EmbeTronicX
Embedded Tutorials Zone

Sidebar

Device Drivers

Linux Device Driver Tutorial Part 2 – First Linux Device Driver

This is the Series on Linux Device Driver. The aim of this series is to provide easy and practical examples that anyone can understand. Now we are going to see Linux Device Driver Tutorial Part 2 – First Linux Device Driver. Before writing the driver, we should give the module information. So First we will see that module information.

4

Post Contents [hide]

Linux Device Driver Tutorial Part 2 – First Linux Device Driver

Module Information

- License
- Author
- Module Description
- Module Version

These pieces of information are present in the `Linux/module.h` as 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.

```
1 MODULE_LICENSE("GPL");
2 MODULE_LICENSE("GPL v2");
3 MODULE_LICENSE("Dual BSD/GPL");
```

Note: It is not strictly necessary, but your module really should specify which license applies to its code.

4 Author

Using this Macro we can mention that who is writing this driver or module. So **modinfo** can show the 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.

```
1 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 a description of the module or driver. So **modinfo** can show a 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.

```
1 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

```
1 MODULE_VERSION("2:1.0");
```

4

Simple Kernel Module Programming

So as of now, we know the very basic things that needed for writing drivers. 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, the 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 are calling from the 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 that 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.

```
1 static int __init hello_world_init(void) /* Constructor */
2 {
3     return 0;
4 }
5 module_init(hello_world_init);
```

This function should register itself by using module_init() macro.

Exit function

This is the function that 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.

```
1 void __exit hello_world_exit(void)
2 {
3
4 }
5 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 the kernel which is **printk().**

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

KERN_EMERG:

4

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

KERN_ALERT:

A 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

```
4
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 log levels 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 the STD_OUT console.

Simple Driver

This is the complete code for our simple device driver (hello_world_module.c). You can download the Project By clicking the below link.

Click Here For Code

```
#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 `sudo make` like the below image.

```
[redacted]:/home/driver/simple_driver$ ls -l
total 8
-rwxrwxrwx 1 root root 649 Aug 10 18:24 hello_world_module.c
-rwxrwxrwx 1 root root 169 Aug 10 18:24 Makefile
[redacted]:/home/driver/simple_driver$ sudo make
make -C /lib/modules/4.4.0-59-generic/build  M=/home/driver/simple_driver modules
make[1]: Entering directory `/usr/src/linux-headers-4.4.0-59-generic'
  CC [M]  /home/driver/simple_driver/hello_world_module.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/driver/simple_driver/hello_world_module.mod.o
  LD [M]  /home/driver/simple_driver/hello_world_module.ko
make[1]: Leaving directory `/usr/src/linux-headers-4.4.0-59-generic'
[redacted]:/home/driver/simple_driver$ ls -l
total 32
www.embetronicx.com
-rwxrwxrwx 1 root root  649 Aug 10 18:24 hello_world_module.c
-rw-r--r-- 1 root root 4444 Aug 10 18:32 hello_world_module.ko
-rw-r--r-- 1 root root  713 Aug 10 18:32 hello_world_module.mod.c
-rw-r--r-- 1 root root 2816 Aug 10 18:32 hello_world_module.mod.o
-rw-r--r-- 1 root root 2496 Aug 10 18:32 hello_world_module.o
-rwxrwxrwx 1 root root  169 Aug 10 18:24 Makefile
-rw-r--r-- 1 root root   56 Aug 10 18:32 modules.order
-rw-r--r-- 1 root root    0 Aug 10 18:32 Module.symvers
[redacted]:/home/driver/simple_driver$
```

Now we got `hello_world_module .ko`. This is the kernel object which is loading into the kernel.

Loading and Unloading the Device driver

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

Loading

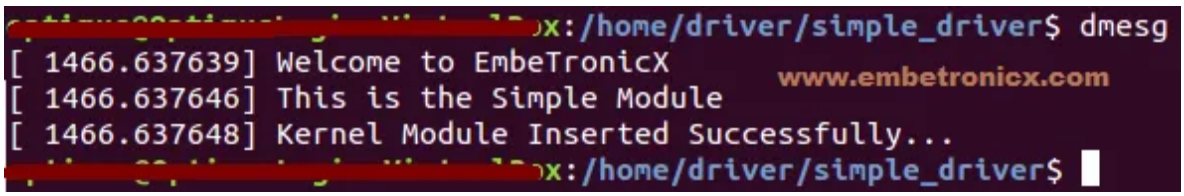
To load a Kernel Module, use the **`insmod`** command with root privileges.

For example, our module file name is hello_world_module.ko

