Javier Vega
5154182
12/03/19

Part 1 – Device Driver Source Code Name motor1.c

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/device.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/uaccess.h>
#include <linux/ctype.h>
#include <linux/mutex.h>
#include <asm/io.h>
#include <linux/types.h>

#define DEVICE_NAME "MotorDevice"
#define CLASS_NAME  "Motor"

MODULE_AUTHOR("Javier Vega");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("A Device Driver that controls a motor.");
MODULE_VERSION("1.0");


static int motor_open(struct inode *, struct file *);
static int motor_release(struct inode *, struct file *);
static ssize_t motor_read(struct file *, char *, size_t, loff_t *);
static ssize_t motor_write(struct file *, const char *, size_t, loff_t *`);
static long motor_ioctl(struct file *, unsigned int, unsigned long);
static ssize_t motor_stop(struct file *file);
static ssize_t motor_rotate(struct file *file, int direction);

static struct file_operations file_operations_t =
{
  .open           = motor_open,
  .read           = motor_read,
  .write          = motor_write,
  .release        = motor_release,
  .unlocked_ioctl = motor_ioctl
};

static DEFINE_MUTEX(motor_mutex);
static struct class *motor_class = NULL;
static struct device *motor_device;
static int major_number;
static char message_buffer[256] = { 0 };
static short size_of_message;

/** @brief The LKM initialization function
 *  The static keyword restricts the visibility of the function to within
 *  this C file. The __init macro means that for a built-in driver (not a LKM)
 *  the function is only used at initialization time and that it can be
 *  discarded and its memory freed up after that point.
 *  @return returns 0 if successful
 */
static int __init motor_init(void)
{
```

```
  printk(KERN_INFO "Motor: Initializing the Motor LKM\n");

  major_number = register_chrdev(0, DEVICE_NAME, &file_operations_t);
  if(major_number < 0)
  {
    printk(KERN_ALERT "Motor: failed to register a major number\n");

    return major_number;
  }
  printk(KERN_INFO "Motor: Registerd Correctrly %d\n", major_number);

  motor_class = class_create(THIS_MODULE, CLASS_NAME);
  if(IS_ERR(motor_class))
  {
    unregister_chrdev(major_number, DEVICE_NAME);

    printk(KERN_ALERT "Motor: Failed to register device class\n");

    return PTR_ERR(motor_class);
  }
  printk(KERN_INFO "Motor: device class registered correctly\n");

  motor_device = device_create(motor_class, NULL, MKDEV(major_number, 0), NULL,
DEVICE_NAME);
  if (IS_ERR(motor_device))
  {
    // Repeated code but the alternative is goto statements
    class_destroy(motor_class);
    unregister_chrdev(major_number, DEVICE_NAME);

    printk(KERN_ALERT "Motor: Failed to create the device\n");

    return PTR_ERR(motor_device);
  }
  printk(KERN_INFO "Motor: device class created correctly\n");

  mutex_init(&motor_mutex);

  return 0;
}

/** @brief The device open function that is called each time the device is
 *  opened. This will only increment the numberOpens counter in this case.
 *  @param inode_ptr A pointer to an inode object (defined in linux/fs.h)
 *  @param file_ptr A pointer to a file object (defined in linux/fs.h)
 */
static int motor_open(struct inode *inode_ptr, struct file *file_ptr)
{
  if(!mutex_trylock(&motor_mutex)){
    printk(KERN_ALERT "Motor: Device in use by another process\n");

    return -EBUSY;
  }
  printk(KERN_INFO "Motor: Driver have been opened\n");

  number_of_opens++;

  return 0;   // Successfully opened
}
```

```
/** @brief The device release function that is called whenever the device is
 *  closed/released by the userspace program.
 *  @param inode_ptr A pointer to an inode object (defined in linux/fs.h)
 *  @param file_ptr A pointer to a file object (defined in linux/fs.h)
 */
static int motor_release(struct inode *inode_ptr, struct file *file_ptr)
{
  mutex_unlock(&motor_mutex);
  printk("Motor: Device Driver has been closed\n");
  return 0;
}

