

Assignment 6 – Device Driver

***NOTE: My references are on the final page of this document where I've cited my sources as well as explained what I referenced and from which source(s). Also, instead of having the "Approach/What I Did" and "Issues and Resolutions" sections come first, I want to begin the writeup by providing the instructions for how to install my driver, followed by explanations for how the module and test applications work, and then the "Approach", "Issues", and "Screenshots" sections last. This is just to make things flow better.

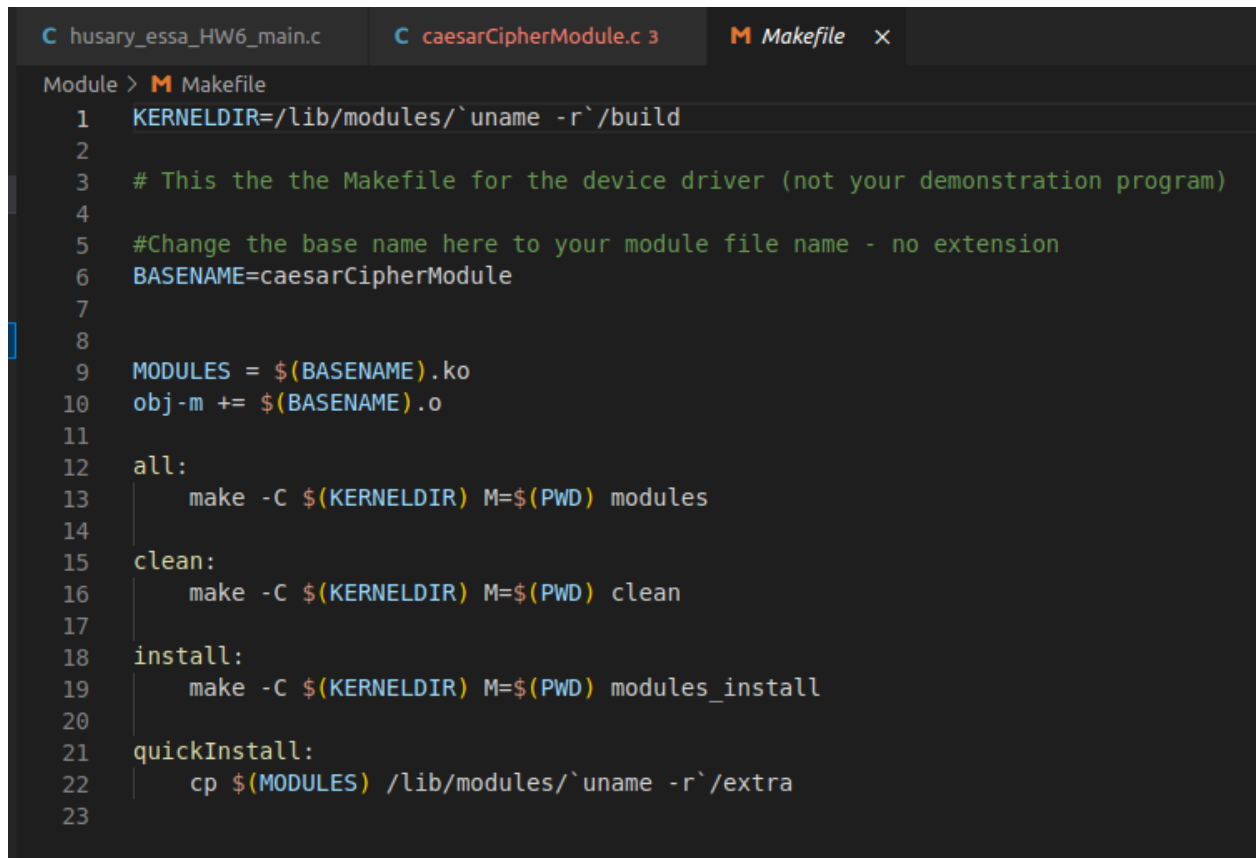
Description:

In this assignment, we are writing our own device drivers. We are tasked with writing the Module for our device drivers to allow a user to open, close, read, write, and utilize ioctl to test and run our device drivers. We are also tasked with writing the associated Test/user application which tests our device drivers. The purpose of this assignment is to acclimate us to device drivers as well as to acclimate us to coding in kernel space.

The device driver that I wrote is a Caesar Cipher. What it does is allow the user to write a message and choose to either encrypt that message, decrypt it, or even set the key that will be utilized in the encryption and decryption operations. The module allows a user to open up the device file via myOpen(), allows a user to use myWrite() to write their message into a buffer stored in a data structure called "cipherDataStructure", allows them to call myIoctl() to issue a command, allows them to read the returned message from a buffer from that same data structure via myRead(), and allows them to close the device file via myClose(). The test application, in order to allow the user to test the device driver, utilizes a 'while' loop to continuously ask the user for a command and then asks them for the respective input.

Instructions on how to build, load, install, and even uninstall and unload my kernel module:

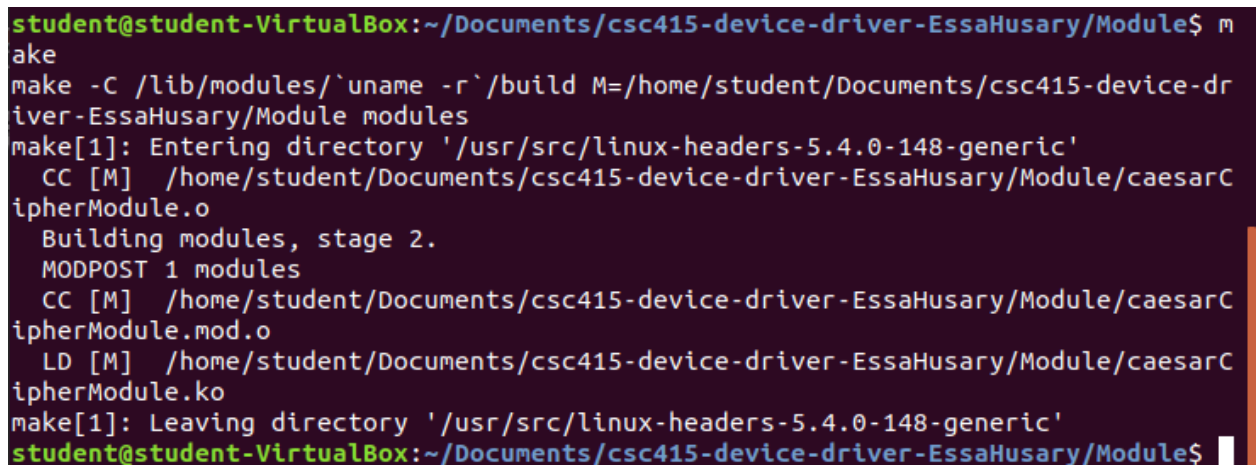
- 1) The first thing that must be done is to change the "BASENAME" in the Makefile of the Module folder. Open the Makefile, and in the 6th line which reads "BASENAME=", type in "caesarCipherModule" as shown here:



```
Module > Makefile
1  KERNELDIR=/lib/modules/`uname -r`/build
2
3  # This the the Makefile for the device driver (not your demonstration program)
4
5  #Change the base name here to your module file name - no extension
6  BASENAME=caesarCipherModule
7
8
9  MODULES = $(BASENAME).ko
10 obj-m += $(BASENAME).o
11
12 all:
13     make -C $(KERNELDIR) M=$(PWD) modules
14
15 clean:
16     make -C $(KERNELDIR) M=$(PWD) clean
17
18 install:
19     make -C $(KERNELDIR) M=$(PWD) modules_install
20
21 quickInstall:
22     cp $(MODULES) /lib/modules/`uname -r`/extra
23
```

Now of course, with the changes I pushed to GitHub, this step is already done. However, this is the first step all of us had to do to get the kernel object to eventually be created so I feel that it is necessary to mention this very important step.

- 2) Next, in the terminal, run the command “make”. Once this is done, the kernel object has successfully been created as evidenced by the compilation:



```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ make
make -C /lib/modules/`uname -r`/build M=/home/student/Documents/csc415-device-driver-EssaHusary/Module modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-148-generic'
  CC [M] /home/student/Documents/csc415-device-driver-EssaHusary/Module/caesarCipherModule.o
Building modules, stage 2.
MODPOST 1 modules
  CC [M] /home/student/Documents/csc415-device-driver-EssaHusary/Module/caesarCipherModule.mod.o
  LD [M] /home/student/Documents/csc415-device-driver-EssaHusary/Module/caesarCipherModule.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-148-generic'
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

3) Next, we begin installing the newly created kernel object by running three commands,

- a. `sudo insmod caesarCipherModule.ko`
- b. `sudo mknod /dev/CaesarCipher c 333 3`
- c. `sudo chmod 666 /dev/CaesarCipher`

in that order. According to the lecture and the man pages, the first command inserts a module into the kernel, the second command creates the device file, which is a special file, and the third command sets file permissions. If successful, you will get the following messages by running “tail /var/log/kern.log” in the command line (the messages are located towards the end):

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ tail /var/log/kern.log
May 12 23:48:41 student-VirtualBox kernel: [ 16.949498] 06:48:41.329783 main OS Release
: 5.4.0-148-generic
May 12 23:48:41 student-VirtualBox kernel: [ 16.949789] 06:48:41.330097 main OS Version
: #165~18.04.1-Ubuntu SMP Thu Apr 20 01:14:06 UTC 2023
May 12 23:48:41 student-VirtualBox kernel: [ 16.950153] 06:48:41.330439 main Executable
: /opt/VBoxGuestAdditions-6.1.2/sbin/VBoxService
May 12 23:48:41 student-VirtualBox kernel: [ 16.950153] 06:48:41.330440 main Process ID
: 1087
May 12 23:48:41 student-VirtualBox kernel: [ 16.950153] 06:48:41.330441 main Package ty
pe: LINUX_64BITS_GENERIC
May 12 23:48:41 student-VirtualBox kernel: [ 16.953177] 06:48:41.333474 main 6.1.2 r135
662 started. Verbose level = 0
May 12 23:48:41 student-VirtualBox kernel: [ 16.955349] 06:48:41.335641 main vbglR3Gues
tCtrlDetectPeekGetCancelSupport: Supported (#1)
May 12 23:48:59 student-VirtualBox kernel: [ 35.259065] rfkill: input handler disabled
May 13 01:25:08 student-VirtualBox kernel: [ 5801.139501] Device registration successful: 0
May 13 01:25:08 student-VirtualBox kernel: [ 5801.139504] Succeeded adding the new chardev! 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

Here's a closer look at what I mean:

