**Device Driver for A Simulated Character**

**Device Driver**

|  |  |
| --- | --- |
| 1. Document Information | |
| Document Name | Device Driver for a simulated character device driver |
| Author | GROUP - A |
| Contributors | Dhriti  Susmitha  Sravani  Srinivas  Badrinath |

OBJECTIVE :

* To build a Device Driver for a simulated Character Device Driver

**Problem Statement :**

* You are required to write a character device driver for an x86 system.
* The driver should interact with a user-space program to perform the following tasks:

* Create a Character Device Driver in the Kernel Space:
* Implement open, release and ioctl functions for the device driver.
* When the user takes input of two numbers as num1, num2, then the driver asks the operation (+,-,\*,/) .
* When the user reads from the device, the result will be displayed as opted operand.
* **Write a User-Space Program:**
* Accept two numbers num1, num2 as input from user.
* Accept the preferred operation from the user.
* Write the inputs to the device by using ioctl command.
* Read the result from device by using ioctl command.

**Requirements:**

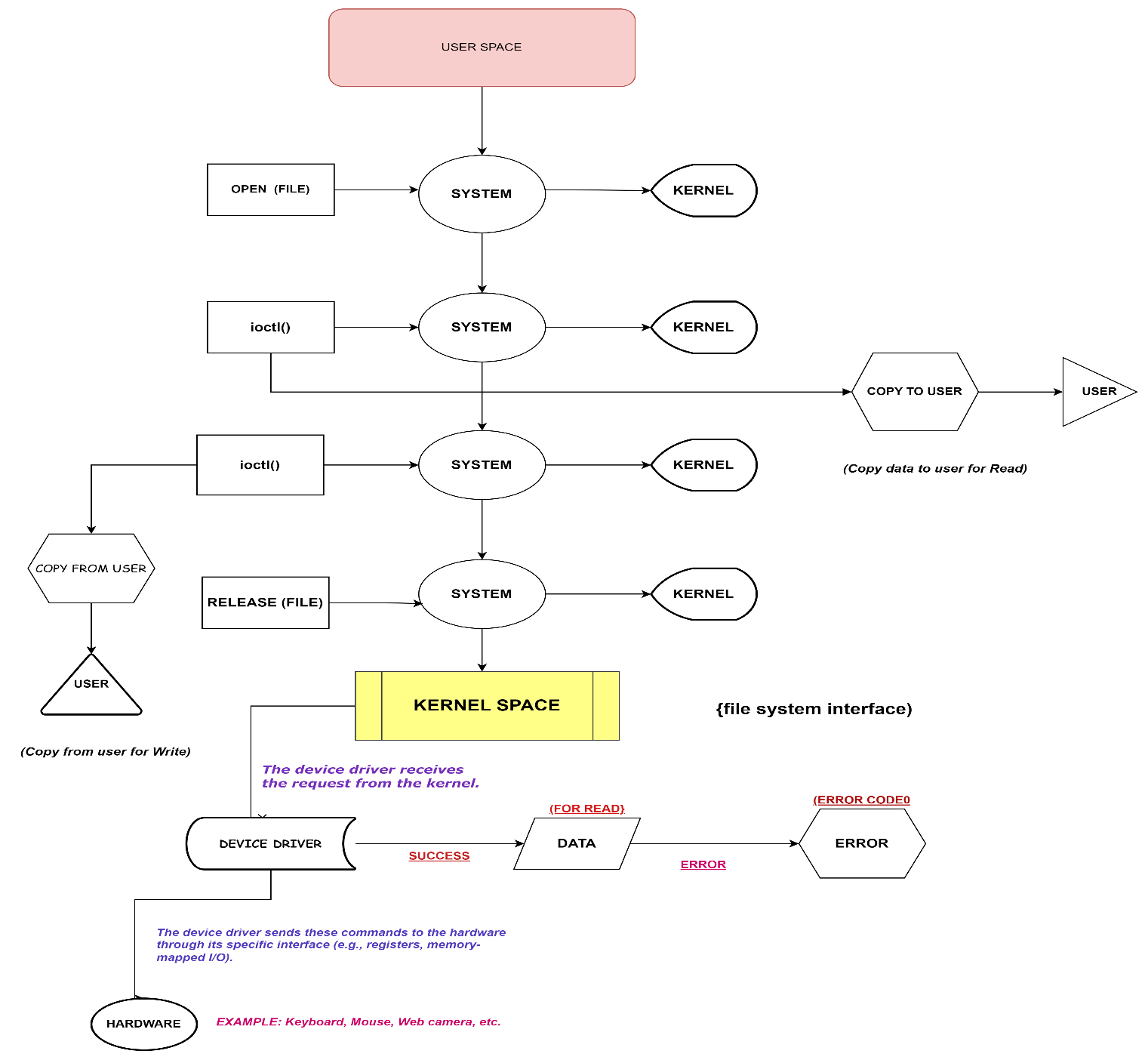
* Kernel Module (Driver):
* Implement the device driver with necessary functions