/** @brief This function is called whenever device is being read from
 *  user space i.e. data is being sent from the device to the user.
 *  In this case is uses the copy_to_user() function to send the buffer string
 *  to the user and captures any errors.
 *  @param file_ptr A pointer to a file object (defined in linux/fs.h)
 *  @param user_buffer The pointer to the buffer to write the data
 *  @param buffer_size The length of the b
 *  @param offset_ptr The offset if required
 */
static ssize_t motor_read(struct file *file_ptr, char *user_buffer, size_t
buffer_size, loff_t *offset_ptr)
{
  int error_number = 0;
  error_number = copy_to_user(user_buffer, message_buffer , size_of_message);
  if(error_number == 0)
  {
    printk(KERN_INFO "Motor: Reading %c characters from Motor Device Driver\n",
size_of_message);
    return (size_of_message = 0);
  }
  else
  {
    printk(KERN_INFO "Motor: Failed read operation\n");
    return -EFAULT;
  }
  return 0;
}

/** @brief This function is called whenever the device is being written
 *  to from user space i.e. data is sent to the device from the user.
 *  The data is copied to the message[] array in this LKM using the
 *  copy_from_user() function along with the length of the string.
 *  @param file_ptr A pointer to a file object
 *  @param user_buffer The buffer with string from the user program
 *  @param buffer_size The length of the user program buffer
 *  @param offset_ptr The offset if required
 */
static ssize_t motor_write(struct file *file_ptr, const char *user_buffer, size_t
buffer_size, loff_t *offset_ptr)
{
  unsigned long bytes_not_copied;

  if(buffer_size <= 0)
  {
    printk(KERN_INFO "Motor: No Data in the buffer\n");
    return -1;
```

```
  }
  printk(KERN_INFO "Motor: Received %lu characters from user\n", buffer_size);

  bytes_not_copied = copy_from_user(message_buffer, user_buffer, buffer_size);
  if(bytes_not_copied > 0)
  {
    printk(KERN_INFO "Motor: Error while writing\n");
    return -1;
  }
  size_of_message = strlen(message);

  return size_of_message;
}

/**
 * Allows the user to set mode of the driver
 */
static long motor_ioctl(struct file *file_ptr, unsigned int command, unsigned long
arg)
{
  // Handles all the control operations for the motor
  switch(command)
  {
    case MOTOR_STOP:
      motor_stop(file_ptr);
      printk(KERN_INFO "Motor: Stoped\n");

      break;
    case MOTOR_ROTATE:
      motor_rotate(file_ptr, arg);
      printk(KERN_INFO "Motor: Started to rotate\n");

      break;
    case default:
      printk(KERN_INFO "Motor: No command received\n");

      return -ENOTTY;
  }

  return 0;
}

static ssize_t motor_stop(struct file *file_ptr)
{
  // Code that will handle stoping the motor

  return 0;
}

static ssize_t motor_rotate(struct file *file_ptr, int direction)
{
  switch (direction)
  {
    case ROTATE_LEFT:
      /* Handle rotate left for motor */
      printk(KERN_INFO "Motor: Rotating to the Right\n");
      break;
    case ROTATE_RIGHT:
      /* Hanlde rotate right for motor */
```