```
May 13 01:25:08 student-VirtualBox kernel: [ 5801.139501] Device registration successful: 0
May 13 01:25:08 student-VirtualBox kernel: [ 5801.139504] Succeeded adding the new chardev! 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

These messages are printed when the kernel object is inserted into the kernel as seen in my `init_module` function:

```
// To begin initializing and registering the device driver to the kernel
int init_module(void){

    // To store the return value of cdev_add
    int returnVal = 0;
    // To store the return value of the chardev registration
    int registrationNum = 0;
    // To store the result of MKDEV
    dev_t devNumber;

    // To make the device driver
    devNumber = MKDEV(MAJOR_VERSION_NO, MINOR_VERSION_NO);

    // To register the newly made driver
    registrationNum = register_chrdev_region(devNumber, 1, NAME_OF_DEVICE);

    // To print a success message to the kernel log
    printk(KERN_INFO "Device registration successful: %d", registrationNum);
    // To initialize the driver and it's file operations
    cdev_init(&new_cdev, &fops);
    new_cdev.owner = THIS_MODULE;

    // To add the new chardev
    returnVal = cdev_add(&new_cdev, devNumber, 1);
    printk(KERN_INFO "Succeeded adding the new chardev! %d", returnVal);
    printk(KERN_INFO "Welcome to my Caesar Cipher device driver!");

    // To check if the addition failed
    if (returnVal < 0){
        printk(KERN_ERR "Failed to add the chardev!%d\n", returnVal);
    }

    // To return the status of cdev_add
```

The kernel object has been successfully installed now and the special device file has also been created. To prove this, here's the screenshot of the device file highlighted when running "ls /dev/C*" in the command line:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ ls /dev/C*
/dev/CaesarCipher
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ ls -l /dev/C*
crw-rw-rw- 1 root root 333, 3 May 13 01:25 /dev/CaesarCipher
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

- 4) The kernel module has been successfully built and installed, and the device file has successfully been created. But what if we wanted to delete the device file and remove the kernel object from the kernel? Well, we can begin by running "make clean". Then

once we run this, we can remove the device file by running “sudo rm /dev/CaesarCipher” in the command line as shown here:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ sudo rm /dev/CaesarCipher
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

If we try to find this file, we get this message:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ ls -l /dev/C*
ls: cannot access '/dev/C*': No such file or directory
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

indicating that it has been removed. Now, we can remove the kernel object from the kernel by running “sudo rmmod caesarCipherModule.ko” in the command line as shown here:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ sudo rmmod caesarCipherModule.ko
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

We get this message from the kernel log indicating that the kernel object has now been unregistered/removed from the kernel:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ tail /var/log/kern.log
May 12 23:48:41 student-VirtualBox kernel: [ 16.950153] 06:48:41.330439 main Executable
: /opt/VBoxGuestAdditions-6.1.2/sbin/VBoxService
May 12 23:48:41 student-VirtualBox kernel: [ 16.950153] 06:48:41.330440 main Process ID
: 1087
May 12 23:48:41 student-VirtualBox kernel: [ 16.950153] 06:48:41.330441 main Package type: LINUX_64BITS_GENERIC
May 12 23:48:41 student-VirtualBox kernel: [ 16.953177] 06:48:41.333474 main 6.1.2 r135662 started. Verbose level = 0
May 12 23:48:41 student-VirtualBox kernel: [ 16.955349] 06:48:41.335641 main vbglR3GuestCtrlDetectPeekGetCancelSupport: Supported (#1)
May 12 23:48:59 student-VirtualBox kernel: [ 35.259065] rfkill: input handler disabled
May 13 01:25:08 student-VirtualBox kernel: [ 5801.139501] Device registration successful: 0
May 13 01:25:08 student-VirtualBox kernel: [ 5801.139504] Succeeded adding the new chardev! 0
May 13 01:42:36 student-VirtualBox kernel: [ 5801.139505] Welcome to my Caesar Cipher device driver!
May 13 01:42:36 student-VirtualBox kernel: [ 6848.709860] Thanks for using my device driver! Hope you had fun!
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

Here’s a closer look:

```
May 13 01:42:36 student-VirtualBox kernel: [ 6848.709860] Thanks for using my device driver! Hope you had fun!
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

This message is printed when the kernel object is removed from the kernel as seen in my cleanup_module function:

```
// To unregister the device driver
void cleanup_module(void){

    dev_t devNumber;
    devNumber = MKDEV(MAJOR_VERSION_NO, MINOR_VERSION_NO);

    // To unregister the device driver
    unregister_chrdev_region(devNumber, 1);

    // To delete the driver
    cdev_del(&new_cdev);

    printk(KERN_INFO "Thanks for using my device driver! Hope you had fun!\n");
}
```

**AN EXPLANATION AS TO HOW THE KERNEL MODULE WORKS IS ON THE NEXT
PAGE**

How the kernel module works:

I will begin by first defining certain terms, then discussing the loading/initialization of the module, then the unloading of the module, and then finally, I will explain my device driver's functionality and what it's about. To begin, at the very top of my module, I have what are called major and minor numbers. According to O'Reilly, in their article titled, "Major and Minor Numbers – Linux Device Drivers, Second Edition [Book]", a major number "...identifies the driver associated with the device" and a minor number allows a driver to distinguish between the various devices a driver may manage ("Major and Minor..." [TEXTNOTE]). They are shown here:

```
// My major version number
#define MAJOR_VERSION_NO 333
// My minor version number
#define MINOR_VERSION_NO 3
```

I also have up top "new_cdev", an instance of the cdev struct. According to O'Reilly, in another one of their articles titled, "Char Drivers – Linux Device Drivers, 3rd Edition [Book]" the cdev struct is a character device struct and is "...the kernel's internal structure that represents char devices..." ("Char Drivers..." [TEXTNOTE]). It is shown here:

```
struct cdev new_cdev;
```

Further, I have a struct called "file_operations". In this struct, I am assigning to the variables the addresses of my functions, myOpen(), myClose(), etc. This enables the user to perform "file functions" using my module functions! When they call read(), myRead() will instead be called. When they call write(), myWrite() will instead be called. This allows the user to use my device driver by calling those file functions. This struct is shown here:

```
// To reassign the linux file ops to our device driver's file functions
struct file_operations fops = {

    .open = myOpen,
    .release = myClose,
    .write = myWrite,
    .read = myRead,
    .unlocked_ioctl = myIoCtl,
    .owner = THIS_MODULE,

};
```

Now let's delve into "init_module". In "init_module", I am initializing, loading, and registering the module into the kernel. I begin by first calling MKDEV(). According to you Professor, MKDEV makes the device number from my major and minor numbers. I then register the device into the kernel as a character device by calling register_chrdev_region(), passing in as one of the parameters the newly generated device number, "devNumber". Next, I initialize the device driver by calling cdev_init(), passing in as parameters the address of "new_cdev" (the instance of the cdev struct) and the address of the file_operations struct. After this initialization, finally, I add the device driver into the system by calling cdev_add(). When I run, in the command line, the commands "make", "sudo insmod caesarCipherModule.ko", and "sudo mknod /dev/CaesarCipher c 333 3", this init_module() function is called, creating the kernel object and inserting it into the kernel (hence, insert module caesarCipherModule.ko). Here is my function:


```
// To begin initializing and registering the device driver to the kernel
int init_module(void){

    // To store the return value of cdev_add
    int returnVal = 0;
    // To store the return value of the chardev registration
    int registrationNum = 0;
    // To store the device driver number, the result of MKDEV
    dev_t devNumber;

    // To make the device driver number
    devNumber = MKDEV(MAJOR_VERSION_NO, MINOR_VERSION_NO);

    // To register the driver
    registrationNum = register_chrdev_region(devNumber, 1, NAME_OF_DEVICE);

    // To print a success message to the kernel log
    printk(KERN_INFO "Device registration successful: %d", registrationNum);
    // To initialize the driver with its file operations
    cdev_init(&new_cdev, &fops);
    new_cdev.owner = THIS_MODULE;

    // To add the new chardev into the system
    returnVal = cdev_add(&new_cdev, devNumber, 1);
    printk(KERN_INFO "Succeeded adding the new chardev! %d", returnVal);
    printk(KERN_INFO "Welcome to my Caesar Cipher device driver!");

    // To check if the addition failed
    if (returnVal < 0){
        printk(KERN_ERR "Failed to add the chardev!%d\n", returnVal);
    }

