Character Device Driver with Simulated Sensor Data

Define

Project Goal

The goal is to develop a character device driver for a sensor, allowing user-space applications to interact with the sensor's data. This project helps to demonstrate my skills in C programming, Linux internals, and device driver development.

Scope

- Develop a character device driver that interfaces with a specific sensor using I2C or SPI.
- Create a user-space application to read data from the sensor via the device driver.
- Ensure the project can be safely executed on a laptop using virtual environments or emulators.

Requirements

- **Hardware**: Sensor (e.g., LM35 temperature sensor or MPU6050 accelerometer), USB to I2C/SPI adapter (if using actual hardware).
- **Software**: Linux environment (preferably in a VM), GCC, make, kernel headers, and libraries for I2C/SPI communication.
- **Skills**: C programming, Linux kernel programming, basic understanding of I2C/SPI protocols.

Design

System Architecture

- **Sensor**: Provides data via I2C/SPI.
- **Device Driver**: A kernel module that handles communication with the sensor and provides a character device interface.
- **User-Space Application**: A simple C program that interacts with the device driver to read sensor data.

Module Components

- **Initialization and Cleanup Functions**: Load and unload the driver.
- **File Operations Structure**: Define `open`, `read`, `write`, and `release` functions.
- **I2C/SPI Communication**: Functions to communicate with the sensor.

- **Device Node**: Create a device node in `/dev` for user-space interaction.

Data Flow

1. **User-Space Application** requests data by opening the device file.
2. The **Device Driver** communicates with the sensor to fetch data.
3. The **Sensor** sends data to the driver, which is then passed to the application.

Development

Setting Up the Environment

1. **Install Virtual Machine (VM)**: Use VirtualBox or VMware to set up a Linux VM.
2. **Install Development Tools**: Inside the VM, install GCC, make, and kernel headers.
3. **Prepare Kernel Source**: If modifying or building kernel modules, ensure the source is available and configured.

Writing the Device Driver

1. **Module Initialization**: Write the module's `init` and `exit` functions.
2. **File Operations**: Implement the necessary file operations:
    - `open()`: Initialize any necessary data structures.
    - `read()`: Communicate with the sensor to read data.
    - `write()`: (Optional) If the sensor allows configuration.
    - `release()`: Clean up resources.
3. **Sensor Communication**: Implement functions for I2C/SPI communication to interact with the sensor.
4. **Device Node Creation**: Use `mknod` or `udev` rules to create the device file.

User-Space Application

1. **Open the Device File**: Use `open()` system call.
2. **Read Data**: Use `read()` to fetch data from the driver.
3. **Display Data**: Print the sensor data to the console.

Testing and Debugging

1. **Load the Module**: Use `insmod` to load the driver and `dmesg` to check for messages.

2. **User-Space Testing**: Run the user-space application to ensure data is correctly read from the sensor.
3. **Debugging**: Use `printk` for kernel debugging and `gdb` for user-space application debugging.

Document

Documentation Structure

1. **Introduction**: Overview of the project, its purpose, and scope.
2. **System Architecture**: Diagrams and explanations of the system components.
3. **Setup and Installation**: Steps to set up the environment and dependencies.
4. **Driver Development**: Detailed explanation of the driver code, including file operations and sensor communication.
5. **User-Space Application**: Explanation of the application code and how it interacts with the driver.
6. **Testing and Results**: Test cases, results, and any issues encountered.
7. **Conclusion**: Summary of what was learned, any potential improvements, and future work.

Character Device Driver with Simulated Sensor Data

1. Project Overview

1.1. Objective

The objective of this project is to create a Linux kernel character device driver that simulates sensor data. The driver provides a device file in the `/dev` directory that user-space applications can interact with to read simulated data.

1.2. Scope

- Develop a character device driver that generates random data to simulate sensor readings.
- Implement basic file operations (`open`, `read`, `write`, `release`) for the driver.
- Create a user-space application to interact with the driver and retrieve simulated data.

- The project is designed to enhance skills in Linux kernel programming, device driver development, and interfacing between kernel and user-space.

2. Design Flow

2.1. Module Initialization

- **Start**
- **Register Character Device**
  - Register the device with a dynamic major number.
  - Define file operations.
- **Create Device Class**
  - Create a device class for the character device.
- **Create Device Node**
  - Create the device node in `/dev` using the obtained major number.
- **Initialization Successful**
  - Log success and proceed to file operations handling.

2.2. File Operations

1. **Open Device File (`dev_open`)**
   - Initialize device-specific data (if necessary).
   - Log that the device has been opened.
2. **Read Data from Device (`dev_read`)**
   - **Generate Simulated Data**
     - Generate random numbers to simulate sensor readings.
   - **Copy Data to User-Space**
     - Copy the simulated data to the user-space buffer provided by the application.
   - **Return Data Length**
     - Return the number of bytes read to the application.
3. **Write Data to Device (`dev_write`)**
   - **Receive User Data**
     - Receive data from the user-space application (optional for configuration commands).
   - **Process User Data**
     - Process or log the received data (optional).
   - **Return Number of Bytes Written**
     - Return the length of data received.

4. **Close Device File (`dev_release`)**
    - **Release Resources**
        - Clean up any resources allocated during the `open` call.
    - **Log Device Closure**
        - Log that the device has been closed.

2.3. Module Cleanup

1. **Start Cleanup**
2. **Destroy Device Node**
    - Remove the device node from `/dev`.
3. **Unregister Device Class**
    - Unregister the device class.
4. **Unregister Character Device**
    - Unregister the character device and release the major number.
5. **Cleanup Successful**
    - Log successful module removal.