```
      printk(KERN_INFO "Motor: Rotating to the Left\n");
      break;
    default:
      printk(KERN_INFO "Motor: Invalid direction\n");
      break;
  }

  return 0;
}

/** @brief The LKM cleanup function
 *  Similar to the initialization function, it is static. The __exit macro
 *  notifies that if this code is used for a built-in driver (not a LKM) that
 *  this function is not required.
 */
static void __exit motor_exit(void)
{
  device_destroy(motor_class, MKDEV(major_number, 0));
  class_unregister(motor_class);
  class_destroy(motor_class);
  unregister_chrdev(major_number, DEVICE_NAME);
  mutex_destroy(&motor_mutex);
  del_timer(&timer);

  printk(KERN_INFO "Motor: Goodbye from the Motor Device Driver!\n");
}

module_init(motor_init);
module_exit(motor_exit);
```

Part 2 – Prepare the Device Driver

(a) From the top-level of the Linux source code directory, change directory into ***<top-level_linux_source_dir>/drivers/char/***, create a new directory called motor, and then add the driver source code motor1.c

(b) Next, we need to modify the Kconfig at ***<top-level_linux_source_dir>/drivers/char/*** so that we can select our device driver when compiling the Linux Kernel. Add the following:

```
config MOTOR
        tristate "Motor"
          default M
          help
             Select this option if you have a motor in your system.
```

(c) After modifying the Kconfig, we need to modify the Makefile at ***<top-level_linux_source_dir>/drivers/char/*** so that it finds the Makefile for our device driver and compiles it. Add the following: ***obj-$(CONFIG_MOTOR) += motor/***

(d) To compile our Motor Device Driver we need to create a new Makefile in the directory where our Motor Device Driver is located, which is **<top-level_linux_source_dir>/drivers/char/motor**. In the Makefile add:
obj-$(CONFIG_MOTOR) += motor1.o
If you want to build once the kernel is compiled we can select it using the configuration, this is done by running the following command:
> make ARCH=arm CROSS_COMPILE=<target_architecture_compiler> gconfig
Once we have the gconfig window you can select the Motor driver under **Device Drivers > Character devices > Morse**. By default we have made it so that it can be dynamically loaded after the kernel has been built. If you want to build the driver with the kernel set the option to **Y**. If you wish not to include it you can set the option to **N**.

(e) If you want to place the device driver manually once you build it copy the driver to ***/lib/module/<kernel_version>***. If you want to place the module during the make process, then after compiling the module you can install it in the desired directory by running the following command:
> **make ARCH=arm CROSS_COMPILE=<target_architecture_compiler> INSTALL_MOD_PATH=<path_to_place_module> modules_install**

(f)

(a) To load the device driver, run **sudo insmod motor1.ko** or **sudo modprobe motor1**. If you want to remove it run **sudo rmmod motor1** or **sudo modprobe -r motor1**.

(b) To load and unload it automatically as part of init process runlevel 3 first add the necessary script to load the driver at the /etc/init.d folder then go into /etc/rc3.d script folder and add a symbolic link to the script previously created at the init.d folder with the correct S and K. The S will be for startup instruction and K for shutting down instruction. We can add S99motor1for startup service and K05motor1killing the service.

Part 3 – C Program testMotor.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sys/ioctl.h>
#include <unistd.h>

#define COMMAND_ROTATE 0
#define COMMAND_STOP 1
#define ROTATE_LEFT 100
#define ROTATE_RIGHT 101

int main(int argc, char *argv[])
{
  int ret, file_descriptor;
  char choice;

  printf("Starting motor test code example...\n");

  // Open the device driver with read/write access
  file_descriptor = open("/dev/motor1", O_RDWR);
  if(file_descriptor < 0)
  {
    perror("Failed to open the device...");
    return errno;
  }

  // Issues a command to rotate the motor to the left
  ret = ioctl(file_descriptor, COMMAND_ROTATE, ROTATE_LEFT);
  if(ret < 0)
  {
    perror("Failed to start rotating the motor to the left.");
    return errno;
  }
  printf("Motor started to rotate to the left\n");

  // Issues a command to stop the motor
```

```
  ret = ioctl(file_descriptor, COMMAND_STOP, NULL);
  if(ret < 0)
  {
    perror("Failed to stop the motor.");
    return errno;
  }
  printf("Motor has been stoped\n");

  printf("Done\n");
  close(file_descriptor);

  return 0;
}
```