Linux Kernel Module Programming: Hello World Program

Kernel modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. Custom codes can be added to Linux kernels via two methods.

- The basic way is to add the code to the kernel source tree and recompile the kernel.
- A more efficient way is to do this is by adding code to the kernel while it is running. This process is called loading the module, where module refers to the code that we want to add to the kernel.

Since we are loading these codes at runtime and they are not part of the official Linux kernel, these are called loadable kernel module(LKM), which is different from the “base kernel”. Base kernel is located in /boot directory and is always loaded when we boot our machine whereas LKMs are loaded after the base kernel is already loaded. Nonetheless, these LKM are very much part of our kernel and they communicate with base kernel to complete their functions.

LKMs can perform a variety of task, but basically they come under three main categories,

- device driver,
- filesystem driver and
- System calls.

So what advantage do LKMs offer?
One major advantage they have is that we don’t need to keep rebuilding the kernel every time we add a new device or if we upgrade an old device. This saves time and also helps in keeping our base kernel error free. A useful rule of thumb is that we should not change our base kernel once we have a working base kernel.
Also, it helps in diagnosing system problems. For example, assume we have added a module to the base kernel(i.e., we have modified our base kernel by recompiling it) and the module has a bug in it. This will cause error in system boot and we will never know which part of the kernel is causing problems. Whereas if we load the module at runtime and it causes problems, we will immediately know the issue and we can unload the module until we fix it.
LKMs are very flexible, in the sense that they can be loaded and unloaded with a single line of command. This helps in saving memory as we load the LKM only when we need them. Moreover, they are not slower than base kernel because calling either one of them is simply loading code from a different part of memory.

****Warning: LKMs are not user space programs. They are part of the kernel. They have free run of the system and can easily crash it.**

So now that we have established the use loadable kernel modules, we are going to write a hello world kernel module. That will print a message when we load the module and an exit message when we unload the module.

```
#include <linux/module.h>     /* Needed by all modules */
#include <linux/kernel.h>     /* Needed for KERN_INFO */
#include <linux/init.h>       /* Needed for the macros */

///< The license type -- this affects runtime behavior
MODULE_LICENSE("GPL");

///< The author -- visible when you use modinfo
MODULE_AUTHOR("Akshat Sinha");

///< The description -- see modinfo
MODULE_DESCRIPTION("A simple Hello world LKM!");

///< The version of the module
MODULE_VERSION("0.1");

static int __init hello_start(void)
{
    printk(KERN_INFO "Loading hello module...\n");
    printk(KERN_INFO "Hello world\n");
    return 0;
}

static void __exit hello_end(void)
{
    printk(KERN_INFO "Goodbye Mr.\n");
}

module_init(hello_start);
module_exit(hello_end);
```

Explanation for the above code:
Kernel modules must have at least two functions: a “start” (initialization) function called init_module() which is called when the module is insmoded into the kernel, and an “end” (cleanup) function called cleanup_module() which is called just before it is rmmoded. Actually, things have changed starting with kernel 2.3.13. You can now use whatever name you like for the start and end functions of a module. In fact, the new method is the preferred method. However, many people still use init_module() and cleanup_module() for their start and end functions. In this code we have used hello_start() as init function and hello_end() as cleanup function.
Another thing that you might have noticed is that instead of printf() function we have used printk(). This is because module will not print anything on the console but it will log the

message in /var/log/kern.log. Hence it is used to debug kernel modules. Moreover, there are eight possible loglevel strings, defined in the header , that are required while using printk(). We have list them in order of decreasing severity:

- KERN_EMERG: 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.

We have used KERN_INFO to print the message.

Preparing the system to run the code:
The system must be prepared to build kernel code, and to do this you must have the Linux headers installed on your device. On a typical Linux desktop machine you can use your package manager to locate the correct package to install. For example, under 64-bit Debian you can use:

```
akshat@gfg:~$ sudo apt-get install build-essential linux-headers-$(uname -r)
```

Makefile to compile the source code:

```
obj-m = hello.o
all:
	make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

****Note: Don’t forget the tab spaces in Makefile**

Compiling and loading the module:
Run the make command to compile the source code. Then use insmod to load the module.

```
akshat@gfg:~$ make
make -C /lib/modules/4.2.0-42-generic/build/ M=/home/akshat/Documents/hello-module modules
make[1]: Entering directory `/usr/src/linux-headers-4.2.0-42-generic'
```

```
  CC [M]  /home/akshat/Documents/hello-module/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/akshat/Documents/hello-module/hello.mod.o
  LD [M]  /home/akshat/Documents/hello-module/hello.ko
make[1]: Leaving directory `/usr/src/linux-headers-4.2.0-42-generic'
```

Now we will use insmod to load the hello.ko object.

```
akshat@gfg:~$ sudo insmod hello.ko
```

Testing the module:

You can get information about the module using the modinfo command, which will identify the description, author and any module parameters that are defined:

```
akshat@gfg:~$ modinfo hello.ko
filename:       /home/akshat/Documents/hello-module/hello.ko
version:        0.1
description:    A simple Hello world LKM
author:         Akshat Sinha
license:        GPL
srcversion:     2F2B1B95DA1F08AC18B09BC
depends:
vermagic:       4.2.0-42-generic SMP mod_unload modversions
```

To see the message, we need to read the kern.log in /var/log directory.

```
akshat@gfg:~$ tail /var/log/kern.log
...
...
Sep 10 17:43:39 akshat-gfg kernel: [26380.327886] Hello world
```

To unload the module, we run rmmod:

```
akshat@gfg:~$ sudo rmmod hello
```

Now run the tail command to get the exit message.

```
akshat@gfg:~$ tail /var/log/kern.log
...
Sep 10 17:43:39 akshat-gfg kernel: [26380.327886] Hello world
Sep 10 17:45:42 akshat-gfg kernel: [26503.773982] Goodbye Mr.
```