Sam Detor

Lab 1

Part 1

I followed this tutorial on how to write and insert a basic module into the kernel. I first installed the required packages needed for kernel development. I then learned that you need to write your kernel init and exit functions and tell the kernel which functions to call when loading or unloading the kernel using the module_init and module_exit functions. The init and exit functions simply used printk() to print "Hello World" and "Goodbye World" respectively. The most difficult part was getting the module to compile, as I didn't realize that it wouldn't compile unless module documentation is added. After I compiled and successfully inserted the module into the kernel, the kernel reported that it was "tainted". Upon further research I found that it is just a warning message saying non-vendor approved software had been inserted into the kernel module.

Source code used in part 1:

I used the following source code from Rober Oliver's "Writing a Simple Kernel Module" tutorial:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE(“GPL”);
MODULE_AUTHOR(“Robert W. Oliver II”);
MODULE_DESCRIPTION(“A simple example Linux module.”);
MODULE_VERSION(“0.01”);

static int __init lkm_example_init(void) {
 printk(KERN_INFO “Hello, World!\n”);
 return 0;
}

static void __exit lkm_example_exit(void) {
 printk(KERN_INFO “Goodbye, World!\n”);
}

module_init(lkm_example_init);
module_exit(lkm_example_exit);
```

I changed the name of the init and exit functions from lkm_example_init and lkm_example_exit to helloWord and goodbyeWorld. I also changed the module description, author, and version.
I also used the makefile from the same tutorial:

```
obj-m += lkm_example.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
```

I only changed the name of the module from lkm_example to hello.

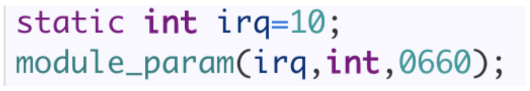
Part 2

At first, I thought we were supposed to create a sysfs file at the location of /sys/module/<module_name>/parameters/<parameter_name> using k_objects. Once I realized that we were just supposed to use module parameters, the problem became simpler. To implement the enable logging parameter, I followed another online tutorial to learn how to use the module_param macro. I created an integer parameter called enable_logging and made each printk() in the module code statement dependent on the value of enable_logging. After this, I ran into a permission issue. The sudo echo 0 > /sys/module/hello/parameters/enable_logging was not able to actually edit the parameter. I tried setting the write and read privileges for the module to global, but the module wouldn't compile. I then learned that you are not allowed to set the write privileges to a kernel module to global. I then found that the "sudo" in the previous command was only "working" on the echo and the redirection did not get sudo privileges. I then used sudo su to act as the root and set the module parameters to user read and write, and the tests worked as expected.

For the double_me parameter, I modified some code from an online tutorial on how to communicate with device drivers. The most difficult part of this assignment was finding the correct macro—module_param_cb—and online tutorial to use as moduleparam.h was difficult to understand. From the tutorial, I learned that module_param_cb macro takes an argument of a kernel_param_ops struct. This tells the macro what methods to call when the parameter is set or accessed by the user. So, I created a method that would be called when the parameter was set by the user, which checked if the parameter value given by the user was an integer —using the macro param_set_int—and then doubled the value set. This method was then put in the kernel_param_ops struct as the .set method and the standard param_get_int macro was used for the .get method in the struct. The rest of the set up for the double_me module parameter was similar to the enable_logging parameter.

<u>Source code used in part 2:</u>

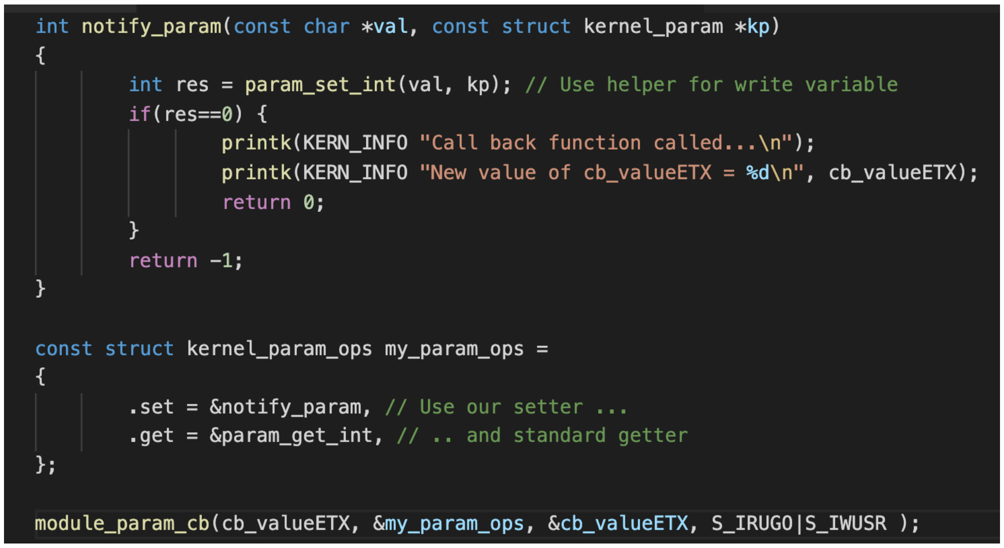
For the enable logging parameter, I used code from Lirhan B.H.'s blog post on kernel module parameters. I used and modified the following two lines from the post:



```
static int irq=10;
module_param(irq,int,0660);
```

I changed the name from irq to enable_logging and changed the permissions from 0660 to S_IRUGO | S_IRUSR.

For the double_me parameter I used and modified some of the following code from an EmbeTronicX tutorial on passing arguments to device drivers.



```
int notify_param(const char *val, const struct kernel_param *kp)
{
        int res = param_set_int(val, kp); // Use helper for write variable
        if(res==0) {
                printk(KERN_INFO "Call back function called...\n");
                printk(KERN_INFO "New value of cb_valueETX = %d\n", cb_valueETX);
                return 0;
        }
        return -1;
}

const struct kernel_param_ops my_param_ops =
{
        .set = &notify_param, // Use our setter ...
        .get = &param_get_int, // .. and standard getter
};

module_param_cb(cb_valueETX, &my_param_ops, &cb_valueETX, S_IRUGO|S_IWUSR );
```

I changed the name of the parameter, kernel_param_ops struct, and the .set method to double_me, double_me_ops, and double_val respectively. In my version of the notify_param method, I doubled the double_me variable and printed a message to the kernel log instead. I also set my permissions to user read and write instead of global read and user write. Lastly, I returned the recommended EINVAL instead of -1 if the input given by the user was not an integer.

My part 2 code is also a modified version of my part 1 code, and sources for my part 1 code can be found in the part 1 section of this lab report.