    // To return the status of cdev_add
    return returnVal;
}
```

The associated device file is also created in this process as well when the command “mknod” is run, with “CaesarCipher” in the dev folder (/dev/CaesarCipher), “c” indicating a chardev, and “333” and “3” indicating my major and minor numbers, respectively. This special file, which in my case is called “CaesarCipher”, is the bridge, so to speak, which allows the user to use my device driver. When they open my device file by calling open() in the user application, they are now accessing my device driver. They are then able to use it, calling functions such as myWrite, myRead, myIoCtl, etc., etc. This file can be thought of as the “golden ticket” to allow them to use my device driver.

Unloading the device is done in the function called “cleanup_module”. The function unregister_chrdev_region() is called, which unregisters the device driver from the kernel. The following function, cdev_del(), deletes it. When the commands “sudo rm /dev/CaesarCipher” and “sudo rmmod caesarCipherModule.ko” are run in the command line, this function is called, unregistering and deleting my device driver as well as the special device file. The cleanup_module() function is shown here:

```
// To unregister and remove the device driver
void cleanup_module(void){

    dev_t devNumber;
    devNumber = MKDEV(MAJOR_VERSION_NO, MINOR_VERSION_NO);

    // To unregister the device driver
    unregister_chrdev_region(devNumber, 1);

    // To delete the driver
    cdev_del(&new_cdev);

    printk(KERN_INFO " Thanks for using my device driver! Hope you had fun!\n");
}
```

Now that I have defined some necessary terms and explained how the module is loaded and unloaded along with their associated commands in the command line, I will now explain the functionality of my device driver. First, let’s look at my data structure aptly named “cipherDataStructure”:

```
/* A data structure to store the message the user wants either encrypted or decrypted, the returned message, "returnedMessage", which has either been encrypted or decrypted, the key for encryption/decryption, and the length of the providedMessage */
struct cipherDataStructure {

    // To store the user's message
    char providedMessage[MAX_MESSAGE_LENGTH];
    // To store the message that has now been either encrypted or decrypted
    char returnedMessage[MAX_MESSAGE_LENGTH];
    // To store the key to use for either the encryption or decryption
    int key;
    /* To store the length of the provided message. I have this in my data structure so that when we read back the encrypted/decrypted message, we only have the returned message and not the other data contained in the buffer. This will make more sense when you look at myRead() */
    int lengthOfProvidedMessage;
} cds;
```

The field “providedMessage” will store the message that the user wants either encrypted or decrypted. The “returnedMessage” field will store the user’s message that has now been either encrypted or decrypted. The “key” field will store the key that the user desires to use for the encryption and decryption operations. When the device file is opened, the “key” will be assigned a default value of 4. The final field, “lengthOfProvidedMessage” will store the length of the user’s message. The reason I have this is because if the user has already done either an encryption or decryption operation, the buffer “returnedMessage” will have some data stored in it. I want to make sure that when myRead() is called, we are only giving back to the user the appropriate encrypted or decrypted message rather than any mumbo jumbo artifacts from previous operations, which may happen if they want an excerpt from the Declaration of Independence encrypted, but next, want the message “Hello” encrypted; they would have “Hello” immediately followed by the excerpt (which is obviously not what they want). The length of the returned message is ONLY going to be as long as the provided message which is why I store the length of the provided message here.

Now that I have explained my data structure, I will now explain every function in my module. Let’s begin with myOpen(). In myOpen(), I am opening up the device file to begin operations. I vmalloc space for my data structure and initialize the “lengthOfProvidedMessage” field to 0, the “key” to a value of 4, well really, to the #define “DEFAULT_KEY_VALUE” value (which has a value of 4), and I make sure to have my data structure be pointed to by “private_data”, to maintain access to it. Next, we have myWrite(). In myWrite(), we set “lengthOfProvidedMessage” to be the length of the user’s message, and we copy the user’s message into the “providedMessage” field of my data structure. This will allow us to either

encrypt or decrypt that message. Next, we have `myIoctl()`. In general, `ioctl()` allows one to perform operations or issue commands that are specific to a certain device driver. Its parameters are a file descriptor, a command, and a third, optional parameter (in the case of my device driver, the third parameter is the key). The user is able to specify what operation they want performed via supported commands 3, 4, and 5. Command 3 allows the user to encrypt their message, command 4 allows them to decrypt their message, and command 5 allows them to set the key to their desired value. Setting the key is as simple as setting that “key” field from my data structure equal to the third parameter passed into `ioctl()` from the user/test application. For commands 3 and 4, we enter their respective ‘if’ statements to call either `encrypt()`, if the command is 3, or `decrypt()`, if the command is equal to 4. As you might guess, `encrypt()` encrypts the user’s message and `decrypt()` decrypts a user’s message. Both functions have the “providedMessage”, “returnedMessage”, and “key” passed in as parameters, and both functions call the `capitalize()` function to capitalize the “providedMessage”. I’ve chosen to capitalize the entire message because capital letters make the encryption and decryption operations easier while still maintaining the message, which is the most important thing to me here. Further, the function `myRead()` does two things. For one, it appends to the end of “returnedMessage” the null terminator using “lengthOfProvidedMessage” like so:

```
/* To append the null terminator to the end of the returned message as we only  
| want that returned message and not what's already contained in the buffer  
cds->returnedMessage[cds->lengthOfProvidedMessage] = '\0';
```

This is so that when we print the message out, we only get the returned message instead of any other data that may have been contained in “returnedMessage” from previous operations (an issue I discussed earlier). Secondly, and just as important, we want to give to the user, or read back, their message that has now been either encrypted or decrypted by copying “returnedMessage” into their buffer using `strcpy()`. Finally, `myClose()` closes the file by freeing the space allocated for my data structure using `vfree`.

Instructions on how to build and run my test/user application:

Building my test application is as simple as running the command “make” in the command line (“make run” also does the job):

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make
gcc -c -o husary_essa_HW6_main.o husary_essa_HW6_main.c -g -I.
gcc -o husary_essa_HW6_main husary_essa_HW6_main.o -g -I. -l pthread
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

Running it is as simple as running the command “make run”:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make
run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

How to use my test/user application and how it works:

I will begin by first explaining how to use my test application and therefore, my device driver, and then I will go into depth about how it works. To use it, all you need to do is run, in the command line, the command “make run”. Once that happens, you will be prompted for a command number of either 3, 4, 5, or 0 (or a non-number key). Typing in command 3 will allow you to encrypt a word and command 4 will allow you to decrypt a word. In both cases, you will then be asked to provide a message to be either encrypted or decrypted. Typing in command 5, however, will allow you to set the key to another value, in which case, you will be prompted for a new key value. And, typing in 0 or a non-number key will allow you to exit the program. It’s very simple and easy to use and was designed to be that way. Now to explain how my test application works in depth.

The way my test application works is quite simple. I first begin by calling `open()`, passing in the device file name as a parameter:

```
// To open the device file and to store the associated file descriptor  
fileDescriptor = open("/dev/CaesarCipher", O_RDWR);
```

As I stated in my explanation for the module, the device file is the bridge between the user application and the device driver so it's of utmost importance that we pass into `open()` the device file. And then once this occurs, and assuming the opening up of the device file is successful, the user enters into a 'while' loop where they are continuously prompted for input. First, they are asked for a command number:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make  
run  
./husary_essa_HW6_main  
  
Returned file descriptor value: 3  
Successfully opened the device file!  
  
Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is s  
etting the key (the default value of the key is 4). Type either 0 or a non-number  
to exit: █
```

As it shows above, command 3 allows the user to encrypt a message, command 4 allows the user to decrypt a message, and command 5 allows them to set the key to a value other than the default value of 4. A command number of 0 will exit the program and any entry that is not a number will be treated as a 0 thereby also causing the program to exit (**screenshots of these cases will provided in the "Screenshots of the execution" section**). Any number other than 3, 4, 5, and 0 will just cause the prompt to reappear, like so:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make
run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 1

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 3333

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: -221

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 
```

Once the user enters in say, commands 3 or 4, they enter this 'if' statement:

```
// Otherwise, to check if the user wants to encrypt or decrypt a message
} else if (commandNo == SUPPORTED_COMMAND_1 ||
           commandNo == SUPPORTED_COMMAND_2){

    // To prompt the user for a message
    printf("\nPlease enter a message you want encrypted or decrypted: ");
    // To get the message from the user
    retValueOfFgets = fgets(userBuffer, MAX_MESSAGE_LENGTH, stdin);
    /* To check if there was an error getting input (taken from MY assignment
    3 submission) */
    if (retValueOfFgets == NULL) {

        if (ferror(stdin)){

            printf("\nAn error has occurred while reading input." "Exiting.");
            clearerr(stdin);
            exit(errno);

        }

    }

    // To handle the input. (see my function before main() for more info)
    handleInput(userBuffer);
}
```

SUPPORTED_COMMAND_1 and SUPPORTED_COMMAND_2 are #defines with values 3 and 4, respectively. So when the condition is met, they are prompted to provide a message to be either encrypted or decrypted:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make
run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 3

Please enter a message you want encrypted or decrypted: 
```


The message they enter will be copied into “userBuffer”, seen below, which will be passed into write(), where the module’s myWrite() function will be called, copying the user’s buffer into the “providedMessage” field of my “cipherDataStructure” struct:

```
290
291 // To write the user's message into their buffer
292 retValueOfWrite = write(fileDescriptor, userBuffer, MAX_MESSAGE_LENGTH);
293
294 // To check if the write succeeded
295 if (retValueOfWrite < 0){
296
297     perror("Error providing input. Here are the details: ");
298     printf("Closing...\n");
299
300     return (-1);
301 }
```

A message longer than 1 MB will truncate the message (**a screenshot of this case will provided in the “Screenshots of the execution” section.** Next, ioctl() will be called where their desired command is passed in along with the key:

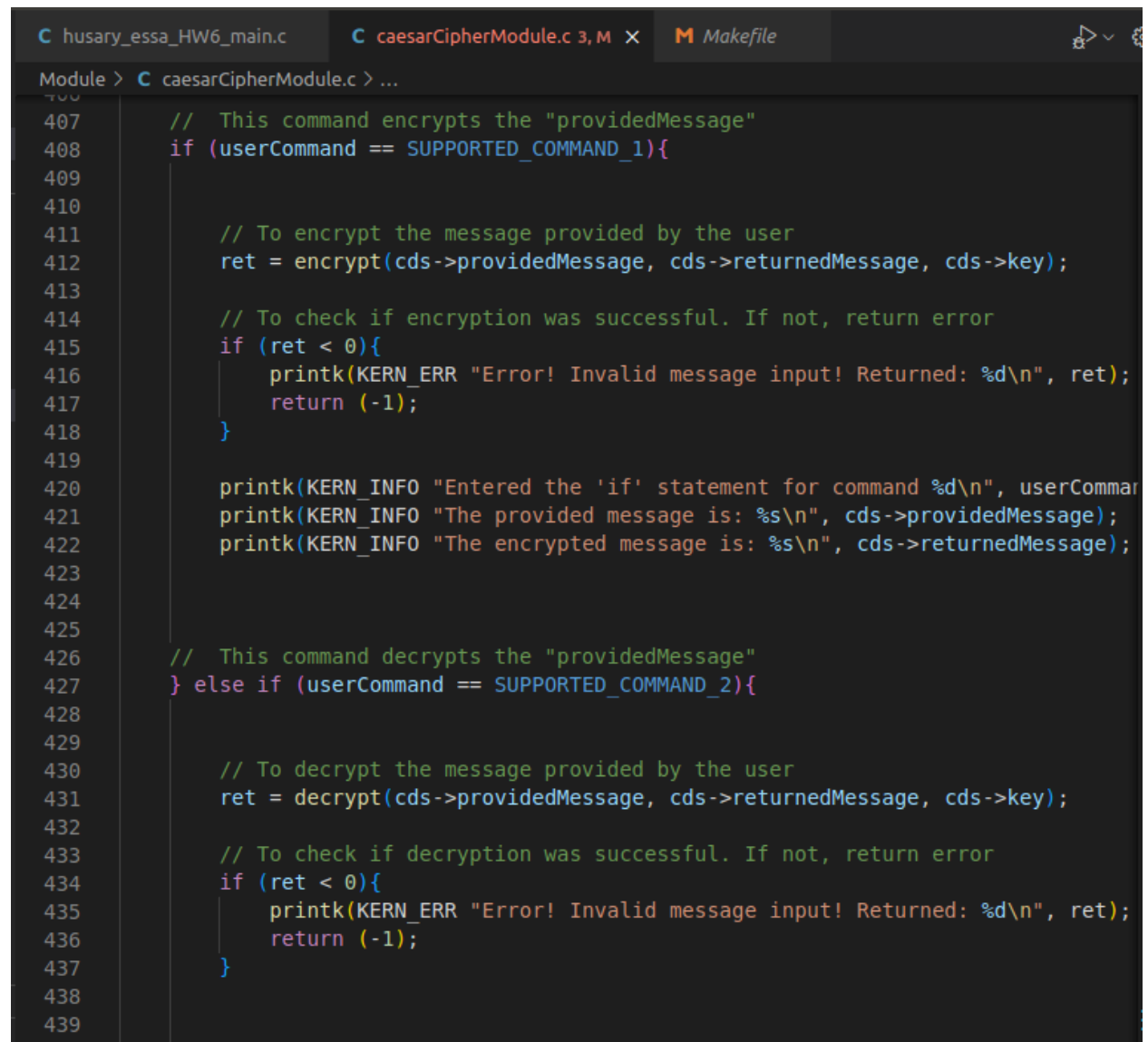
```
304
305 /* To process either the encryption or decryption the user desires
306    for their message */
307 retValueOfIOCTL = ioctl(fileDescriptor, commandNo, key);
308
309 // To check if the io control succeeded
310 if (retValueOfIOCTL < 0){
311
312     printf("Error! You entered invalid input!\n");
313     perror("Error details: ");
314     printf("\n\n");
315
316     continue;
317 } else {
318
319     printf("\nIOCTL operation successful!\n\n");
320
321 }
322
323 }
```

Note that the key will have a value of 0 in this case, but nothing happens in ioctl to set it to 0, unless the user enters in a command of 5. The key will only ever be set when commanded to in ioctl, meaning, it maintains its current value (maybe the default value, maybe something else) despite the fact that what’s being passed in is initially 0. With that being said, here in ioctl is where the magic happens. Depending on whether the command is a 3 or 4, the message will either be encrypted or decrypted, as can be seen in my myIoCtl() function in “caesarCipherModule”:

Here are the parameters:

```
373
374 // A function to allow the user to perform either encryption or decryption
375 static long myIoCtl(struct file *fs, unsigned int command, unsigned long desiredVa
376
377
```

And here are the 'if' statements:



The screenshot shows a code editor with three tabs: 'husary_essa_HW6_main.c', 'caesarCipherModule.c 3, M', and 'Makefile'. The active tab is 'caesarCipherModule.c 3, M'. The code is written in C and shows the implementation of encryption and decryption logic. The code is as follows:

```
Module > C caesarCipherModule.c > ...
407 // This command encrypts the "providedMessage"
408 if (userCommand == SUPPORTED_COMMAND_1){
409
410     // To encrypt the message provided by the user
411     ret = encrypt(cds->providedMessage, cds->returnedMessage, cds->key);
412
413     // To check if encryption was successful. If not, return error
414     if (ret < 0){
415         printk(KERN_ERR "Error! Invalid message input! Returned: %d\n", ret);
416         return (-1);
417     }
418
419     printk(KERN_INFO "Entered the 'if' statement for command %d\n", userCommand);
420     printk(KERN_INFO "The provided message is: %s\n", cds->providedMessage);
421     printk(KERN_INFO "The encrypted message is: %s\n", cds->returnedMessage);
422
423
424
425
426 // This command decrypts the "providedMessage"
427 } else if (userCommand == SUPPORTED_COMMAND_2){
428
429     // To decrypt the message provided by the user
430     ret = decrypt(cds->providedMessage, cds->returnedMessage, cds->key);
431
432     // To check if decryption was successful. If not, return error
433     if (ret < 0){
434         printk(KERN_ERR "Error! Invalid message input! Returned: %d\n", ret);
435         return (-1);
436     }
437
438
439     printk(KERN_INFO "Entered the 'if' statement for command %d\n", userCommand);
440     printk(KERN_INFO "The provided message is: %s\n", cds->providedMessage);
441     printk(KERN_INFO "The decrypted message is: %s\n", cds->returnedMessage);
442
443
444
445
```

Again, SUPPORTED_COMMAND_1 and SUPPORTED_COMMAND_2 are #defines with values 3 and 4, respectively. Next in the user application, the function read() is called, where the buffer to store the returned message is passed in and myRead() from the module is utilized:

```
325
326     /* To read the user's now encrypted/decrypted message. Alternatively,
327     think of this as "writing" the returned message into the new buffer
328     retValueOfRead = read(fileDescriptor, newBuffer, MAX_MESSAGE_LENGTH);
329
330     // To check if the read succeeded
331     if (retValueOfRead < 0){
332
333         perror("\nError reading input. Here are the details: ");
334         printf("Closing...\n\n");
335
336         return (-1);
337     }
338
339     printf("Return value of read: %d\n", retValueOfRead);
340
341
342     printf("Your returned message is the following: %s\n\n", newBuffer);
343
344
345 }
346
347
348
349
350
351
```

This reads the user's returned message into that "newBuffer" so that it can be printed out for them in the subsequent printf statements. Now let's consider what happens when the user enters in a command number of 5.

Now if a user enters in a command number of 5, they'll enter into this 'if' statement:

```
204
205 // To check if the user wants to set the key to a new value
206 if (commandNo == SUPPORTED_COMMAND_3){
207
208
209
210 // To prompt the user to set a new key
211 printf("\n\nPlease now set the key to your desired value: ");
212 // To get the new key from the user
213 retValueOfFgets = fgets(userKey, MAX_KEY_LENGTH, stdin);
214 /* To check if there was an error getting input (taken from MY assignm
215 3 submission) */
216 if (retValueOfFgets == NULL) {
217
218     if (ferror(stdin)){
219
220         printf("\nAn error has ocured while reading input." "Exiting.
221 clearerr(stdin);
222 exit(errno);
223
224     }
225 }
226 // To handle the input. (see my function before main() for more info)
227 handleInput(userKey);
228 // To convert the input into an int
229 temporaryKey = strtol(userKey, &ptr, 10);
230 /* To check if we have a negative number, continue to the next
231 iteration (non-number chars will be treated as 0s)*/
232 if (temporaryKey < 0){
233     printf("Error! You've entered an invalid key! Ensure that ");
234     printf("your input is a positive number!\n\n");
235     continue;
236 }
237 key = temporaryKey;
```

Where SUPPORTED_COMMAND_3 is a #define with a value of 5. Here, they'll be prompted to provide a new value for the key:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make
run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is s
etting the key (the default value of the key is 4). Type either 0 or a non-number
to exit: 5

Please now set the key to your desired value: 
```

Negative key values will cause an error and bring the user back to the original prompt (as I don't support negative numbers), inputs that are too long will be truncated, and inputs that are non-numbers, including no input, will be treated as 0s and will set the key to a value of 0 (**screenshots of these cases will be provided in the "Screenshots of the execution" section**). Once they input a value, `ioctl()` will be called with the command and key passed in:

```
240
241
242     // To process/reset the key the user desires for their future messages
243     retValueOfIOCTL = ioctl(fileDescriptor, commandNo, key);
244
245     // To check if the io control succeeded
246     if (retValueOfIOCTL < 0){
247
248         printf("Error processing request!\n");
249
250         perror("Error details: ");
251         printf("\n\n");
252
253         return (-1);
254
255     } else {
256
257         printf("IOCTL operation successful!\n\n");
258
259     }
260
261
262
263     // Otherwise, to check if the user wants to encrypt or decrypt a message
264     } else if (commandNo == SUPPORTED_COMMAND_1 ||
265               commandNo == SUPPORTED_COMMAND_2){
266
```

It follows then that `myIoctl()` from the module will be called and we'll have this 'if' statement in `myIoctl()` being reached and executed:

Here again are the parameters of `myIoctl`:

```
373
374     // A function to allow the user to perform either encryption or decryption
375     static long myIoctl(struct file *fs, unsigned int command, unsigned long desiredVa
376
377
```

And here is the 'if' statement:

```
445
446 // This command sets the key to a new value
447 } else if (userCommand == SUPPORTED_COMMAND_3){
448
449
450     printk(KERN_INFO "Entered the 'if' statement for command %d\n", userCommand);
451     printk(KERN_INFO "The old key was: %d\n", cds->key);
452
453
454     // To set the key to a new value for the user
455     cds->key = key;
456
457
458     printk(KERN_INFO "The new key is: %d\n", cds->key);
459
460
461
```

Where again, SUPPORTED_COMMAND_3 is a #define with a value of 5. If the user chooses to enter a command of either 0 or a non-number, they will exit the program, where close() and my module's corresponding myClose() function will be called:

```
354
355 // To close the file associated with the file descriptor
356 close(fileDescriptor);
357
```

So as can be seen, the user application is very simple. All I'm doing is just opening up my special file and then having the user enter a 'while' loop that will run for as long as the user desires, prompting them for input until they enter a 0 or a non-number to exit the program.

(SEE NEXT PAGE FOR THE NEXT SECTIONS)

Approach / What I Did:

The first thing I did was what I always do when beginning an assignment in this class, which was to read the README.md document. After reading the entire document, I reviewed the class lecture held and recorded on April 26 at 11 am. This was the lecture in which you covered the process of writing our own device drivers, going through the process step by step with us. After I did these two things, I broke down the assignment down into 5 main steps. Step one would be to think about the module, what tool(s) I'll need in it and how I can write the file functions to achieve my goal. Step two would be to write the function that encrypts messages. Step three would be to write the function that decrypts messages. Step four would be to write the test application that allows a user to give my device driver out for a try. And finally, step five, would be to refine my device driver before submitting the assignment.

Most of step one was already done for me, in fact, for the entire class, since most of the writing of the module and all of the things pertaining to creating, installing, and registering the device driver were done in the class lecture that I mentioned previously. What I had to do now, however, was to think about how to cater the "template", for lack of a better term, towards my own device driver, my own ideas. So I got to thinking. "Well", I thought, "In the data structure that I'll maintain (called 'cipherDataStructure'), I'm most definitely going to need two buffers. Let's have buffer #1 be the buffer that houses the message that the user wants either encrypted or decrypted and let's call it, 'providedMessage'. The device file function myWrite can be utilized to write from the user's buffer into this buffer." This turned out to be a great idea. I continued, "I can have buffer #2 be the buffer that houses the message that results from either the encryption or decryption operation. Let's call it, 'returnedMessage'. The device file function myRead can be utilized to read from this buffer into a user's buffer, giving them the encrypted or decrypted message that they desire." This also turned out to be a great idea. I then started thinking about the third functionality which was to allow the user to set a key of their choosing. The command for this functionality would be command #5 (commands 3 and 4 are for encryption and decryption, respectively). I figured that I can store into that data structure an int value called "key", which, in my myOpen function, would be assigned a default value of 4. "By entering the command '5', the user can reset this value to the key of their choosing. This request would be processed in myIoCtl." Yet again, this turned out to be a fantastic idea. After I thought about these things, I got to finishing writing the device file functions that would be suited for my particular device driver, such as myWrite, myRead, myIoCtl, and myClose. I had issues creating the kernel object when I ran the "make" command via the command line, as well as issues coding in kernel space for the first time (see "Issues and Resolutions" below), but once I figured those things out, I moved on step two which was to write the function that performs encryption on a message. Step one complete.

Step two revolved around writing the encryption function called "encrypt". Now I didn't write this function in the module initially because debugging would be a nightmare as I'd constantly need to "make clean" and then uninstall everything, and then reinstall everything. I

figured that it would be best if I wrote the code in CLion (which I downloaded for free since I'm a student) and then copy and pasted the function into the module file. So I got to thinking about how to go about the encryption. I thought, "An easy way of capturing the overall message is to capitalize the entire message first of all." This isn't required of course, but I figured that it would simplify things immensely if I capitalized the user's entire message first and then began the process of encryption. After all, what matters to me in my eyes is that the overall message is maintained, rather than the particulars such as capitalization. Turned out I was right (well at least for my purposes). And so I utilized the function from the C library called "strupr", which allows one to capitalize a C string. This function was helpful in allowing me to write my function in CLion, but it was causing me bugs and issues over time in VS Code which is why I had to write my own custom capitalization function (see "Issues and Resolutions" below). But it was useful at the time so that's why I'm mentioning it briefly. I then continued thinking, "Okay, so what I can do is this. I can pass in three parameters: the buffer containing the message we want encrypted, called 'givenMessage', the buffer that will store the newly encrypted message, called 'returnedMessage', and the key that will allow the encryption to take place, called 'key'. I can iterate across givenMessage, going char by char, and subtract the value of 'key' from the captured (or current) character. This difference will be the new/encrypted ASCII character and I can store this character into 'returnedMessage', slowly filling in this buffer with the newly encrypted message." However, there were two cases that I needed to consider. The first one is the case in which we encounter a character that is not a letter. This is simple to deal with as all I did was just "continue" in the 'for' loop along to the next iteration. I'm not concerning myself with characters that are numbers, space, tabs, slashes, etc., etc., so that's all I had to do.

The second case was the case in which the key was great enough and the ASCII value of the current capital letter was small enough such that I had to loop from the ASCII capital A, which has a value of 65, back around to the ASCII capital Z, which has an ASCII value of 90. I dealt with this case in the following way. I thought, "Okay, so what I can do is take the difference between the current letter and the capital 'A', value 65. I can split this difference up into two parts where part 1 will allow us to get to the letter 'A' and then part 2, whatever is leftover, can descend below ASCII 90, or capital Z." So take for instance the letter 'D'. It's ASCII value is 68. If I had a key of 17, what that means is that I first take the difference between 68 and 65 (capital A), which is 3, the first portion, and then subtract this difference from the key to get the second, "leftover" portion, 14, such that when we loop back around to capital Z, ASCII value 90, we descend to the letter M, ASCII value 77, since M is 14 values below Z. So now that the second case had been dealt with, I copied and pasted this function into my module file for my device driver. Step two complete.

Step three is about writing the function that decrypts messages, which I called "decrypt". This function is extremely, extremely similar to the encrypt() function though there are some differences which I will explain. My approach was the same though. I pass in the same three parameters as before: "givenMessage", "returnedMessage", and "key", and I slowly fill in the "returnedMessage" buffer by capturing characters from "givenMessage", but this time,

ascending up the ASCII table to a new character. So going from let's say 'A' to 'D', if the key had a value of 4. And I had to deal with that same first case where in the 'for' loop, I skip over ASCII characters that are not letters. The second case was slightly different this time around as instead of looping back around to 'Z', I was looping back around to 'A', but the logic is so similar that I don't feel that it's necessary to go into detail. My approach here is very similar as it was before in encrypt(). Once I finished up this function, I copied and pasted it into my module file for my device driver. Step three done.

Step four was the testing phase. This was the step in which I had to write the test application that allows the user to test out my code. I thought to myself, "Okay, well, we don't want the user to try out the device driver once and then have the program end. What we can do is write a 'while' loop to allow the user to test as much and for as long as they want." But first I had to set some things up. I first had to declare a whole host of variables and buffers that will take in user input via the C library function fgets(). After this, I began thinking about the best way of prompting the user for input. I thought, "Hmm, it would make the most sense to prompt the user for a command first. I already have a default key with a value of 4 for encryption and decryption, a value stated in the prompt, so maybe they're content with this value and just want to get straight to encrypting and decrypting messages. Or maybe, they don't. Maybe they immediately want to change the key to something crazy big like '3333'. I feel like it would be most appropriate and user-friendly if I gave them this choice from the very beginning." And that's what I did. In the 'while' loop, the first series of statements is a prompt for a user command, with those options being command '3' for encryption, '4' for decryption, '5' for a key reset, and '0' or any non-number character to exit the program. (Note that 0 is not a command supported by the ioctl, it's just an option for the user to type in to finish testing. Also note that when I say "non-number character", I mean input such as "W" or "\\wowza". These are treated as 0s.)

After ascertaining what the best way of interacting with the user is, I thought to myself, "Okay, so now I'm going to need to handle two possible cases using two 'if' statements. Case #1 is the case in which the user wants to set the key by typing in the command '5'. Case #2 is when they type in either '3' or '4' for encryption and decryption, respectively. In the first case, the first 'if' statement, once the aforementioned condition has been reached, I can prompt the user for a new key value. Once they type this value in, I can call ioctl(), passing in the file descriptor, the command, and the new key, allowing the user to have a new key to use for future encryption and decryption operations. In the second case, the second 'if' statement, I can prompt the user for a message and write their message into the buffer called 'providedMessage' from my module data structure, 'cipherDataStructure', by calling write(), passing in the file descriptor, the user's inputted message, and the key (the key doesn't matter here as the module keeps track of it, but I mention it here anyways). Then I call ioctl to process the request, passing in the command which is either '3' for encryption or '4' for decryption, respectively. And finally, I can call read() to read from the buffer 'returnedMessage' from 'cipherDataStructure', into the user's 'newBuffer', which houses the returned message." It was a full proof plan that worked very well. And it did. The only thing though is that as with all

testing, I did run into some bugs. Most things were seamless and ran smoothly, but there were two big bugs. One of them was the issue I mentioned before, where the function `strupr()` was not doing what I expected it to do, which was to capitalize a message, forcing me to write my own custom capitalization function, and the other bug involved returning a message that had leftover artifacts from previous runs. In other words, if I had a long message that was now encrypted, such as an excerpt from the Declaration of Independence, on the next iteration of the 'while' loop, a smaller message such as "Hello" wouldn't be returned. What would be returned instead was "**Hell**onanimous Declaration of the thirteen united States of America", which is clearly something the user would not want to see. They'd much rather just see "Hello" printed (see "Issues and Resolutions" below). But I eventually fixed these bugs. And once I did, I was on to step 5, refining. Step four complete.

Step five was the step in which I aimed to refine everything. By refine, I mean making sure that code is well documented, there are no bugs, every edge case is accounted for, and output is exactly as expected. Fortunately, all of these are the case with my assignment. I tried implementing an idea you gave me, Professor, which was to include encryption for numbers, but I just found it very difficult to do that on top of the encryption I'm already doing for characters. It also caused me to have a lot of bugs (see "Issues and Resolutions" below). So I just decided to abort mission with that and just stick to encrypting and decrypting letters. So to complete this step, as I said, I went back to both files, made sure nothing was hard-coded, edge cases were accounted for, and the user application was user-friendly, and I deemed it in great shape to submit, which is what I did. Step five complete. Assignment complete. Semester complete.

Issues and Resolutions:

The first issue I had revolved around creating the kernel object. When I entered the command "make", I ran into so many warnings and errors because this is the first time I have ever written code in kernel space. I started getting warnings along the lines of "ISO C90 forbids mixed declarations and code" and these other weird errors that went away after I received help from you during an office hours session. I cleared those "ISO C90..." errors myself but I had a heap of other errors that were cleared when you showed me that one of the biggest mistakes I was making was having incorrect signatures for `myRead`, `myWrite`, and `myIoCtl`. I had an incorrect parameter for `myRead` and `myWrite` and I added one too many parameters for `myIoCtl`. You explained to me that because the device driver is part of the kernel, the signatures and structures of the functions are fixed and must be obeyed by everybody. I also made the mistake of not having proper `#include` statements for functions such as `strcpy`. Since we are in the kernel now, the proper `#include` statement is "`#include <linux/string.h>`". After this office

hours session with you, I was finally able to create the kernel object. So I resolved the issues I was having by simply attending office hours and receiving help from you.

The second issue I had was not getting the expected results from the function `strupr()`. As I mentioned in "Approach/What I Did", `strupr()` capitalizes letters for you. The issue is that it wasn't doing a good job during testing. It was giving me unexpected and weird output for some reason and this was preventing me from progressing further in the assignment. So what I was forced to do was to create my own custom capitalization function. It's specific to my needs and my goals for my Caesar Cipher device driver which is to capitalize letters and ignore all other characters. And it was easy to implement. My approach was to simply pass in an array, specifically, the "providedMessage" from the "cipherDataStructure", and iterate across this array, char by char, subtracting 32 to get a lower-case letter's upper-case equivalent (since lower and upper-case letter equivalents are 32 values apart in the ASCII table). So that was my resolution, simply creating my own custom function to capitalize letters which allowed me to get the output I was expecting to see.

The third issue I had was returning a message that had leftover artifacts from previous runs. In other words, if I had a long message that was now encrypted, such as an excerpt from the Declaration of Independence, on the next iteration of the 'while' loop, a smaller message such as "Hello" wouldn't be returned. What would be returned instead was "**H**ellonanimous Declaration of the thirteen united States of America", which is clearly something the user would not want to see. I was stuck on this issue for around 4 hours, specifically, the night before my office hours session with you. I tried everything, from creating a function to set null terminators throughout the entire array and even using `memset()` to set all of the values to 0. Little did I know, there was a simpler solution than any of those. During office hours, you stated that I'm not keeping track of the length of the user's message. You stated that that can be helpful in solving my issue because I can use that length to slap on a null terminator to the end of the "returnedMessage" array from the "cipherDataStructure". Lo and behold, that was the solution to my issue. I created a field in the "cipherDataStructure" called "lengthOfProvidedMessage" and in `myRead()`, I used this field to access index # "lengthOfProvidedMessage" and set that index to be the null terminator. So for example, if I was doing decryption and the decrypted message ("returnedMessage") was "awesome", since "awesome" has 7 letters, I'm going to slap '\0' to the end of it like so: `returnedMessage[lengthOfProvidedMessage] = '\0'`. So the resolution was attending office hours and applying this strategy that you gave me which saved my grade.

The fourth and final issue I had revolved around the encryption and decryption of numbers. You've stated that encryption and decryption for our Caesar Ciphers are implementation dependent and that it's not required to have more than the encryption and decryption of letters. But you did, during office hours, somewhat challenge me to try to do encryption and decryption with numbers. I tried doing so, but unfortunately, it was too difficult. I know given enough time I'd be able to do it but attempting it had caused so many bugs in my code to the point to where encryption and decryption for letters was starting to fail. So I figured

that the best solution, the best resolution, was to stick to handling letters. This is the best way to resolve this issue as I don't want to ruin the tremendous work I have put into this assignment just for the sake of more functionality. I feel that what I have now is very cool and fun so I don't think I'm going to add encryption and decryption for numbers. So you could say that I resolved this issue by taking it easy and playing it safe.

Analysis: NOT REQUIRED

Screenshots of the compilations:

Compilation of the device driver module, the creation of the kernel object:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ make
make -C /lib/modules/`uname -r`/build M=/home/student/Documents/csc415-device-driver-EssaHusary/Module modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-148-generic'
  CC [M] /home/student/Documents/csc415-device-driver-EssaHusary/Module/caesarCipherModule.o
Building modules, stage 2.
MODPOST 1 modules
  CC [M] /home/student/Documents/csc415-device-driver-EssaHusary/Module/caesarCipherModule.mod.o
  LD [M] /home/student/Documents/csc415-device-driver-EssaHusary/Module/caesarCipherModule.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-148-generic'
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

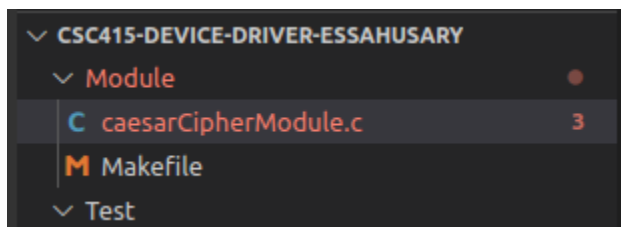
Compilation of the user test application:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make
gcc -c -o husary_essa_HW6_main.o husary_essa_HW6_main.c -g -I.
gcc -o husary_essa_HW6_main husary_essa_HW6_main.o -g -I. -l pthread
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

Screenshot(s) of the execution of the program:

The rubric asks for screenshots of the loading and of the unloading of the module, and the application interaction, so first, I will provide screenshots of the loading of the device driver, followed by screenshots of the actual execution, and then last, the screenshots of the unloading of the device driver.

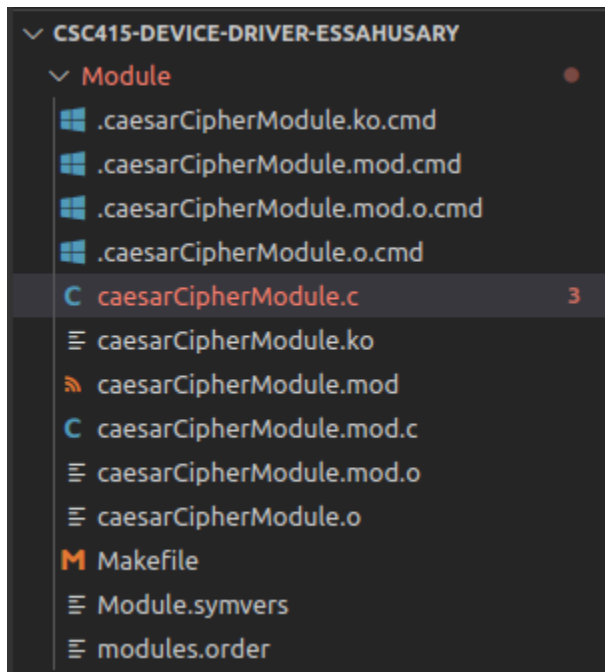
Nothing initially:



Loading the device driver, first by creating the kernel object:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ make
make -C /lib/modules/`uname -r`/build M=/home/student/Documents/csc415-device-driver-EssaHusary/Module modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-148-generic'
  CC [M] /home/student/Documents/csc415-device-driver-EssaHusary/Module/caesarCipherModule.o
Building modules, stage 2.
MODPOST 1 modules
  CC [M] /home/student/Documents/csc415-device-driver-EssaHusary/Module/caesarCipherModule.mod.o
  LD [M] /home/student/Documents/csc415-device-driver-EssaHusary/Module/caesarCipherModule.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-148-generic'
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

Now we have:



Inserting the module into the kernel and creating the device file:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ su
do insmod caesarCipherModule.ko
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ su
do mknod /dev/CaesarCipher c 333 3
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ su
do chmod 666 /dev/CaesarCipher
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

Kernel log:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ tail /var/log/kern.log
May 15 00:47:13 student-VirtualBox kernel: [23894.402380] The provided message is: RCNRPJ WREK
RJKZT DP DRE 415 FGVIRKZEX JPJKVDJ!!!! NFFF!
May 15 00:47:13 student-VirtualBox kernel: [23894.402381] The decrypted message is: ALWAYS FAN
TASTIC MY MAN 415 OPERATING SYSTEMS!!!! WOOO!
May 15 00:47:13 student-VirtualBox kernel: [23894.402388] You got back the message: ALWAYS FAN
TASTIC MY MAN 415 OPERATING SYSTEMS!!!! WOOO!
May 15 01:32:06 student-VirtualBox kernel: [26587.178763] Unloading! Thanks for using my devic
e driver! Hope you had fun!
May 15 01:33:35 student-VirtualBox kernel: [26676.426798] Device registration successful: 0
May 15 01:33:35 student-VirtualBox kernel: [26676.426800] Succeeded adding the new chardev! 0
May 15 01:34:41 student-VirtualBox kernel: [26676.426801] Welcome to my Caesar Cipher device d
river!
May 15 01:34:41 student-VirtualBox kernel: [26742.960928] Unloading! Thanks for using my devic
e driver! Hope you had fun!
May 15 01:35:56 student-VirtualBox kernel: [26817.961888] Device registration successful: 0
May 15 01:35:56 student-VirtualBox kernel: [26817.961889] Succeeded adding the new chardev! 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

A closer look:

```
May 15 01:35:56 student-VirtualBox kernel: [26817.961888] Device registration successful: 0
May 15 01:35:56 student-VirtualBox kernel: [26817.961889] Succeeded adding the new chardev! 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

The file is created:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ ls /dev/C*
/dev/CaesarCipher
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

Now screenshots of the execution-----

Let's encrypt a message with the default key value of 4:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 3

Please enter a message you want encrypted or decrypted: Always FanTastic man!!!

IOCTL operation successful!

Return value of read: 0
Your returned message is the following: WHSWUO BWJPWOPEY IWJ!!!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

Now let's decrypt that message with that same default key value of 4:


```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 4

Please enter a message you want encrypted or decrypted: WHSWUO BWJPWOPEY IWJ!!!

IOCTL operation successful!

Return value of read: 0
Your returned message is the following: ALWAYS FANTASTIC MAN!!!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

Just for fun, let's look at the kernel log:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ tail /var/log/kernel.log
May 15 01:43:11 student-VirtualBox kernel: [27252.891799] You wrote the message: Always Fantastic man!!!
May 15 01:43:11 student-VirtualBox kernel: [27252.891804] Entered the 'if' statement for command 3
May 15 01:43:11 student-VirtualBox kernel: [27252.891805] The provided message is: ALWAYS FANTASTIC MAN!!!
May 15 01:43:11 student-VirtualBox kernel: [27252.891805] The encrypted message is: WHSWUO BWJPWOPEY IWJ!!!
May 15 01:43:11 student-VirtualBox kernel: [27252.891824] You got back the message: WHSWUO BWJPWOPEY IWJ!!!
May 15 01:44:44 student-VirtualBox kernel: [27345.918591] You wrote the message: WHSWUO BWJPWOPEY IWJ!!!
May 15 01:44:44 student-VirtualBox kernel: [27345.918595] Entered the 'if' statement for command 4
May 15 01:44:44 student-VirtualBox kernel: [27345.918595] The provided message is: WHSWUO BWJPWOPEY IWJ!!!
May 15 01:44:44 student-VirtualBox kernel: [27345.918596] The decrypted message is: ALWAYS FANTASTIC MAN!!!
May 15 01:44:44 student-VirtualBox kernel: [27345.918612] You got back the message: ALWAYS FANTASTIC MAN!!!
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

Now let's set the key to a value other than the default value, encrypt a message with that new key, and then decrypt that returned message back into the original message:


```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 5

Please now set the key to your desired value: 3245
New key number: 3245
IOCTL operation successful!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 3

Please enter a message you want encrypted or decrypted: CSC 415 is an operating systems course !
IOCTL operation successful!

Return value of read: 0
Your returned message is the following: HXH 415 NX FS TUJWFYNLS XDXYJRX HTZWXJ!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 4

Please enter a message you want encrypted or decrypted: HXH 415 NX FS TUJWFYNLS XDXYJRX HTZWXJ !
IOCTL operation successful!

Return value of read: 0
Your returned message is the following: CSC 415 IS AN OPERATING SYSTEMS COURSE!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

Kernel log for setting the key and the initial encryption:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ tail /var/log/kernel.log
May 15 01:51:27 student-VirtualBox kernel: [27748.853660] The old key was: 4
May 15 01:51:27 student-VirtualBox kernel: [27748.853661] The new key is: 21
May 15 01:51:59 student-VirtualBox kernel: [27780.344188] Entered the 'if' statement for command 5
May 15 01:51:59 student-VirtualBox kernel: [27780.344189] The old key was: 4
May 15 01:51:59 student-VirtualBox kernel: [27780.344190] The new key is: 21
May 15 01:52:22 student-VirtualBox kernel: [27803.136650] You wrote the message: CSC 415 is an operating systems course!
May 15 01:52:22 student-VirtualBox kernel: [27803.136657] Entered the 'if' statement for command 3
May 15 01:52:22 student-VirtualBox kernel: [27803.136658] The provided message is: CSC 415 IS AN OPERATING SYSTEMS COURSE!
May 15 01:52:22 student-VirtualBox kernel: [27803.136659] The encrypted message is: HXH 415 NX FS TUJWFYNSL XDXYJRX HTZWXJ!
May 15 01:52:22 student-VirtualBox kernel: [27803.136677] You got back the message: HXH 415 NX FS TUJWFYNSL XDXYJRX HTZWXJ!
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

We can see that that key is treated as a value of 21 due to performing a modulus.

Kernel log for the full encryption-decryption cycle:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ tail /var/log/kernel.log
May 15 01:55:36 student-VirtualBox kernel: [27997.390390] You wrote the message: CSC 415 is an operating systems course!
May 15 01:55:36 student-VirtualBox kernel: [27997.390396] Entered the 'if' statement for command 3
May 15 01:55:36 student-VirtualBox kernel: [27997.390396] The provided message is: CSC 415 IS AN OPERATING SYSTEMS COURSE!
May 15 01:55:36 student-VirtualBox kernel: [27997.390397] The encrypted message is: HXH 415 NX FS TUJWFYNSL XDXYJRX HTZWXJ!
May 15 01:55:36 student-VirtualBox kernel: [27997.390470] You got back the message: HXH 415 NX FS TUJWFYNSL XDXYJRX HTZWXJ!
May 15 01:55:48 student-VirtualBox kernel: [28009.438500] You wrote the message: HXH 415 NX FS TUJWFYNSL XDXYJRX HTZWXJ!
May 15 01:55:48 student-VirtualBox kernel: [28009.438504] Entered the 'if' statement for command 4
May 15 01:55:48 student-VirtualBox kernel: [28009.438505] The provided message is: HXH 415 NX FS TUJWFYNSL XDXYJRX HTZWXJ!
May 15 01:55:48 student-VirtualBox kernel: [28009.438506] The decrypted message is: CSC 415 IS AN OPERATING SYSTEMS COURSE!
May 15 01:55:48 student-VirtualBox kernel: [28009.438572] You got back the message: CSC 415 IS AN OPERATING SYSTEMS COURSE!
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

Let's look at some edge cases, starting with entering non-numbers when prompted for a command (non-numbers are treated as the number 0 and so, will exit the program):

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: w
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: --111

Your input is too long. Truncating...

student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: bad command man!

Your input is too long. Truncating...

student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: @@!@Ddhvf

Your input is too long. Truncating...

student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

If input is too long, it'll be truncated:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 2344242

Your input is too long. Truncating...

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

If the user inputs a number other than 0, 3, 4, or 5, the prompt will just keep reappearing:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 1

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 2

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 6

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 7

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 8

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

Another example of this:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 23453

Your input is too long. Truncating...

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: -2

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: -234253453453

Your input is too long. Truncating...

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

If a user types in 3 or 4 and are prompted with a message, any input is valid input. The only edge case occurs when the input exceeds 1 MB, in which case, it'll be truncated. I'm going to set the buffer to 128 bytes just for the sake of example, write a message that'll be encrypted but truncated first, and then decrypt it:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 3

Please enter a message you want encrypted or decrypted: If a user types in 3 or 4 and are prompted with a message, any input is valid input. The only edge case occurs when the input exceeds 1 MB, in which case, it'll be truncated. I'm going to set the buffer to 128 bytes just for the sake of example:

Your input is too long. Truncating...

IOCTL operation successful!

Return value of read: 0
Your returned message is the following: EB W QOAN PULAO EJ 3 KN 4 WJZ WNA LNKILPAZ SEPD W IAOWCA, WJU EJLQP EO RWHEZ EJLQP. PDA KJHU AZCA YWOA KYQNO SDAJ PDA EJLQP A

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 4

Please enter a message you want encrypted or decrypted: EB W QOAN PULAO EJ 3 KN 4 WJZ WNA LNKILPAZ SEPD W IAOWCA, WJU EJLQP EO RWHEZ EJLQP. PDA KJHU AZCA YWOA KYQNO SDAJ PDA EJLQP A

Your input is too long. Truncating...