(open, ioctl, release).

* Do the operation requested by the user.
* Return the result to the user space.
* User-Space Program:
* Open the character device.
* Give inputs to the device.
* Read the result from the device.
* Print the result.

**Block Diagram:**

**Kernel user space block diagram:**

****

The block diagram illustrates the interaction between the user-space program and the kernel-space driver for Character device driver. Here's an explanation of each component:

 **Device File**:

* **User Space**: The user space program accesses the device through a special file (e.g., /dev/my\_calculator\_device). This file represents the device and allows the user program to interact with the driver using standard file operations.
* **Kernel Space**: The kernel associates this file with the device driver, which handles the file operations on behalf of the user space program.

1. **File Operations Structure (struct file\_operations)**:
   * This structure in the kernel module defines the functions that handle the various file operations (open, read, write, ioctl, close) for the device

static struct file\_operations fops = {

.owner = THIS\_MODULE,

.open = device\_open,

.release = device\_release,

.unlocked\_ioctl = device\_ioctl,

};

1. **Device Registration**:
   * **Kernel Space**: The device is registered with the kernel using functions like register\_chrdev or cdev\_add. This associates the device file with the file operations structure

major\_number = register\_chrdev(0, DEVICE\_NAME, &fops);

1. **Open and Close Operations**:
   * **User Space**: The user program opens the device file using the open system call and closes it using the close system call.

int fd = open("/dev/my\_calculator\_device", O\_RDWR);

close(fd);

* + **Kernel Space**: Corresponding open and release functions in the kernel module handle these operations.

static int device\_open(struct inode \*inode, struct file \*file) {

printk(KERN\_INFO "Calculator device opened\n");

return 0;

}

static int device\_release(struct inode \*inode, struct file \*file) {

printk(KERN\_INFO "Calculator device closed\n");

return 0;

}

1. **Ioctl (Input/Output Control)**:
   * **User Space**: The user program sends commands and data to the device using the ioctl system call.

ioctl(fd, IOCTL\_CALC, &data);

* + **Kernel Space**: The unlocked\_ioctl function in the kernel module handles these commands. This function can perform various device-specific operations and transfer data between user space and kernel space.

static long device\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg) {

struct calc\_data data;

switch (cmd) {

case IOCTL\_CALC:

if (copy\_from\_user(&data, (struct calc\_data \_\_user \*)arg, sizeof(data))) {

return -EFAULT;

}

// Perform calculation

if (copy\_to\_user((struct calc\_data \_\_user \*)arg, &data, sizeof(data))) {

return -EFAULT;

}

break;

default:

return -ENOTTY;

}

return 0;

}

1. **Data Structures and Copy Functions**:
   * **Data Structures**: Data exchanged between user space and kernel space is often encapsulated in structures. For example, the calc\_data structure holds the operands, operation, and result.

struct calc\_data {

int num1;

int num2;

char operation;

int result;

};

**Copy Functions**: Data is copied between user space and kernel space using functions like copy\_from\_user and copy\_to\_user.

if (copy\_from\_user(&data, (struct calc\_data \_\_user \*)arg, sizeof(data))) {

return -EFAULT;

}

if (copy\_to\_user((struct calc\_data \_\_user \*)arg, &data, sizeof(data))) {

return -EFAULT;

}

1. **Kernel Logging**:
   * **Kernel Space**: The kernel module can log messages to the kernel log buffer using functions like printk, which helps in debugging and monitoring the driver's behavior.

c

Copy code

printk(KERN\_INFO "Calculator device opened\n");

printk(KERN\_INFO "Operation %c: %d %c %d = %d\n", data.operation, data.num1, data.operation, data.num2, data.result);

1. **Error Handling**:
   * **Kernel Space**: The kernel module must handle errors appropriately, returning standard error codes (e.g., -EFAULT, -EINVAL, -ENOTTY) when operations fail. This ensures that the user program can detect and respond to errors.

c

Copy code

if (data.num2 == 0 && data.operation == '/') {

return -EINVAL;

}

**Flow Summary**:

1. **Initialization**: The kernel module initializes the device, registers it, and sets up file operations.
2. **User Interaction**: The user program opens the device file, sends data and commands via ioctl, and closes the device file.
3. **Ioctl Handling**: The kernel module processes the ioctl command, performs the necessary calculations, and copies the result back to user space.
4. **Completion**: The user program reads the result and displays it.

