```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <linux/printk.h>
#include <linux/kdev_t.h>
#include <linux/cdev.h>
#include <linux/string.h>
#include <asm/uaccess.h>

// starting number of minor number
#define S_N 1
// number of minor number (number of device)
#define N_D 2
#define DEVICE_NAME "helloworld char driver"


static char msg[] = "Hello World!!!";
// dev_v is __u32: unsigned 32 bit, and this is a global variable
// with out extern keyword -> this is a definition(alloc 32 bits)
// It store both the major and minor number: top 12 bit for major, 20 bit for minor
static dev_t devno;
// cdev is a struct, one attribute is file_operations, one is dev_t
static struct cdev mydev;


/**
 * in user space: int fd = open("/dev/helloworld", O_RDONLY);
 * inode contains device info from /dev/helloworld
 * fp is the kernel's file structure for this opened instance
```

```
 * The user's fd (like 3) maps to this fp in kernel space
 */
static int myopen(struct inode *inode, struct file *fp) {
    printk("Device " DEVICE_NAME " opened.\n");
    return 0;
}

/**
 * in user space: char buffer[100];
 *                read(fd, buffer, sizeof(buffer));
 * fp is the kernel's file structure for this opened instance
 * The user's fd (like 3) maps to this fp in kernel space
 * the user space pointer buffer = buf
 * count means the space offered by user
 * position useless here
 */
static ssize_t myread(struct file *fp, char __user *buf, size_t
count, loff_t *position) {
    int num;
    int ret;
    if (count < strlen(msg)) {
        num = count;
    } else {
        // another strlen in <linux/string.h>
        num = strlen(msg);
    }
    // let the kernel pointer and user pointer talking to each
other
    ret = copy_to_user(buf, msg, num); // (dst, src, num)
    if (ret) { //non-zero means failure -> if condition is true
        printk("Fail to copy data from the kernel space to the
user space.\n");
    }
    return num;
}

static int myclose(struct inode *inode, struct file *fp) {
```

```
65      printk("Device " DEVICE_NAME " closed.\n");
66      return 0;
67  }
68
69  // a collection of function pointers specify the available
    operations for the user and how to do
70  // kind of function overwritting
71  static struct file_operations myfops = {
72      owner: THIS_MODULE,
73      open: myopen,
74      read: myread,
75      release: myclose
76  };
77
78  // __init is a macro: telling compiler to put this function to
    the .init.text section (default is .text section) of the
    assembly code so that the kernel could find it
79  static int __init helloworldinit(void) {
80      int ret;
81      // register a major number
82      /**
83       * Usage: int alloc_chrdev_region(dev_t *dev, unsigned
    baseminor,
84                     unsigned count, const char *name)
85                     (pointer points to name is const)
86       * Explaination:
87       * alloc_chrdev_region() - register a range of char device
    numbers
88       * @dev: output parameter for first assigned number
89       * @baseminor: first of the requested range of minor numbers
90       * @count: the number of minor numbers required
91       * @name: the name of the associated device or driver
92       *
93       * Allocates a range of char device numbers.  The major
    number will be
94       * chosen dynamically, and returned (along with the first
    minor number)
```

```
95       * in @dev.  Returns zero or a negative error code.
96       */
97      ret = alloc_chrdev_region(&devno, S_N, N_D, DEVICE_NAME);
98      if (ret < 0) {
99          // inside kernel memory space, you cannot use perror or
    printf
100         // usage is differnt, it will concate automatically!
101         printk("failure" DEVICE_NAME " cannot get major
    number.\n");
102         return ret;
103     }
104     int major = MAJOR(devno);
105     printk("Device " DEVICE_NAME " initiailized (major number =
    %d).\n", major);
106     // register a char device
107     // init the mydev struct with myfops info
108     cdev_init(&mydev, &myfops);
109     mydev.owner = THIS_MODULE;
110     /**
111      * Usage: int cdev_add(struct cdev *p, dev_t dev, unsigned
    count)
112      * Explanation:
113      * cdev_add() - add a char device to the system
114      * @p: the cdev structure for the device
115      * @dev: the first device number for which this device is
    responsible
116      * @count: the number of consecutive minor numbers
    corresponding to this
117      *         device
118      *
119      * cdev_add() adds the device represented by @p to the
    system, making it
120      * live immediately.  A negative error code is returned on
    failure.
121      */
122     ret = cdev_add(&mydev, devno, N_D);
123     if (ret) {
```

```
124         printk("Device " DEVICE_NAME " register fail.\n");
125         return ret;
126     }
127     return 0;
128 }
129 
130 # __exit is a macro: telling compiler to put this function to
    the .exit.text section (default is .text section) of the
    assembly code so that the kernel could find it
131 static void __exit helloworldexit(void) {
132     // delete the row of the Char device table
133     cdev_del(&mydev);
134     // unrigester the major and minor number and free space
135     unregister_chrdev_region(devno, N_D);
136     printk("Device " DEVICE_NAME " unloaded.\n");
137 }
138 
139 /**
140  * Usage: #define module_init(x)    __initcall(x);
141  * Explanation:
142  * module_init() - driver initialization entry point
143  * @x: function to be run at kernel boot time or module
    insertion
144  *
145  * module_init() will either be called during do_initcalls() (if
146  * builtin, driver is a part of the kernal) or at module
    insertion time (if a module, can be
147  * loaded into the kernel).  There can only be one per module.
148  *
149  */
150 module_init(helloworldinit);
151 
152 /**
153  * Usage: #define module_exit(x)    __exitcall(x);
154  * Explanation:
155  * module_exit() - driver exit entry point
156  * @x: function to be run when driver is removed
```

```
157  *
158  * module_exit() will wrap the driver clean-up code
159  * with cleanup_module() when used with rmmod when
160  * the driver is a module.  If the driver is statically
161  * compiled into the kernel, module_exit() has no effect.
162  * There can only be one per module.
163  */
164 module_exit(helloworldexit);
165
166 MODULE_LICENSE("GPL");
167 MODULE_AUTHOR("Qixin Wang");
168 MODULE_DESCRIPTION("Hello world character device driver");
169
```

