# Part 15 – Workqueue in Linux Kernel Part 2

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

In our previous tutorial we have seen the [Workqueue using Static method](https://www.embetronicx.com/tutorials/linux/device-drivers/workqueue-in-linux-kernel/) through Device Driver Programming. Now we are going to see Linux Device Driver Tutorial Part 15 – Workqueue in Linux (Dynamic Creation Method).

# Initialize work using Static Method

The below call creates a workqueue by the name work and the function that gets scheduled in the queue is work\_fn.

**INIT\_WORK(work,work\_fn)**

Where,

*name:* The name of the “work\_struct” structure that has to be created.  
*func:* The function to be scheduled in this workqueue.

# Schedule work to the Workqueue

These below functions used to allocate the work to the queue.

## Schedule\_work

|  |
| --- |
| This function puts a job in the kernel-global workqueue if it was not already queued and leaves it in the same position on the kernel-global workqueue otherwise.  **int schedule\_work( struct work\_struct \*work );**  where,  *work* – job to be done  Returns zero if *work* was already on the kernel-global workqueue and non-zero otherwise. |

## Scheduled\_delayed\_work

|  |
| --- |
| After waiting for a given time this function puts a job in the kernel-global workqueue.  **int scheduled\_delayed\_work( struct delayed\_work \*dwork, unsigned long delay );**  where,  *dwork* – job to be done  delay – number of jiffies to wait or 0 for immediate execution. |

## Schedule\_work\_on

|  |
| --- |
| This puts a job on a specific cpu.  **int schedule\_work\_on( int cpu, struct work\_struct \*work );**  where,  *cpu*– cpu to put the work task on  *work*– job to be done |

## Scheduled\_delayed\_work\_on

|  |
| --- |
| After waiting for a given time this puts a job in the kernel-global workqueue on the specified CPU.  **int scheduled\_delayed\_work\_on(**  **int cpu, struct delayed\_work \*dwork, unsigned long delay );**  where,  cpu – cpu to put the work task on  *dwork* – job to be done  delay – number of jiffies to wait or 0 for immediate execution |

# Delete work from workqueue

There are also a number of helper functions that you can use to flush or cancel work on work queues. To flush a particular work item and block until the work is complete, you can make a call to flush\_work. All work on a given work queue can be completed using a call to . In both cases, the caller blocks until the operation is complete. To flush the kernel-global work queue, call flush\_scheduled\_work.

int flush\_work( struct work\_struct \*work );

void flush\_scheduled\_work( void );

# Cancel Work from workqueue

You can cancel work if it is not already executing in a handler. A call to cancel\_work\_sync() will terminate the work in the queue or block until the callback has finished (if the work is already in progress in the handler). If the work is delayed, you can use a call to cancel\_delayed\_work\_sync.

int cancel\_work\_sync( struct work\_struct \*work );

int cancel\_delayed\_work\_sync( struct delayed\_work \*dwork );

# Check workqueue

Finally, you can find out whether a work item is pending (not yet executed by the handler) with a call to work\_pending() or delayed\_work\_pending().

work\_pending( work );

delayed\_work\_pending( work );

## Driver Source Code

In that source code, When we read the /dev/etx\_device interrupt will hit (To understand interrupts in Linux go to [this tutorial](https://www.embetronicx.com/tutorials/linux/device-drivers/linux-device-driver-tutorial-part-13-interrupt-example-program-in-linux-kernel/)). Whenever interrupt hits, I’m scheduling the work to the workqueue. In real workqueues, this function can be used to carry out any operations that need to be scheduled.

