# Part 2 – First Linux Device Driver

[ <https://embetronicx.com/tutorials/linux/device-drivers/linux-device-driver-tutorial-part-2-first-device-driver/> ]

Before writing driver, we should give the module information. So First we will see about those module information.

**Module Information**

* License
* Author
* Module Description
* Module Version

All these information are present in the linux/module.h as a macros.

## License

GPL, or the GNU General Public License, is an open source license meant for software. If your software is licensed under the terms of the GPL, it is free. However, “free” here does not essentially mean freeware—it can also be a paid software. Instead, “free” as per the GPL means freedom. As proponents of GPL proudly proclaim, free as in freedom, not free beer.

The following license idents are currently accepted as indicating free software modules.

"GPL" [GNU Public License v2 or later]

"GPL v2" [GNU Public License v2]

"GPL and additional rights" [GNU Public License v2 rights and more]

"Dual BSD/GPL" [GNU Public License v2 or BSD license choice]

"Dual MIT/GPL" [GNU Public License v2 or MIT license choice]

"Dual MPL/GPL" [GNU Public License v2 or Mozilla license choice]

The following other idents are available

"Proprietary" [Non free products]

There are dual licensed components, but when running with Linux it is the GPL that is relevant so this is a non-issue. Similarly LGPL linked with GPL is a GPL combined work.

This exists for several reasons,

1. modinfo can show license info for users wanting to vet their setup is free
2. The community can ignore bug reports including proprietary modules
3. Vendors can do likewise based on their own policies

We can give the License for our driver (module) like below. For this you need to include the Linux/module.h header file.

MODULE\_LICENSE("GPL");

MODULE\_LICENSE("GPL v2");

MODULE\_LICENSE("Dual BSD/GPL");

## Author

Using this Macro we can mention that who is wrote this driver or module. So modinfo can show author name for users wanting to know. We can give the Author name for our driver (module) like below. For this you need to include the Linux/module.h header file.

MODULE\_AUTHOR("Author");

Note: Use “Name <email>” or just “Name”, for multiple authors use multiple MODULE\_AUTHOR() statements/lines.

## Module Description

Using this Macro we can give the description of the module or driver. So modinfo can show module description for users wanting to know. We can give the description for our driver (module) like below. For this you need to include the linux/module.h header file.

MODULE\_DESCRIPTION("A sample driver");

## Module Version

Using this Macro we can give the version of the module or driver. So modinfo can show module version for users wanting to know.

Version of form [<epoch>:]<version>[-<extra-version>].

<epoch>: A (small) unsigned integer which allows you to start versions anew. If not mentioned, it’s zero. eg. “2:1.0” is after “1:2.0”.

<version>: The <version> may contain only alphanumerics and the character `.’. Ordered by numeric sort for numeric parts, ascii sort for ascii parts (as per RPM or DEB algorithm).

<extraversion>: Like <version>, but inserted for local customizations, eg “rh3” or “rusty1”.

**Example**

MODULE\_VERSION("2:1.0");

# Simple Kernel Module Programming

So as of now we know the very basic things that needed for writing driver. Now we will move into programming. In every programming language, how we will start to write the code? Any ideas? Well, in all programming there would be a starting point and ending point. If you take C Language, starting point would be the main function, Isn’t it? It will start from the starting of the main function and run through the functions which is calling from main function. Finally it exits at the main function closing point. But Here two separate functions used for that starting and ending.

1. Init function
2. Exit function

Kernel modules require a different set of header files than user programs require.And keep in mind, Module code should not invoke user space Libraries or API’s or System calls.

## Init function

This is the function which will execute first when the driver is loaded into the kernel.

For example when we load the driver using **insmod**, this function will execute. Please see below to know the syntax of this function.

static int \_\_init hello\_world\_init(void) /\* Constructor \*/

{

    return 0;

}

module\_init(hello\_world\_init);

This function should register itself by using module\_init() macro.

## Exit function

This is the function which will execute last when the driver is unloaded from the kernel.

For example when we unload the driver using **rmmod**, this function will execute. Please see below to know the syntax of this function.

void \_\_exit hello\_world\_exit(void)

{

}

module\_exit(hello\_world\_exit);

This function should register itself by using module\_exit() macro.

## Printk()

In C programming how we will print the values or whatever? Correct. Using printf() function. printf() is a user space function. So we cant use this here. So they created one another function for kernel which is printk().

One of the differences is that printk lets you classify messages according to their severity by associating different loglevels, or priorities, with the messages. You usually indicate the loglevel with a macro. I will explain about the macros now. There are several macros used for printk.

**KERN\_EMERG:**

Used for emergency messages, usually those that precede a crash.

**KERN\_ALERT:**

Situation requiring immediate action.

**KERN\_CRIT:**

Critical conditions, often related to serious hardware or software failures.

**KERN\_ERR:**

Used to report error conditions; device drivers often use KERN\_ERR to report hardware difficulties.

**KERN\_WARNING:**

Warnings about problematic situations that do not, in themselves, create serious problems with the system.

**KERN\_NOTICE:**

Situations that are normal, but still worthy of note. A number of security-related conditions are reported at this level.

**KERN\_INFO:**

Informational messages. Many drivers print information about the hardware they find at startup time at this level.

**KERN\_DEBUG:**

Used for debugging messages.

**Example**

printk(KERN\_INFO "Welcome To EmbeTronicX");

## Difference between printf and printk

* Printk() is a kernel level function, which has the ability to print out to different loglevels as defined in . We can see the prints using dmesg command.
* printf() will always print to a file descriptor – STD\_OUT. We can see the prints in STD\_OUT console.

## Simple Driver

This is the complete code for our simple device driver (hello\_world\_module.c).

#include<linux/kernel.h>

#include<linux/init.h>

#include<linux/module.h>

static int \_\_init hello\_world\_init(void)

{

    printk(KERN\_INFO "Welcome to EmbeTronicX\n");

        printk(KERN\_INFO "This is the Simple Module\n");

        printk(KERN\_INFO "Kernel Module Inserted Successfully...\n");

    return 0;

}

void \_\_exit hello\_world\_exit(void)

{

    printk(KERN\_INFO "Kernel Module Removed Successfully...\n");

}

module\_init(hello\_world\_init);

module\_exit(hello\_world\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>");

MODULE\_DESCRIPTION("A simple hello world driver");

MODULE\_VERSION("2:1.0");

# Compiling our driver

Once we have the C code, it is time to compile it and create the module file hello\_world\_module.ko.

Creating a **Makefile** for your module is straightforward.

obj-m += hello\_world\_module.o

KDIR = /lib/modules/$(shell uname -r)/build

all:

    make -C $(KDIR)  M=$(shell pwd) modules

clean:

    make -C $(KDIR)  M=$(shell pwd) clean

With the C code (hello\_world\_module.c) and Makefile ready, all we need to do is invoke make to build our first driver (hello\_world\_module .ko).

In Terminal you need to enter make like below.

$ make

Now we got hello\_world\_module.ko. This is the kernel object which is loading into kernel.

# Loading and Unloading the Device driver

A Kernel Module is a small file that may be loaded into the running Kernel, and unloaded.

A Kernel Module is a small file that may be loaded into the running Kernel, and unloaded.

## Loading

To load a Kernel Module, use the insmod command with root privileges.

For example our module file name is hello\_world\_module.ko

$ sudo insmod hello\_world\_module.ko

lsmod used to see the modules were inserted. In below image, i’ve shown the prints in init function. Use dmesg to see the kernel prints.

So when i load the module, it executes the init function.

## Listing the Modules

In order to see the list of currently loaded Modules, use the lsmod command.

## Unloading

To un-load a Kernel Module, use the rmmod command with root privileges.

In our case,

$ sudo rmmod hello\_world\_module.ko

or

$ sudo rmmod hello\_world\_module

So when i unload the module, it executes the exit function.

## Getting Module Details

In order to get information about a Module (author, supported options), we may use the modinfo command.

For example

$ modinfo hello\_world\_module.ko