1. Most of the kernels are interrupt handlers. System Calls are software interrupts. The interrupt routines are so-called drivers. The lookup table is where the driver programs residence and needed to be registered here (`insmod`)
2. Kconfig software helps to customize how to compile the files
3. `static` keyword for identifiers (function or global variable) -> this thing is only visible for this file (some kind of encapsulation)
4. Declaration of a variable: know the name + type

   Definition of a variable: alloc the memory for it

   Declaration of a function: know signature (name + return type + parameter list)

   Definition of a variable: signature + function body (alloc program memory)
5. `extern` for a global variable: it is declared but not defined

   Without `extern` -> the global variable is defined
6. Major number -> id of the driver

Minor number -> parameter used by the driver program to distinguish the hardware instances that share this driver.

1 major number could be mapped to multiple minor numbers

(Major number, Minor number) -> specify a device file

7. Register a char device?

    1. Character device driver table: rows are indexed by the major number. columns are indexed by the system call function name.
    2. if the user calls a character device open system call, then the kernel will search for the character device driver table based on the device file that the user is calling from the device files. I know it will get the major number.
    3. Based on the major number, search for the columns labeled open and from that open column of the major row, it will find a function pointer.
    4. So setting up these function pointers in the character device driver table, that's what the register a character device means.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void main() {
    int fd;
    char buf[1024];
    // calls -> myopen
    fd = open("/dev/helloworld", O_RDONLY);
    if (fd < 0) {
```

```
14          perror("Open /dev/helloworld failure.");
15          exit(EXIT_FAILURE);
16      }
17      // calls -> myread
18      int num = read(fd, buf, sizeof(buf)-1);
19      buf[num] = 0;
20      printf("Got the message from /dev/helloworld: \"%s\".\n",
    buf);
21      // calls -> myclose
22      close(fd);
23      exit(EXIT_SUCCESS);
24  }
25
```