#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/sysfs.h>

#include<linux/kobject.h>

#include <linux/interrupt.h>

#include <asm/io.h>

#include <linux/workqueue.h> // Required for workqueues

#define IRQ\_NO 11

/\* Work structure \*/

static struct work\_struct workqueue;

void workqueue\_fn(struct work\_struct \*work);

/\*Workqueue Function\*/

void workqueue\_fn(struct work\_struct \*work)

{

printk(KERN\_INFO "Executing Workqueue Function\n");

}

//Interrupt handler for IRQ 11.

static irqreturn\_t irq\_handler(int irq,void \*dev\_id) {

printk(KERN\_INFO "Shared IRQ: Interrupt Occurred");

/\*Allocating work to queue\*/

schedule\_work(&workqueue);

return IRQ\_HANDLED;

}

volatile int etx\_value = 0;

dev\_t dev = 0;

static struct class \*dev\_class;

static struct cdev etx\_cdev;

struct kobject \*kobj\_ref;

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

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

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Driver Fuctions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Sysfs Fuctions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static ssize\_t sysfs\_show(struct kobject \*kobj,

struct kobj\_attribute \*attr, char \*buf);

static ssize\_t sysfs\_store(struct kobject \*kobj,

struct kobj\_attribute \*attr,const char \*buf, size\_t count);

struct kobj\_attribute etx\_attr = \_\_ATTR(etx\_value, 0660, sysfs\_show, sysfs\_store);

static struct file\_operations fops =

{

.owner = THIS\_MODULE,

.read = etx\_read,

.write = etx\_write,

.open = etx\_open,

.release = etx\_release,

};

static ssize\_t sysfs\_show(struct kobject \*kobj,

struct kobj\_attribute \*attr, char \*buf)

{

printk(KERN\_INFO "Sysfs - Read!!!\n");

return sprintf(buf, "%d", etx\_value);

}

static ssize\_t sysfs\_store(struct kobject \*kobj,

struct kobj\_attribute \*attr,const char \*buf, size\_t count)

{

printk(KERN\_INFO "Sysfs - Write!!!\n");

sscanf(buf,"%d",&etx\_value);

return count;

}

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");

asm("int $0x3B"); // Corresponding to irq 11

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;

}

/\*Creating a directory in /sys/kernel/ \*/

kobj\_ref = kobject\_create\_and\_add("etx\_sysfs",kernel\_kobj);

/\*Creating sysfs file for etx\_value\*/

if(sysfs\_create\_file(kobj\_ref,&etx\_attr.attr)){

printk(KERN\_INFO"Cannot create sysfs file......\n");

goto r\_sysfs;

}

if (request\_irq(IRQ\_NO, irq\_handler, IRQF\_SHARED, "etx\_device", (void \*)(irq\_handler))) {

printk(KERN\_INFO "my\_device: cannot register IRQ ");

goto irq;

}

/\*Creating work by Dynamic Method \*/

INIT\_WORK(&workqueue,workqueue\_fn);

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

return 0;

irq:

free\_irq(IRQ\_NO,(void \*)(irq\_handler));

r\_sysfs:

kobject\_put(kobj\_ref);

sysfs\_remove\_file(kernel\_kobj, &etx\_attr.attr);

r\_device:

class\_destroy(dev\_class);

r\_class:

unregister\_chrdev\_region(dev,1);

cdev\_del(&etx\_cdev);

return -1;

}

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

{

free\_irq(IRQ\_NO,(void \*)(irq\_handler));

kobject\_put(kobj\_ref);

sysfs\_remove\_file(kernel\_kobj, &etx\_attr.attr);

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 - Workqueue part 2");

MODULE\_VERSION("1.11");

## 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
* To trigger interrupt read device file (sudo cat /dev/etx\_device)
* Now see the Dmesg (dmesg)

*$ dmesg*

*[11213.943071] Major = 246 Minor = 0  
[11213.945181] Device Driver Insert…Done!!!  
[11217.255727] Device File Opened…!!!  
[11217.255747] Read function  
[11217.255783] Shared IRQ: Interrupt Occurred  
[11217.255845] Executing Workqueue Function  
[11217.255860] Device File Closed…!!!*

* We can able to see the print “**Shared IRQ: Interrupt Occurred**“ and “**Executing Workqueue Function**“
* Unload the module using sudo rmmod driver

In our [next tutorial](https://www.embetronicx.com/tutorials/linux/device-drivers/work-queue-in-linux-own-workqueue/) we will discuss Workqueue using own worker thread.