DISCLAIMER

Everything in this document shall not, under any circumstances, hold any legal liability whatsoever. Any usage of the data and information in this document shall be solely on the responsibility of the user. This document user has to take written consent from the author.

CHARACTER DEVICE DRIVER

A character device driver in Linux is a type of device driver that provides access to devices at the character level. Character devices handle data in a sequential manner, usually one character at a time, such as keyboards, mice, serial ports, sound cards, etc.

To develop a character device driver in Linux, you need to perform the following steps:

1. **Set up the development environment:** Install the necessary tools such as a C compiler, kernel headers, and development packages.
2. **Create the driver source file:** Write the driver code in C programming language. This code will define the driver functionality, including initialization, cleanup, and operations on the device.
3. **Define the device file:** Each character device is represented by a device file in the **/dev** directory. You need to create or identify a suitable device file name and major number for your driver.
4. **Implement the driver operations:** The driver code needs to implement several operations, such as **open**, **release**, **read**, **write**, and **ioctl**, to handle interactions with the device. These functions are defined in a **file_operations** structure and associated with your device using the **cdev_init()** and **cdev_add()** functions.
5. **Register the driver:** Use the **register_chrdev** or **alloc_chrdev_region** function to register your driver with the kernel. This step associates your driver's operations with the major number and makes the device file accessible to user-space applications.
6. **Create a device file:** Use the **mknod** command or the **udev** subsystem to create a device file entry in the **/dev** directory. This step allows user space applications to access the device through the file interface.
7. **Build and load the driver module:** Compile your driver source code into a loadable kernel module (.ko file) using the appropriate Makefile. Load the module into the kernel using the **insmod** command or automatically via **modprobe**.
8. **Test and debug:** Write user space test applications that interact with your driver through the device file. Use kernel logs (**dmesg** command) and kernel debugging tools to diagnose and fix any issues that

Email: info@motionzen.com

arise.

9. Clean up and unload the module: When your driver is no longer needed, unload it from the kernel using the **rmmod** command or **modprobe –r** <module_name>.

Important Structures:

1. file_operations: This structure defines the file operations supported by your driver, such as open(), read(), write(), release(), llseek(), and unlocked_ioctl(). and others. It is defined in <linux/fs.h>.
2. cdev: This structure represents a character device and is used to associate file operations with a device. It contains the file operations structure and other device-specific data. It is defined in <linux/cdev.h>.
3. struct file: This structure represents an open file descriptor and contains information about the opened file, such as the current file position, flags, and other details. It is defined in <linux/fs.h>.

Important Kernel APIs:

1. register_chrdev: This function is used to register a character device and obtain a major number. It takes the major number as a parameter and returns 0 on success. It is defined in <linux/fs.h>.
2. alloc_chrdev_region: This function is used to dynamically allocate a range of character device numbers. It assigns a major number to your driver. It is defined in <linux/fs.h>.
3. unregister_chrdev : This function unregisters a character device and frees the major number allocated during registration. It is defined in <linux/fs.h>.
4. unregister_chrdev_region : This function unregisters a character device and frees the major number allocated during registration. It is defined in <linux/fs.h>.
5. cdev_init: This function initializes a cdev structure with the specified file operations. It is typically called before adding the device to the system. It is defined in <linux/cdev.h>.
6. cdev_add: This function adds a cdev structure to the system, making the associated file operations available for the device. It takes the device structure and the device number (major and minor) as parameters. It is defined in <linux/cdev.h>.
7. copy_to_user and copy_from_user: These functions are used to copy data between user space and kernel space. They ensure proper memory access and handle memory alignment. They are defined in <linux/uaccess.h>.
8. kmalloc and kfree: These functions are used for dynamic memory allocation and deallocation in the kernel. They are similar to malloc and free in user space. They are defined in <linux/slab.h>.
9. mutex_lock and mutex_unlock: These functions provide mutual exclusion and are used to protect critical sections of code from concurrent access. They are defined in <linux/mutex.h>.

These are just a few examples of the important structures and kernel APIs used in Linux character device driver development. The Linux kernel provides a rich set of functions and data structures to interact with devices and manage I/O operations. You can refer to the Linux kernel documentation, specifically the header files mentioned

above, for more details on these structures and APIs, as well as other relevant functions and data structures.

Here are some common uses and use cases of character device drivers in Linux:

1. Interfacing with hardware devices: Character device drivers allow the Linux kernel to interact with hardware devices at the character level. This includes devices such as serial ports, terminals, printers, sound cards, and other peripherals that transmit and receive data in a stream of characters. By implementing the appropriate file operations, character device drivers enable user space applications to communicate with these devices through the file interface.
2. Terminal emulation: Character device drivers are used to emulate virtual terminals or consoles. These drivers provide the functionality to read keyboard input and display output on a virtual terminal. They are crucial for text-based interfaces and console-based applications.
3. Serial communication: Character device drivers are commonly used for serial communication with devices connected via serial ports, such as RS-232, UART, or USB-to-serial converters. They handle the low-level communication protocols and provide an interface for applications to read and write data from and to the serial port.
4. Input/output devices: Character device drivers are used for input/output devices that operate at the character level, such as keyboards, mice, and touchpads. These drivers handle the device-specific protocols and provide a standardized interface for reading input events and controlling the device.
5. Virtual devices: Character device drivers are used to create virtual devices that do not have a physical counterpart. These devices can be used for various purposes, such as creating virtual file systems, simulating hardware devices for testing or debugging, or implementing inter-process communication mechanisms.
6. Device control and configuration: Character device drivers are responsible for controlling and configuring hardware devices. They provide an interface for applications to send control commands, retrieve device status, and configure device-specific parameters.
7. User space interfaces: Character device drivers allow user space applications to interact with the kernel and access device functionality without directly manipulating hardware registers or implementing complex communication protocols. They provide a standardized and simplified interface for application developers.
8. Data streaming and processing: Character device drivers are used for devices that stream data in a continuous flow, such as audio or video capture devices. These drivers handle the data transfer and provide an interface for applications to process and manipulate the streamed data.

In Linux, character device drivers use major and minor numbers to uniquely identify and manage device files associated with a driver.

Major Number:

- The major number identifies the driver or the driver family that controls a particular device.

- The major number is a non-negative integer.
- It is assigned dynamically or statically when registering the driver.
- Major 0 – Unnamed devices (e.g. non-device mounts)
- Character devices that request a dynamic allocation of major number will take numbers starting from 511 and downward, once the 234-254 range is full

Minor Number:

- The minor number is used to differentiate between multiple devices controlled by the same driver.
- It represents a specific instance or a specific device within a driver's family.
- The minor number is a non-negative integer.
- Some special minor numbers are reserved for specific purposes. For example, minor number 0 is typically reserved for the first device instance.

Please consult Documentation/admin-guide/devices.txt

In Linux, the **dev_t type** is used to represent device numbers, which consists of major and minor numbers combined into a single 32-bit value. The dev_t type is typically defined as an unsigned 32-bit integer. Within the 32 bits of dev_t, the allocation of bits for major and minor numbers can vary depending on the system configuration and requirements.

By convention, the higher 12 bits are usually reserved for the major number, while the lower 20 bits are used for the minor number. However, it's important to note that these conventions can differ depending on the specific Linux distribution or configuration.

To extract the major and minor numbers from a dev_t value, you can use the following macros:

- MAJOR(dev): This macro extracts the major number from a dev_t value.
- MINOR(dev): This macro extracts the minor number from a dev_t value.

Here's an example of how you can use these macros:

```
dev_t device_number = MKDEV(250, 3);  // Example dev_t value
unsigned int major_number = MAJOR(device_number);
unsigned int minor_number = MINOR(device_number);
```

In this example, major_number would be assigned the value 250, and minor_number would be assigned the value 3.

It's important to note that the specific bit allocation and range for major and minor numbers can vary based on the system configuration and the specific Linux distribution being used. To ensure compatibility, it is generally recommended to use the provided macros (MAJOR and MINOR) rather than making assumptions about the specific bit allocation.

Please note that developing a character device driver requires a good understanding of the Linux kernel and

driver development concepts. It's recommended to refer to relevant documentation, such as the Linux Device Drivers (LDD3) book or the kernel's official documentation, to learn more about the specific details and best practices for writing Linux device drivers.

Sample char driver code :

```
$ cat  mydevice.c

#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

#define DEVICE_NAME "mydevice"
#define EXAMPLE_MSG "Hello from mydevice\n"
#define MSG_BUFFER_LEN 16

static int major;
static char msg_buffer[MSG_BUFFER_LEN];
static int msg_size;

static int mydevice_open(struct inode *inode, struct file *file)
{
    pr_info("mydevice: device opened\n");
    return 0;
}

static int mydevice_release(struct inode *inode, struct file *file)
{
    pr_info("mydevice: device released\n");
    return 0;
}

static ssize_t mydevice_read(struct file *file, char __user *buffer, size_t len, loff_t *offset)
{
    int bytes_read = 0;

    if (*offset >= msg_size)
        return 0;
```

```
    if (*offset + len > msg_size)
        len = msg_size - *offset;

    bytes_read = raw_copy_to_user(buffer, msg_buffer + *offset, len);
    *offset += len;

    pr_info("mydevice: device read\n");

    return len - bytes_read;
}

static ssize_t mydevice_write(struct file *file, const char __user *buffer, size_t len, loff_t *offset)
{
    int bytes_written = 0;

    if (len > MSG_BUFFER_LEN)
        return -EINVAL;

    bytes_written = raw_copy_from_user(msg_buffer, buffer, len);
    msg_size = len - bytes_written;

    pr_info("mydevice: device write\n");

    return len - bytes_written;
}

static struct file_operations mydevice_fops = {
    .open = mydevice_open,
    .release = mydevice_release,
    .read = mydevice_read,
    .write = mydevice_write,
};

static int __init mydevice_init(void)
{
    major = register_chrdev(0, DEVICE_NAME, &mydevice_fops);

    if (major < 0) {
        pr_err("Failed to register character device\n");
        return major;
    }

    pr_info("mydevice: device registered, major number: %d\n", major);
```


Email: info@motionzen.com

```
    return 0;
}

static void __exit mydevice_exit(void)
{
    unregister_chrdev(major, DEVICE_NAME);
    pr_info("mydevice: device unregistered\n");
}

module_init(mydevice_init);
module_exit(mydevice_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple character driver for hardware I/O");
```

```
$ make -C /lib/modules/$(uname -r)/build M= $(pwd) modules

$ sudo insmod mydevice.ko
mydevice: device registered, major number: 237

$ mknod /dev/mydevice c 237 0

$cat /dev/mydevice

$echo "Hello char driver" > /dev/mydevice

$cat /dev/mydevice
Hello char driver
```

BY

SATEESH KUMAR G