```
sudo insmod hello_world_module.ko
```

```
[redacted]x:/home/driver/simple_driver$ sudo insmod hello_world_module.ko
[redacted]x:/home/driver/simple_driver$ lsmod
Module                  Size  Used by
hello_world_module     16384  0
www.embetronicx.com
nls_utf8               16384  1
isofs                  40960  1
pci_stub               16384  1
vboxpci                24576  0
```

`lsmod` used to see the modules were inserted. In the 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

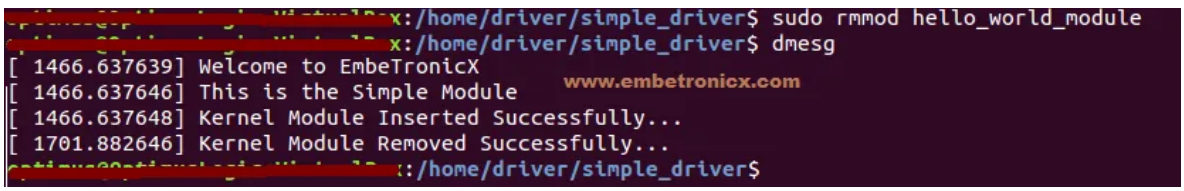
In order to see the list of currently loaded modules, use the `lsmod` command. In the above image, you can see that I have used `lsmod` command.

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`



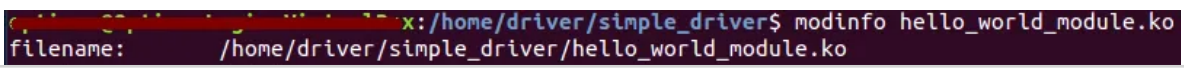
So when I unload the module, it executes the exit function.

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
```



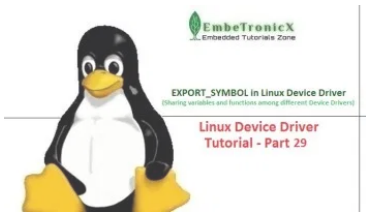
```
description:    A simple hello world driver
author:         EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>
license:        GPL
srcversion:     A0D3444D86F187EAB75B977
depends:
vermagic:       4.4.0-59-generic SMP mod_unload modversions
                x:/home/driver/simple_driver$
```


So now we know where to start to write the Linux device driver. Thank you for reading :-).

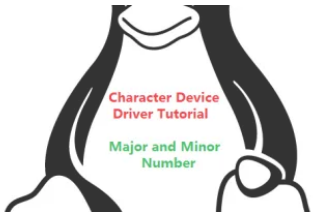
4

Article Rating

Related



Linux Device Driver Tutorial Part 29 – EXPORT_SYMBOL in Linux Device Driver
In "Device Drivers"



Linux Device Driver Tutorial Part 4 – Character Device Driver
In "Device Drivers"

Linux Device Driver Part 1 : Introduction
In "Device Drivers"

Website

POST COMMENT

This site uses Akismet to reduce spam. Learn how your comment data is processed.

4 COMMENTS Oldest

vikram

January 30, 2019 9:50 AM

while loading the module i am get this error.

insmod: ERROR: could not insert module hello_world_module.ko: Required key not available

Loading...

0 Reply

EmbeTronicx India

Reply to vikram January 31, 2019 2:01 AM

Please see https://askubuntu.com/questions/762254/why-do-i-get-required-key-not-available-when-install-3rd-party-kernel-modulesfor your issue. It is not the problem with our kernel driver. It might be the issue with your linux OS.

Loading...

0 Reply

4

vikram

January 30, 2019 9:50 AM

while loading i got this error

insmod: ERROR: could not insert module hello_world_module.ko: Required key not available

Loading...

0 Reply

Kalpesh

August 13, 2019 6:38 AM

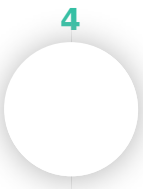
For the make file why to use .o format instead of .c

obj-m += hello_world_module.o

Loading...

0 Reply

4


4