By following these components and flow, the user space and kernel space communicate effectively, enabling user programs to interact with hardware or virtual devices managed by kernel modules.

Kernel space program :

#include <linux/init.h> // Required for module initialization and exit macros

#include <linux/module.h> // Required for module-related macros and functions

#include <linux/fs.h> // Required for file operations structure

#include <linux/uaccess.h> // Required for copy\_to\_user and copy\_from\_user functions

#include <linux/cdev.h> // Required for character device registration

#define DEVICE\_NAME "my\_calculator\_device" // Define the name of the device

#define IOCTL\_MAGIC 'c' // Define a magic number for ioctl

#define IOCTL\_CALC \_IOWR(IOCTL\_MAGIC, 1, struct calc\_data) // Define ioctl command with magic number and command number

// Define a structure to hold the calculation data

struct calc\_data {

int num1; // First operand

int num2; // Second operand

char operation; // Operation to perform (+, -, \*, /)

int result; // Result of the operation

};

// Function to handle device open

static int device\_open(struct inode \*inode, struct file \*file) {

printk(KERN\_INFO "Calculator device opened\n"); // Log message

return 0; // Return success

}

// Function to handle device close

static int device\_release(struct inode \*inode, struct file \*file) {

printk(KERN\_INFO "Calculator device closed\n"); // Log message

return 0; // Return success

}

// Function to handle ioctl calls

static long device\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg) {

struct calc\_data data; // Declare a structure to hold calculation data

switch (cmd) { // Check which ioctl command was received

case IOCTL\_CALC: // If the command is IOCTL\_CALC

if (copy\_from\_user(&data, (struct calc\_data \_\_user \*)arg, sizeof(data))) { // Copy data from user space

return -EFAULT; // Return error if copy fails

}

// Perform the requested operation

switch (data.operation) {

case '+':

data.result = data.num1 + data.num2; // Addition

break;

case '-':

data.result = data.num1 - data.num2; // Subtraction

break;

case '\*':

data.result = data.num1 \* data.num2; // Multiplication

break;

case '/':

if (data.num2 == 0) { // Check for division by zero

return -EINVAL; // Return invalid argument error

}

data.result = data.num1 / data.num2; // Division

break;

default:

return -EINVAL; // Return invalid argument error for unknown operations

}

if (copy\_to\_user((struct calc\_data \_\_user \*)arg, &data, sizeof(data))) { // Copy result back to user space

return -EFAULT; // Return error if copy fails

}

// Log the operation performed

printk(KERN\_INFO "Operation %c: %d %c %d = %d\n", data.operation, data.num1, data.operation, data.num2, data.result);

break;

default:

return -ENOTTY; // Return "inappropriate ioctl" error for unknown commands

}

return 0; // Return success

}

// Define the file operations structure

static struct file\_operations fops = {

.owner = THIS\_MODULE, // Set the owner of the module

.open = device\_open, // Set the open function

.release = device\_release, // Set the release function

.unlocked\_ioctl = device\_ioctl, // Set the ioctl function

};

static int major\_number; // Declare a variable to hold the major number

static struct cdev my\_cdev; // Declare a character device structure

// Module initialization function

static int \_\_init my\_calculator\_init(void) {

major\_number = register\_chrdev(0, DEVICE\_NAME, &fops); // Register the character device and get the major number

if (major\_number < 0) { // Check if registration failed

printk(KERN\_ALERT "Registering char device failed with %d\n", major\_number); // Log error message

return major\_number; // Return the error code

}

cdev\_init(&my\_cdev, &fops); // Initialize the character device with file operations

my\_cdev.owner = THIS\_MODULE; // Set the owner of the character device

if (cdev\_add(&my\_cdev, MKDEV(major\_number, 0), 1)) { // Add the character device to the system