2.4. User-Space Application Interaction

1. **Open Device File**
    - The application opens the device file in read-only mode.
2. **Read Data**
    - The application reads data from the device file.
3. **Process Data**
    - The application processes or displays the received data.
4. **Close Device File**
    - The application closes the device file.

3. Implementation

3.1. Module Code

The module code includes:

- **Module Initialization (`sim_sensor_init`)**: Registers the device, creates a class, and creates the device node.
- **File Operations**: Implements `open`, `read`, `write`, and `release` functions.
- **Module Cleanup (`sim_sensor_exit`)**: Destroys the device node, unregisters the class, and unregisters the device.

3.1.1. Code Structure

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/random.h>

#define DEVICE_NAME "sim_sensor"
#define CLASS_NAME "sim"

// Module metadata
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Linux char driver for simulated sensor
data");
MODULE_VERSION("0.1");

static int majorNumber;
static char simulated_data[256] = {0};
static struct class* simClass = NULL;
static struct device* simDevice = NULL;

// File operations prototypes
static int dev_open(struct inode *, struct file *);
static int dev_release(struct inode *, struct file *);
static ssize_t dev_read(struct file *, char *, size_t, loff_t *);
static ssize_t dev_write(struct file *, const char *, size_t, loff_t
*);

// File operations structure
static struct file_operations fops = {
   .open = dev_open,
   .read = dev_read,
   .write = dev_write,
   .release = dev_release,
};

// Module initialization
```

```
static int __init sim_sensor_init(void){
   printk(KERN_INFO "SimSensor: Initializing the SimSensor LKM\n");
   majorNumber = register_chrdev(0, DEVICE_NAME, &fops);
   if (majorNumber < 0){
      printk(KERN_ALERT "SimSensor failed to register a major
number\n");
      return majorNumber;
   }
   simClass = class_create(THIS_MODULE, CLASS_NAME);
   if (IS_ERR(simClass)){
      unregister_chrdev(majorNumber, DEVICE_NAME);
      return PTR_ERR(simClass);
   }
   simDevice = device_create(simClass, NULL, MKDEV(majorNumber, 0),
NULL, DEVICE_NAME);
   if (IS_ERR(simDevice)){
      class_destroy(simClass);
      unregister_chrdev(majorNumber, DEVICE_NAME);
      return PTR_ERR(simDevice);
   }
   return 0;
}

// Module cleanup
static void __exit sim_sensor_exit(void){
   device_destroy(simClass, MKDEV(majorNumber, 0));
   class_unregister(simClass);
   class_destroy(simClass);
   unregister_chrdev(majorNumber, DEVICE_NAME);
   printk(KERN_INFO "SimSensor: Goodbye from the LKM!\n");
}

// Open function
static int dev_open(struct inode *inodep, struct file *filep){
   printk(KERN_INFO "SimSensor: Device has been opened\n");
   return 0;
}

// Read function
```

```
static ssize_t dev_read(struct file *filep, char *buffer, size_t len,
loff_t *offset){
   int random_number;
   get_random_bytes(&random_number, sizeof(random_number));
   random_number = random_number % 100;
   sprintf(simulated_data, "%d\n", random_number);
   int error_count = copy_to_user(buffer, simulated_data,
strlen(simulated_data));
   if (error_count == 0) {
      return strlen(simulated_data);
   }
   else {
      return -EFAULT;
  }
}



// Write function
static ssize_t dev_write(struct file *filep, const char *buffer,
size_t len, loff_t *offset){
   printk(KERN_INFO "SimSensor: Received %zu characters from the
user\n", len);
   return len;
}

// Release function
static int dev_release(struct inode *inodep, struct file *filep){
   printk(KERN_INFO "SimSensor: Device successfully closed\n");
   return 0;
}

module_init(sim_sensor_init);
module_exit(sim_sensor_exit);
```

3.2. User-Space Application

- **Purpose**: Interacts with the driver to read the simulated data.
- **Operations**:
    - Open the device file.
    - Read data from the device.
    - Process and display the data.

3.2.1. Sample User-Space Application Code

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
      int fd;
      char buf[256];
      ssize_t ret;

      fd = open("/dev/sim_sensor", O_RDONLY);
      if (fd < 0) {
      perror("Failed to open the device...");
      return errno;
      }

      ret = read(fd, buf, sizeof(buf)-1);
      if (ret < 0) {
      perror("Failed to read the message from the device.");
      return errno;
      }

      buf[ret] = '\0';
      printf("The received message is: [%s]\n", buf);
      close(fd);
      return 0;
}
```

4. Testing and Validation

4.1. Module Testing

1. **Compile the Module**
    - Use `make` to compile the driver.
2. **Insert the Module**
    - Load the module using `insmod sim_sensor.ko`.
3. **Check Kernel Messages**
    - Use `dmesg` to verify successful initialization.
4. **Create Device Node**
    - Create a device file using `mknod` if not automatically created.

4.2. User-Space Testing

1. **Compile the Application**
    - Compile the user-space application using `gcc`.
2. **Run the Application**
    - Execute the application and verify the output.
3. **Validate Data**
    - Check that the application receives and displays simulated data.

4.3. Cleanup

1. **Remove the Module**
    - Unload the module using `rmmod sim_sensor`.
2. **Delete the Device Node**
    - Remove the device file from `/dev`.
3. **Cleanup Artifacts**
    - Use `make clean` to remove compiled files.

5. Conclusion

The Character Device Driver with Simulated Sensor Data project provides practical experience in Linux kernel programming, particularly in developing character device drivers. The project demonstrates how to interface with user-space applications, handle file operations, and simulate hardware behavior in a controlled environment.