IOCTL operation successful!

Return value of read: 0
Your returned message is the following: IF A USER TYPES IN 3 OR 4 AND ARE PROMPTED WITH A MESSAGE, ANY INPUT IS VALID INPUT. THE ONLY EDGE CASE OCCURS WHEN THE INPUT EXCEEDS 1 MB, IN WHICH CASE, IT'LL BE TRUNCATED. I'M GOING TO SET THE BUFFER TO 128 BYTES JUST FOR THE SAKE OF EXAMPLE:

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

There was a second truncation when decrypting because I accidentally pasted additional spaces from the encrypted message, but hopefully you see my point.

If a user types in the command 5 and tries setting a key, the following will happen if the key is negative:


```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
gcc -c -o husary_essa_HW6_main.o husary_essa_HW6_main.c -g -I.
gcc -o husary_essa_HW6_main husary_essa_HW6_main.o -g -I. -l pthread
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 5

Please now set the key to your desired value: -12
Error! You've entered an invalid key! Ensure that your input is a positive number!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

The original prompt will appear again. I don't support negative numbers for my device driver so that's why they must try again. Now let's see what happens when they type a non-number:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 5

Please now set the key to your desired value: wrong key man!
Your input is too long. Truncating...

New key number: 0
IOCTL operation successful!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

Or even:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 5

Please now set the key to your desired value: --1
New key number: 0
IOCTL operation successful!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

So non-numbers are treated as 0s and will therefore set the key to a value of 0. Now what will happen if the key input is too long?:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ make run
./husary_essa_HW6_main

Returned file descriptor value: 3
Successfully opened the device file!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 5

Please now set the key to your desired value: 235325352353453241658687687

Your input is too long. Truncating...

New key number: 23532
IOCTL operation successful!

Please enter a command number, where 3 is encryption, 4 is decryption, and 5 is setting the key (the default value of the key is 4). Type either 0 or a non-number to exit: 0
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

It will be truncated because for both the command and the key, I've set a max of 6 bytes including the newline character, '\n', that gets slapped on because of the nature of fgets, so really, only 5 digit numbers are accepted by my test program.

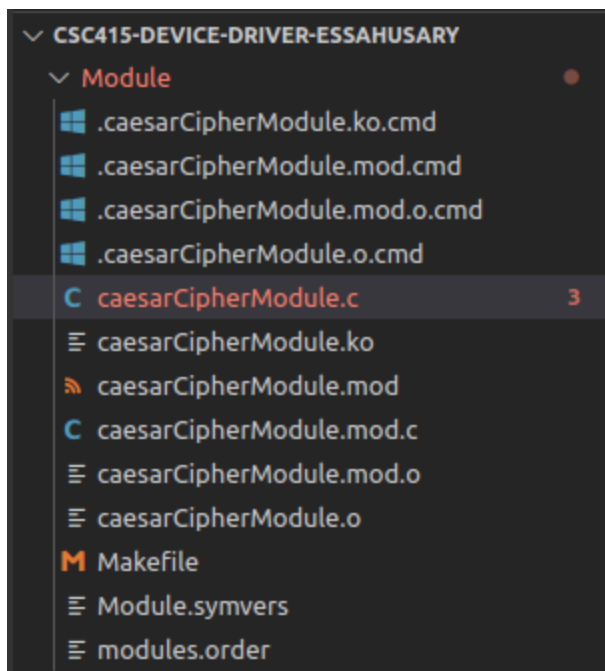
Kernel log:


```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$ tail /var/log/kernel.log
May 15 02:07:20 student-VirtualBox kernel: [28701.572051] You got back the message: IF A USER TYPES IN 3 OR 4 AND ARE PROMPTED WITH A MESSAGE, ANY INPUT IS VALID INPUT. THE ONLY EDGE CASE OCCURS WHEN THE INPUT E
May 15 02:14:15 student-VirtualBox kernel: [29116.163967] Entered the 'if' statement for command 5
May 15 02:14:15 student-VirtualBox kernel: [29116.163969] The old key was: 4
May 15 02:14:15 student-VirtualBox kernel: [29116.163969] The new key is: 0
May 15 02:14:49 student-VirtualBox kernel: [29150.200662] Entered the 'if' statement for command 5
May 15 02:14:49 student-VirtualBox kernel: [29150.200664] The old key was: 4
May 15 02:14:49 student-VirtualBox kernel: [29150.200664] The new key is: 0
May 15 02:16:14 student-VirtualBox kernel: [29235.185710] Entered the 'if' statement for command 5
May 15 02:16:14 student-VirtualBox kernel: [29235.185712] The old key was: 4
May 15 02:16:14 student-VirtualBox kernel: [29235.185713] The new key is: 2
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Test$
```

As we saw before, because of the non-numbers, the new keys will be 0. On the other hand, the super, super long input I just entered that is now truncated really is just a key of 2 due to a truncated number % 26.

Now for the unloading of the module-----

Before, the kernel object was here:



It will now disappear, starting with a make clean:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ make clean
make -C /lib/modules/`uname -r`/build M=/home/student/Documents/csc415-device-driver-EssaHusary/Module clean
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-148-generic'
CLEAN /home/student/Documents/csc415-device-driver-EssaHusary/Module/Module.symvers
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-148-generic'
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ tail /var/log/kern.log
```

Deletion of the module and the device file:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ sudo rm /dev/CaesarCipher
[sudo] password for student:
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ sudo rmmod caesarCipherModule.ko
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

Proof the device file is deleted:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ ls /dev/C*
ls: cannot access '/dev/C*': No such file or directory
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

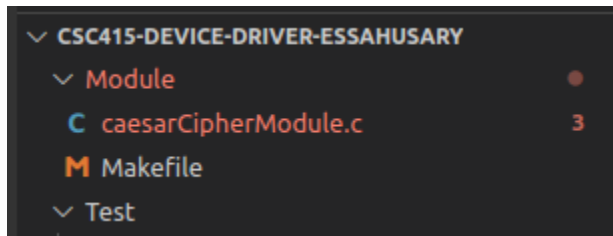
Kernel log:

```
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$ tail /var/log/kern.log
May 15 02:14:15 student-VirtualBox kernel: [29116.163967] Entered the 'if' statement for command 5
May 15 02:14:15 student-VirtualBox kernel: [29116.163969] The old key was: 4
May 15 02:14:15 student-VirtualBox kernel: [29116.163969] The new key is: 0
May 15 02:14:49 student-VirtualBox kernel: [29150.200662] Entered the 'if' statement for command 5
May 15 02:14:49 student-VirtualBox kernel: [29150.200664] The old key was: 4
May 15 02:14:49 student-VirtualBox kernel: [29150.200664] The new key is: 0
May 15 02:16:14 student-VirtualBox kernel: [29235.185710] Entered the 'if' statement for command 5
May 15 02:16:14 student-VirtualBox kernel: [29235.185712] The old key was: 4
May 15 02:16:14 student-VirtualBox kernel: [29235.185713] The new key is: 2
May 15 02:28:38 student-VirtualBox kernel: [29979.046247] Unloading! Thanks for using my device driver! Hope you had fun!
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

A closer look:

```
May 15 02:28:38 student-VirtualBox kernel: [29979.046247] Unloading! Thanks for using my device driver! Hope you had fun!
student@student-VirtualBox:~/Documents/csc415-device-driver-EssaHusary/Module$
```

After:



(WORKS CITED PAGE ON NEXT PAGE)

WORKS CITED PAGE:

All of the C library functions that I used are the following: `strlen`, `strcpy`, `strchr`, `getchar`, `open`, `close`, `read`, `write`, `ioctl`, `fgets`, `ferror`, `clearer`, `exit`, `strtol`, and `perror`. Some of them are new to me, while most of them I've used multiple times before. In any case, I'm citing the man pages below. As for the kernel functions, the kernel structs, and things pertaining to the kernel found in my module, there are too many to list. However, every kernel function and kernel struct from the module, including everything from `printk` to `cdev` to `register_chardev_region` were taught to me by you Professor during the class lecture, so there isn't a need to "cite" them since the lecture isn't an outside source, but I am mentioning this fact here because it's obvious that I didn't know anything about this topic prior to this assignment. In this writeup, I have used two sources to help me define major numbers, minor numbers, and the `cdev` struct. They are the articles by O'Reilly titled, "Char Drivers – Linux Device Drivers, 3rd Edition [Book]" and "Major and Minor Numbers – Linux Device Drivers, Second Edition [Book]". Everything used in my code and in my writeup is cited below, except for the kernel functions and structs and things of that nature, since I used the lecture for everything pertaining to the module and not any outside sources. And as you know, most of the code found in the module is inspired by your code from the lecture.

"Char Drivers – Linux Device Drivers, 3rd Edition [Book]." *oreilly.com*, O'Reilly Media, Inc. Web. 13 May 2023.

[3. Char Drivers - Linux Device Drivers, 3rd Edition \[Book\] \(oreilly.com\)](#)

"Linux Programmer's Manual." 15 Sept. 2017. (Man pages)

"Major and Minor Numbers – Linux Device Drivers, Second Edition [Book]." *oreilly.com*, O'Reilly Media, Inc. Web. 13 May 2023.

[Major and Minor Numbers - Linux Device Drivers, Second Edition \[Book\] \(oreilly.com\)](#)