# Part 28 – Thread Completion in Linux Device Driver

[ <https://embetronicx.com/tutorials/linux/device-drivers/completion-in-linux/> ]

**Prerequisites**

In the example section, I had used kthread to explain this completion. If you don’t know what is kthread and how to use it, then I would recommend you to explore that by using below link.

1. [Kthread Tutorial in Linux Kernel](https://embetronicx.com/tutorials/linux/device-drivers/linux-device-drivers-tutorial-kernel-thread)
2. [Waitqueue Tutorial in Linux Kernel](https://www.embetronicx.com/tutorials/linux/device-drivers/waitqueue-in-linux-device-driver-tutorial/)

# Completion

**Completion**, the name itself says. When we want to notify or wakeup some thread or something when we finished some work, then we can use completion.

We’ll take one situation. We want to wait one thread for something to run. Until that time that thread has to sleep. Once that process finished then we need to wake up that thread which is sleeping. We can do this by using completion without race conditions.

This completions are a synchronization mechanism which is good method in the above situation mentioned rather than using improper locks/semaphores and busy-loops.

# Completion in Linux Device Driver

I

n Linux kernel, Completions are developed by using [waitqueue](https://www.embetronicx.com/tutorials/linux/device-drivers/waitqueue-in-linux-device-driver-tutorial/).

The advantage of using completions is that they have a well-defined, focused purpose which makes it very easy to see the intent of the code, but they also result in more efficient code as all threads can continue execution until the result is actually needed, and both the waiting and the signalling is highly efficient using low level scheduler sleep/wakeup facilities.

There are 5 important steps in Completions.

1. Initializing Completion.
2. Re-Initializing Completion.
3. Waiting for completion (The code is waiting and sleeping for something to finish)
4. Waking Up Task (Sending signal to sleeping part)
5. Check the status

## Initialize Completion

We have to include **<linux/completion.h>**and creating a variable of type **struct completion,**which has only two fields:

struct completion {

unsigned int done;

wait\_queue\_head\_t wait;

};

Where, **wait**is the waitqueue to place tasks on for waiting (if any). **done**is the completion flag for indicating whether it’s completed or not.

We can create the struct variable in two ways.

1. Static Method
2. Dynamic Method

You can use any one of the method.

### Static Method

**DECLARE\_COMPLETION(data\_read\_done);**

Where the “**data\_read\_done**” is the name of the struct which is going to create statically.

### Dynamic Method

**init\_completion (struct completion \* x);**

Where, **x** – completion structure that is to be initialized

**Example**

struct completion data\_read\_done;

init\_completion(&data\_read\_done);

In this **init\_completion** call we initialize the waitqueue and set **done**to 0, i.e. “not completed” or “not done”.

## Re-Initializing Completion

**reinit\_completion (struct completion \* x);**

Where, **x** – completion structure that is to be reinitialized

**Example:**

reinit\_completion(&data\_read\_done);

This function should be used to reinitialize a completion structure so it can be reused. This is especially important after **complete\_all** is used. This simply resets the ->done field to 0 (“not done”), without touching the waitqueue. Callers of this function must make sure that there are no racy wait\_for\_completion() calls going on in parallel.

## Waiting for completion

For a thread to wait for some concurrent activity to finish, it calls the any one of the function based on the use case.

### wait\_for\_completion

This is used to make the function waits for completion of a task.

**void wait\_for\_completion (struct completion \* x);**

Where, **x** – holds the state of this particular completion

This waits to be signaled for completion of a specific task. It is NOT interruptible and there is no timeout.

**Example:**

wait\_for\_completion (&data\_read\_done);

Note that **wait\_for\_completion()** is calling **spin\_lock\_irq()/spin\_unlock\_irq()**, so it can only be called safely when you know that interrupts are enabled. Calling it from IRQs-off atomic contexts will result in hard-to-detect spurious enabling of interrupts.

### wait\_for\_completion\_timeout

This is used to make the function waits for completion of a task with timeout. Timeouts are preferably calculated with msecs\_to\_jiffies() or usecs\_to\_jiffies(), to make the code largely HZ-invariant.

**unsigned long wait\_for\_completion\_timeout (struct completion \* x, unsigned long timeout);**

where,**x** – holds the state of this particular completion

**timeout**– timeout value in jiffies

This waits for either a completion of a specific task to be signaled or for a specified timeout to expire. The timeout is in jiffies. It is not interruptible.

It **returns** **0**if timed out, and **positive** (at least 1, or number of jiffies left till timeout) if completed.

**Example:**

wait\_for\_completion\_timeout (&data\_read\_done, 10000);

### wait\_for\_completion\_interruptible

This waits for completion of a specific task to be signaled. It is interruptible.

**int wait\_for\_completion\_interruptible (struct completion \* x);**

where, **x** – holds the state of this particular completion

It return **-ERESTARTSYS** if interrupted, **0** if completed.

### wait\_for\_completion\_interruptible\_timeout

This waits for either a completion of a specific task to be signaled or for a specified timeout to expire. It is interruptible. The timeout is in jiffies. Timeouts are preferably calculated with msecs\_to\_jiffies() or usecs\_to\_jiffies(), to make the code largely HZ-invariant.

**long wait\_for\_completion\_interruptible\_timeout (struct completion \* x, unsigned long timeout);**

where,**x** – holds the state of this particular completion

**timeout**– timeout value in jiffies

It return **-ERESTARTSYS** if interrupted, **0** if timed out, positive (at least 1, or number of jiffies left till timeout) if completed.

### wait\_for\_completion\_killable

This waits to be signaled for completion of a specific task. It can be interrupted by a kill signal.

**int wait\_for\_completion\_killable (struct completion \* x);**

where,**x** – holds the state of this particular completion

It return **-ERESTARTSYS** if interrupted, **0** if completed.

### wait\_for\_completion\_killable\_timeout

This waits for either a completion of a specific task to be signaled or for a specified timeout to expire. It can be interrupted by a kill signal. The timeout is in jiffies. Timeouts are preferably calculated with msecs\_to\_jiffies() or usecs\_to\_jiffies(), to make the code largely HZ-invariant.

**long wait\_for\_completion\_killable\_timeout (struct completion \* x, unsigned long timeout);**

where,**x** – holds the state of this particular completion

**timeout**– timeout value in jiffies

It return **-ERESTARTSYS** if interrupted, **0** if timed out, positive (at least 1, or number of jiffies left till timeout) if completed.

### try\_wait\_for\_completion

This function will not put the thread on the wait queue but rather returns false if it would need to enqueue (block) the thread, else it consumes one posted completion and returns true.

**bool try\_wait\_for\_completion (struct completion \* x);**

where,**x** – holds the state of this particular completion

It returns **0** if a completion is not available **1** if a got it succeeded.

This **try\_wait\_for\_completion()** is safe to be called in IRQ or atomic context.

## Waking Up Task

### complete

This will wake up a single thread waiting on this completion. Threads will be awakened in the same order in which they were queued.

**void complete (struct completion \* x);**

where,**x** – holds the state of this particular completion

**Example:**

complete(&data\_read\_done);

### complete\_all

This will wake up all threads waiting on this particular completion event.

**void complete\_all (struct completion \* x);**

where,**x** – holds the state of this particular completion

## Check the status

### completion\_done

This is the test to see if a completion has any waiters.

**bool completion\_done (struct completion \* x);**

where,**x** – holds the state of this particular completion

It returns 0 if there are waiters (wait\_for\_completion in progress) **1** if there are no waiters.

This **completion\_done()** is safe to be called in IRQ or atomic context.

# Driver Source Code – Completion in Linux

First i will explain you the concept of driver code.

In this source code, two places we are sending complete call. One from read function and another one from driver exit function.

I’ve created one thread (**wait\_function**) which has **while(1)**. That thread will always wait for the event to complete. It will be sleeping until it gets complete call. When it gets the complete call, it will check the condition. If condition is 1 then the complete came from read function. It it is 2, then the complete came from exit function. If complete came from read, it will print the read count  and it will again wait.If its from exit function, it will exit from the thread.

Here I’ve added two versions of code.

1. Completion created by static method
2. Completion created by dynamic method

But operation wise, both are same.

You can also find the source code here.

## Completion created by static method

#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/slab.h> //kmalloc()

#include <linux/uaccess.h> //copy\_to/from\_user()

#include <linux/kthread.h>

#include <linux/completion.h> // Required for the completion

uint32\_t read\_count = 0;

static struct task\_struct \*wait\_thread;

DECLARE\_COMPLETION(data\_read\_done);

dev\_t dev = 0;

static struct class \*dev\_class;

static struct cdev etx\_cdev;

int completion\_flag = 0;

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

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

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Driver Functions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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,

};

static int wait\_function(void \*unused)

{

while(1) {

printk(KERN\_INFO "Waiting For Event...\n");

wait\_for\_completion (&data\_read\_done);

if(completion\_flag == 2) {

printk(KERN\_INFO "Event Came From Exit Function\n");

return 0;

}

printk(KERN\_INFO "Event Came From Read Function - %d\n", ++read\_count);

completion\_flag = 0;

}

do\_exit(0);

return 0;

}

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");

completion\_flag = 1;

if(!completion\_done (&data\_read\_done)) {

complete (&data\_read\_done);

}

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);

etx\_cdev.owner = THIS\_MODULE;

etx\_cdev.ops = &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;

}

//Create the kernel thread with name 'mythread'

wait\_thread = kthread\_create(wait\_function, NULL, "WaitThread");

if (wait\_thread) {

printk("Thread Created successfully\n");

wake\_up\_process(wait\_thread);

} else

printk(KERN\_INFO "Thread creation failed\n");

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)

{

completion\_flag = 2;

if(!completion\_done (&data\_read\_done)) {

complete (&data\_read\_done);

}

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 or admin@embetronicx.com>");

MODULE\_DESCRIPTION("A simple device driver - Completion (Static Method)");

MODULE\_VERSION("1.23");

## Completion created by dynamic method

#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/slab.h> //kmalloc()

#include <linux/uaccess.h> //copy\_to/from\_user()

#include <linux/kthread.h>

#include <linux/completion.h> // Required for the completion

uint32\_t read\_count = 0;

static struct task\_struct \*wait\_thread;

struct completion data\_read\_done;

dev\_t dev = 0;

static struct class \*dev\_class;

static struct cdev etx\_cdev;

int completion\_flag = 0;

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

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

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Driver Functions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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,

};

static int wait\_function(void \*unused)

{

while(1) {

printk(KERN\_INFO "Waiting For Event...\n");

wait\_for\_completion (&data\_read\_done);

if(completion\_flag == 2) {

printk(KERN\_INFO "Event Came From Exit Function\n");

return 0;

}

printk(KERN\_INFO "Event Came From Read Function - %d\n", ++read\_count);

completion\_flag = 0;

}

do\_exit(0);

return 0;

}

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");

completion\_flag = 1;

if(!completion\_done (&data\_read\_done)) {

complete (&data\_read\_done);

}

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);

etx\_cdev.owner = THIS\_MODULE;

etx\_cdev.ops = &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;

}

//Create the kernel thread with name 'mythread'

wait\_thread = kthread\_create(wait\_function, NULL, "WaitThread");

if (wait\_thread) {

printk("Thread Created successfully\n");

wake\_up\_process(wait\_thread);

} else

printk(KERN\_INFO "Thread creation failed\n");

//Initializing Completion

init\_completion(&data\_read\_done);

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)

{

completion\_flag = 2;

if(!completion\_done (&data\_read\_done)) {

complete (&data\_read\_done);

}

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 or admin@embetronicx.com>");

MODULE\_DESCRIPTION("A simple device driver - Completion (Dynamic Method)");

MODULE\_VERSION("1.24");

## MakeFile

obj-m += completion\_static.o

obj-m += completion\_dynamic.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**
* Then Check the Dmesg

*Major = 246 Minor = 0  
Thread Created successfully  
Device Driver Insert…Done!!!  
Waiting For Event…*

* So that thread is waiting for the event. Now we will send the event by reading the driver using **sudo cat /dev/etx\_device**
* Now check the dmesg

*Device File Opened…!!!  
Read Function  
Event Came From Read Function – 1  
Waiting For Event…  
Device File Closed…!!!*

* We send the complete  from read function, So it will print the read count and then again it will sleep. Now send the event from exit function by **sudo rmmod driver**

*Event Came From Exit Function  
Device Driver Remove…Done!!!*

* Now the condition was 2. So it will return from the thread and remove the driver.