# Part 26 – Using Kernel Timer In Linux Device Driver

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

# Timer

## Introduction

*What is a timer in general?* A timer is a specialized type of clock used for measuring specific time intervals. Timers can be categorized into two main types. A timer which counts upwards from zero for measuring elapsed time is often called a stopwatch, while a device which 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 cancelled 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 option for you. These timers are used to schedule 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.
* User wants to send some message to other device at regular intervals.
* Send error when some action didn’t happened in particular time period.
* Etc.

# Kernel Timer API

Linux Kernel provides the driver to create timers which 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 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 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 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 newer kernel version, then setup\_timer won’t work. So you need to use this **timer\_setup** function.

**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 in our side. 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 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 start

## 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 be 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 deactivate

**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);**

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 deactivate

**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 which 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 status

**Return:**

The function returns whether 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](https://embetronicx.com/tutorials/linux/device-drivers/linux-device-driver-tutorial-programming/) 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 timeout, 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 periodic timer. Otherwise we can ignore this.
4. Once we done, we can disable the timer.

#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 \* 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

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 \*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)

{

/\*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 using older kernel version, then you can try setup\_timer API(Change Callback function's argument to unsingned long instead of struct timer\_list \*.

/\* 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**)

*$ 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 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

In our [next tutorial](https://embetronicx.com/tutorials/linux/device-drivers/using-high-resolution-timer-in-linux-device-driver) we will see the High Resolution Timer (hrtimer).