go to sleep or relinquish the processor
- Acquire a mutex
- Perform time-consuming tasks
- Access user space virtual memory

In our next tutorial we will see the High Resolution Timer (hrtimer).

Related

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

Linux Device Driver Tutorial Part 13 – Interrupts Example Program in Linux Kernel
In "Device Drivers"



Linux Device Driver Tutorial Part 12 - Interrupts in Linux Kernel
In "Device Drivers"

Subscribe ▾

Connect with | Login

Join the discussion

B I U S {} [+]

This site uses Akismet to reduce spam. [Learn how your comment data is processed](#).

2 COMMENTS

Oldest ▾

Hug

January 31, 2020 8:52 AM

NOTE: its NOT "setup_timer", that function does not exist!

Its "timer_setup"

2

Loading...

0 Reply

EmbeTronicX

Reply to Hug April 1, 2020 3:19 AM

Dear Hug,
Thanks for your input. We have updated our driver source to support newer kernel.

Loading...

0 Reply

2