unregister\_chrdev(major\_number, DEVICE\_NAME); // Unregister the character device if adding fails

printk(KERN\_ALERT "Adding cdev failed\n"); // Log error message

return -1; // Return error code

}

printk(KERN\_INFO "Calculator device registered with major number %d\n", major\_number); // Log success message

return 0; // Return success

}

// Module exit function

static void \_\_exit my\_calculator\_exit(void) {

cdev\_del(&my\_cdev); // Delete the character device

unregister\_chrdev(major\_number, DEVICE\_NAME); // Unregister the character device

printk(KERN\_INFO "Calculator device unregistered\n"); // Log success message

}

module\_init(my\_calculator\_init); // Set the initialization function

module\_exit(my\_calculator\_exit); // Set the exit function

MODULE\_LICENSE("GPL"); // Set the license of the module

MODULE\_AUTHOR("Group A"); // Set the author of the module

MODULE\_DESCRIPTION("A simple calculator char driver with ioctl"); // Set the description of the module

MODULE\_VERSION("0.1"); // Set the version of the module

Steps for creating kernel space program :

The code you provided is an example of a simple character device driver in the Linux kernel, specifically designed to function as a calculator using ioctl commands. Let's break down the steps involved in creating such a kernel module:

**1.Include Necessary Header Files**

#include <linux/init.h>

#include <linux/module.h>

#include <linux/fs.h>

#include <linux/uaccess.h>

#include <linux/cdev.h>

These headers include necessary definitions and functions for module initialization, file operations, user-space access, and character device management (cdev).

**2. Define Constants and Structures**

#define DEVICE\_NAME "my\_calculator\_device"

#define IOCTL\_MAGIC 'c'

#define IOCTL\_CALC \_IOWR(IOCTL\_MAGIC, 1, struct calc\_data)

struct calc\_data {

int num1;

int num2;

char operation;

int result;

};

DEVICE\_NAME: Defines the name of the device.

IOCTL\_MAGIC and IOCTL\_CALC: Defines a magic number and an ioctl command for performing calculations.

struct calc\_data: Structure to hold operands, operation, and result of a calculation.

**3. Define File Operation Handlers**

static int device\_open(struct inode \*inode, struct file \*file);

static int device\_release(struct inode \*inode, struct file \*file);

static long device\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg);

static struct file\_operations fops = {

.owner = THIS\_MODULE,

.open = device\_open,

.release = device\_release,

.unlocked\_ioctl = device\_ioctl,

};

device\_open: Called when the device file is opened.

device\_release: Called when the device file is closed.

device\_ioctl: Handles ioctl commands for performing calculations.

**4. Implement File Operation Functions**

static int device\_open(struct inode \*inode, struct file \*file) {

// Implementation of device open

}

static int device\_release(struct inode \*inode, struct file \*file) {

// Implementation of device release

}

static long device\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg) {

// Implementation of ioctl handling

}

These functions implement opening, closing, and handling ioctl commands respectively. The device\_ioctl function performs calculations based on the received ioctl command (IOCTL\_CALC).

**5. Initialize and Register the Character Device**

static int major\_number;

static struct cdev my\_cdev;

static int \_\_init my\_calculator\_init(void) {

major\_number = register\_chrdev(0, DEVICE\_NAME, &fops);

if (major\_number < 0) {

// Error handling if registration fails

}

cdev\_init(&my\_cdev, &fops);

// Other initialization steps

}

register\_chrdev: Registers the character device and obtains a major number.

cdev\_init and cdev\_add: Initializes and adds the character device to the system.

**6. Module Initialization and Exit Functions**

module\_init(my\_calculator\_init);

module\_exit(my\_calculator\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Group A");

MODULE\_DESCRIPTION("A simple calculator char driver with ioctl");

MODULE\_VERSION("0.1");

module\_init: Specifies the initialization function (my\_calculator\_init).

module\_exit: Specifies the exit function (my\_calculator\_exit).

MODULE\_LICENSE, MODULE\_AUTHOR, MODULE\_DESCRIPTION, MODULE\_VERSION: Metadata about the module.

* The provided code outlines the creation of a character device driver that acts as a simple calculator. It registers itself as a character device, implements open/close/ioctl operations, and performs arithmetic operations based on user commands. This example showcases how to interact with user-space applications through ioctl commands to perform device-specific operations. Remember, kernel development requires careful handling of memory, synchronization, and error checking to ensure stability and security.

User-space program:

#include <stdio.h> // For standard input/output functions

#include <stdlib.h> // For standard library functions, including EXIT\_FAILURE and EXIT\_SUCCESS

#include <fcntl.h> // For file control options (open function)

#include <unistd.h> // For close function

#include <sys/ioctl.h> // For ioctl function

#define DEVICE\_PATH "/dev/my\_calculator\_device" // Path to the calculator device

#define IOCTL\_MAGIC 'c' // Magic number for ioctl

#define IOCTL\_CALC \_IOWR(IOCTL\_MAGIC, 1, struct calc\_data) // ioctl command definition

// Define a structure to hold the calculation data

struct calc\_data {

int num1; // First operand

int num2; // Second operand

char operation; // Operation to perform (+, -, \*, /)

int result; // Result of the operation

};

int main() {

int fd; // File descriptor for the device

struct calc\_data data; // Structure to hold user input and result

// Open the device for reading and writing

fd = open(DEVICE\_PATH, O\_RDWR);

if (fd < 0) { // Check if the device failed to open

perror("Failed to open the device"); // Print error message

return EXIT\_FAILURE; // Return failure status

}

// Prompt user for input and read the values

printf("Enter first number: ");

scanf("%d", &data.num1); // Read first number

printf("Enter second number: ");

scanf("%d", &data.num2); // Read second number

printf("Enter operation (+, -, \*, /): ");

scanf(" %c", &data.operation); // Read operation (note the space to handle newline)

// Perform ioctl to send data to the kernel module and get the result

if (ioctl(fd, IOCTL\_CALC, &data) < 0) { // Check if ioctl failed

perror("Failed to execute ioctl"); // Print error message

close(fd); // Close the device

return EXIT\_FAILURE; // Return failure status

}

// Print the result of the calculation

printf("Result: %d %c %d = %d\n", data.num1, data.operation, data.num2, data.result);

close(fd); // Close the device

return EXIT\_SUCCESS; // Return success status

}

Steps for creating user-space program:

The user-space program you provided interacts with a kernel-space device driver (my\_calculator\_device) through ioctl commands to perform arithmetic calculations. Let's break down the steps involved in creating and understanding this program:

**1. Include Necessary Header Files**

#include <stdio.h> // For standard input/output functions

#include <stdlib.h> // For standard library functions, including EXIT\_FAILURE and EXIT\_SUCCESS

#include <fcntl.h> // For file control options (open function)

#include <unistd.h> // For close function

#include <sys/ioctl.h> // For ioctl function

These headers include functions and constants needed for file operations (open, close), handling errors (perror), and performing ioctl operations.

**2. Define Constants**

#define DEVICE\_PATH "/dev/my\_calculator\_device" // Path to the calculator device

#define IOCTL\_MAGIC 'c' // Magic number for ioctl

#define IOCTL\_CALC \_IOWR(IOCTL\_MAGIC, 1, struct calc\_data) // ioctl command definition

DEVICE\_PATH: Path to the character device file (/dev/my\_calculator\_device) provided by the kernel module.

IOCTL\_MAGIC: A magic number used for ioctl operations to uniquely identify them.

IOCTL\_CALC: An ioctl command definition that specifies the direction, command number, and the structure (struct calc\_data) used for communication.

**3. Define Structure for Calculation Data**

struct calc\_data {

int num1; // First operand

int num2; // Second operand

char operation; // Operation to perform (+, -, \*, /)

int result; // Result of the operation

};

struct calc\_data: Holds operands (num1, num2), operation (operation), and result (result) of the calculation.

**4. Main Function Implementation**

int main() {

int fd; // File descriptor for the device

struct calc\_data data; // Structure to hold user input and result

// Open the device for reading and writing

fd = open(DEVICE\_PATH, O\_RDWR);

if (fd < 0) { // Check if the device failed to open

perror("Failed to open the device"); // Print error message

return EXIT\_FAILURE; // Return failure status

}

// Prompt user for input and read the values

printf("Enter first number: ");

scanf("%d", &data.num1); // Read first number

printf("Enter second number: ");

scanf("%d", &data.num2); // Read second number

printf("Enter operation (+, -, \*, /): ");

scanf(" %c", &data.operation); // Read operation (note the space to handle newline)

// Perform ioctl to send data to the kernel module and get the result

if (ioctl(fd, IOCTL\_CALC, &data) < 0) { // Check if ioctl failed

perror("Failed to execute ioctl"); // Print error message

close(fd); // Close the device

return EXIT\_FAILURE; // Return failure status

}

// Print the result of the calculation

printf("Result: %d %c %d = %d\n", data.num1, data.operation, data.num2, data.result);

close(fd); // Close the device

return EXIT\_SUCCESS; // Return success status

}

* **Opening the Device**:
* open(DEVICE\_PATH, O\_RDWR) opens the character device file (/dev/my\_calculator\_device) for reading and writing (O\_RDWR). If unsuccessful (fd < 0), it prints an error message using perror and exits with EXIT\_FAILURE.
* **Reading User Input:**
* Prompts the user to enter two numbers and an arithmetic operation (+, -, \*, /). These values are stored in data.num1, data.num2, and data.operation.
* **Performing ioctl Operation:**
* ioctl(fd, IOCTL\_CALC, &data) sends the structure data to the kernel module through the character device file descriptor fd. The kernel module performs the specified arithmetic operation and stores the result back in data.result.
* **Displaying the Result**:
* After receiving the result from the kernel module, it prints the operation and the calculated result using printf.
* **Closing the Device**:
* close(fd) closes the device file descriptor to release resources.
* This user-space program demonstrates how to communicate with a custom kernel module (my\_calculator\_device) using ioctl commands. It reads user input for numbers and an arithmetic operation, sends this data to the kernel for processing, receives the result, and then displays it to the user. This interaction showcases the use of ioctl for bidirectional communication between user-space and kernel-space in Linux.
* **Error Handling**:
* Uses perror() to print descriptive error messages along with the error number obtained from errno in case of file operations failure.

Make file :

* This is a simple Makefile used to build Linux kernel modules.

obj-m += cal.o

all :

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules clean :

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

* This obj-m += cal.o line indicates that the kernel module to be built is named cal.o. The obj-m variable is used by the kernel build system to determine which object files to build.

* This rule, named all, specifies how to build the module. It invokes make with the -C option to change directory to the kernel source directory (specified by /lib/modules/$(shell uname -

r)/build) and passes the M parameter to that Makefile. This instructs the kernel build system to use the current directory ($(PWD)) to find the source files for the module. The modules target tells make to build all kernel modules specified in the Makefile.

* This rule, named clean, specifies how to clean up the build artifacts. Similar to the all rule, it uses make to change directory to the kernel source directory and then executes the clean target, which removes all built files and temporary build artifacts.

* After make command we create cal.ko which is kernel object file that is used to build kernel understandable code.

* These .ko module file is inserted into kernel by usind insmod command.

**Switchng to user :**

* + By using sudo su command we able to switch into kernel space and we need to enter password to switch user space to kernel.



**Inserting module into kernel :**

* + Sudo insmod cal.ko // inserts module into kernel
  + Sudo dmesg | tail //”display message” it is used to print the message from kernel log.

**Creating special file using mknod :**

* + The character special file is used by user-space with kernel space to get information from hardware.
  + Sudo mknod /dev/my\_calculator\_device c 241 0
  + sudo: This command is used to execute the subsequent command with superuser privileges.
  + mknod: This is a command used to create device nodes (special files) in Unix-like operating systems.
  + /dev/my\_calculator\_device: This specifies the path and name of the device node to be created. In this case, it creates a device node named "my\_calculator\_device" in the /dev directory.
  + c: This flag indicates that the device node to be created is a character device
  + 241: This is the major number assigned to the device. Major numbers are used by the kernel to identify device drivers.
  + 0: This is the minor number assigned to the device. Minor numbers are used to identify specific devices managed by the same driver.
  + After executing this command, the device node

/dev/my\_calculator\_device will be created,

allowing user space programs to interact with the corresponding kernel module.

User-space make :

* By using make user command we create object file for user-space that makes understand to operating system.
* By using ./user command the user and kernel work with each other by transferring data. Here ./user is executable file.

OUTPUT :

* In the output we will be ask to enter the message to get reverse from userspace. And prints the result to console .
* Enter first number:3
* Enter second number:5
* Enter operation(+ , - , \* , / ): +
* Result : 3 + 5 = 8

Clean up instead modules :

➢ By using rmmod command we can exit or cleans the inserted module which makes kernel environment free of conflicts while working with other files.

Output:

