Development of a Generic Linux Device Driver for BeagleBone-based boards

Minor Project Report

Submitted in Partial Fulfillment of the Requirements for the Degree of

BACHELOR OF TECHNOLOGY

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

By

Vedant Rokad (21BEC106)
Deep Lad (21BEC023)

Department of Electronics and Communication Engineering,
Institute of Technology, Nirma University,
Ahmedabad 382481

November 2021

CERTIFICATE

This is to certify that the Minor Project Report submitted by Mr. Deep Lad (21BEC023) towards the partial fulfillment of the requirements for the award of a degree in Bachelor of Technology in the field of Electronics & Communication Engineering of Nirma University is the record of work carried out by him under our supervision and guidance. The work submitted has in our opinion reached a level required for being accepted for examination. The results embodied in this minor project work to the best of our knowledge have not been submitted to any other University or Institution for the award of any degree or diploma.

Dr. Vijay Savani
Project Guide
Department of Electronics &
Communication Engineering,
Institute of Technology,
Nirma University,
Ahmedabad 382481

Dr. Usha Mehta
Head of Department
Department of Electronics &
Communication Engineering,
Institute of Technology,
Nirma University,
Ahmedabad 382481

Date: 11/16/2024

CERTIFICATE

This is to certify that the Minor Project Report submitted by Mr. Vedant Ashokbhai Rokad (21BEC106) towards the partial fulfillment of the requirements for the award of a degree in Bachelor of Technology in the field of Electronics & Communication Engineering of Nirma University is the record of work carried out by him under our supervision and guidance. The work submitted has in our opinion reached a level required for being accepted for examination. The results embodied in this minor project work to the best of our knowledge have not been submitted to any other University or Institution for the award of any degree or diploma.

Dr. Vijay Savani
Project Guide
Department of Electronics &
Communication Engineering,
Institute of Technology,
Nirma University,
Ahmedabad 382481

Dr. Usha Mehta
Head of Department
Department of Electronics &
Communication Engineering,
Institute of Technology,
Nirma University,
Ahmedabad 382481

Date: 11/16/2024

Undertaking for Originality of the Work

We, Deep Lad and Vedant Rokad, Roll No. 21BEC023, 21BEC106, give undertaking that the Minor Project entitled "Development of a Generic Linux Device Driver for BeagleBone-based boards" submitted by us, towards the partial fulfillment of the requirements for the degree of Bachelor of Technology in Electronics and Communication of Nirma University, Ahmedabad 382 481, is the original work carried out by us and we give assurance that no attempt of plagiarism has been made. We understand that in the event of any similarity found subsequently with any other published work or any project report elsewhere; it will result in severe disciplinary action.

Signature of the Student(s)

Date: **11/16/2024**

Place: Ahmedabad

Endorsed by: Dr. Vijay Savani

(Signature of the Guide)

Acknowledgment

Completing this project, titled "Development of a Generic Linux Device Driver for BeagleBone-based Boards," would not have been possible without the support and guidance of several individuals to whom I am deeply indebted.

First and foremost, I would like to extend my sincere appreciation to my project guide, Dr. Vijay Savani - Assistant Professor, Electronics and Communication Engineering Department, Institute of Technology, Nirma University. His mentorship, deep technical knowledge, and constructive feedback were instrumental throughout this project. Dr. Vijay Savani's expertise in embedded systems and dedication to fostering academic growth gave me invaluable insights and direction, enabling me to approach complex challenges confidently and precisely.

I am also profoundly thankful to Mr. Kiran Nayak, Lead Instructor at Fast Bit Embedded Academy, whose course on Embedded Linux on BeagleBone Black was a vital resource in the successful execution of this project. His structured curriculum, practical exercises, and approachable teaching style empowered me with a solid foundation in embedded Linux and device driver development. The skills and knowledge I gained from his course were directly applicable and essential to the development of this driver.

Additionally, I extend my gratitude to the faculty and laboratory staff at Nirma University for providing a conducive learning environment and the necessary technical resources. The access to equipment, materials, and software, along with the supportive academic environment, greatly facilitated my ability to carry out research and experimentation for this project.

Deep Lad (21BEC023)

Vedant Rokad(21BEC106)

Abstract

This report explores the development of a Generic Linux Device Driver for BeagleBone-based boards, focusing on interfacing with the MPU6050 sensor, a 6-axis accelerometer and gyroscope. The project involves understanding the Linux kernel's device driver framework, configuring the BeagleBone Black hardware for development, and implementing a robust driver to enable seamless communication with the MPU6050 sensor over the I2C bus. The work begins with setting up the target and host platforms, preparing the board for booting via SD card, and integrating cross-compilation toolchains for efficient development. Key concepts like Linux kernel modules, user-space and kernel-space interactions, and Makefile integration for module compilation are explored in detail. The report also provides an in-depth examination of the MPU6050's register set, showcasing how low-level register manipulations enable sensor data acquisition. Finally, a user-level application is developed to demonstrate practical interaction with the MPU6050 sensor, validating the driver's functionality. The project highlights the critical role of Linux device drivers in embedded systems, providing a foundation for future work in developing versatile and scalable device drivers for similar platforms. This approach ensures adaptability and reusability across different hardware architectures

INDEX

List of Figures

Nomenclature

RBL – ROM Boot Loader

MLO – Memory Loader

RFS – Root File System

LKM – Linux Kernel Module

LDD – Linux Device Driver

IMU – Inertial Measurement Unit

I2C – Inter-Integrated Circuits

OS – Operating system

Pcd – Pseudo Character Device

1 Introduction

1.1 Prologue

GNU Linux is the most popular operating system in Embedded Systems due to its **open-source nature and stable and efficient kernel size**. Kernel size is an important parameter here, as the embedded system consists of a small memory size. Modern-day embedded applications involve real-time decision-making and higher processing speed which perhaps the calls Operating System to run between the hardware and software. An Operating System provides many functions to the user application running. Some of them include – resources management, I/O management, Inter-process Communication, Task Synchronization, etc.

Operating systems (OS) play a vital role in modern computing by providing an environment where software can interact with hardware in a structured, manageable, and secure way. In embedded systems, the role of an OS becomes even more critical due to the resource constraints and real-time requirements typical of such systems. Unlike general-purpose computing, embedded systems are designed for dedicated tasks, often operating within specific hardware configurations and low power budgets. Here, the OS functions to manage these limited resources efficiently, enabling **multitasking**, **handling interrupts**, and providing **predictable execution times**, which are essential for **real-time applications**. Additionally, embedded OSs support hardware abstraction, allowing developers to work at a higher level without needing to manage low-level hardware specifics.

Device drivers are specialized software components within an OS that enable it to communicate with various hardware peripherals like sensors, actuators, and communication modules. A device driver **acts as a translator**, converting generalized commands from the OS into specific instructions the hardware can execute. This abstraction is especially valuable in embedded systems where various custom and off-the-shelf hardware components may be used. The device driver handles the complexities of hardware communication, enabling applications to interact with devices seamlessly. Without a device driver, the OS would lack the means to understand or control specific hardware functionalities, which would limit the utility of embedded systems in complex or **custom-built applications**

Embedded Linux has emerged as a preferred OS for many embedded platforms, offering the flexibility of Linux combined with optimizations for embedded environments. Unlike full-fledged Linux distributions, embedded Linux is streamlined for low-power and low-memory devices. It provides a lightweight yet robust environment with essential Linux capabilities such as multitasking, memory management, and file system support. Being open-source, embedded Linux provides developers with access to a wide variety of tools, libraries, and community support, making it an ideal choice for embedded development. Embedded Linux also supports a modular approach, allowing developers to customize the OS by removing unnecessary components, which is essential for fitting it within the tight resource limits of many embedded devices. Its compatibility with a wide array of hardware and I/O protocols, including I2C, makes it especially useful for interfacing with sensors and actuators, enabling it to serve as the backbone of complex embedded applications like our project involving the BeagleBone and MPU6050 sensor.

1.2 Motivation

The motivation behind developing a generic Linux device driver for the MPU6050 sensor on a BeagleBone-based board stems from the numerous benefits it offers in terms of hardware integration, control, and educational value. In embedded systems, the need to incorporate custom hardware components or external sensor modules like the MPU6050 accelerometer and gyroscope sensor is common, yet challenging without a dedicated driver. The MPU6050 is widely used in applications requiring motion detection or orientation tracking, such as robotics, drones, and health monitoring systems. However, without a specific device driver, accessing the sensor's data from a Linux-based embedded platform like the BeagleBone is not feasible, as the OS cannot directly communicate with the device. Developing a generic device driver provides a solution, enabling not only the MPU6050 but potentially other similar sensors to be integrated with minimal changes.

Security and access control are also vital motivations. In an embedded Linux environment, device drivers help control access to sensitive hardware resources, allowing only authorized processes to interact with specific devices. This control layer enhances the overall security of the system, particularly in applications where unauthorized access to hardware components could lead to system vulnerabilities or failures. By implementing a dedicated device driver, we create a controlled entry point for accessing the MPU6050's data, ensuring that interactions with the sensor are secure and managed by the OS.

Developing a device driver for the MPU6050 also presents a valuable opportunity to understand the underlying software stack of an embedded system. The driver serves as a bridge between hardware and application software, providing insights into how Linux manages hardware resources, handles communication protocols like I2C, and interfaces with user-space applications. For those involved in embedded systems, this experience provides a deeper understanding of how software abstracts hardware details, allowing developers to access and manipulate hardware at a high level without directly managing low-level intricacies. Additionally, a generic driver framework developed through this project would facilitate hardware expansion for future projects, making it easier to interface with additional I2C devices on the BeagleBone without redesigning the driver from scratch.

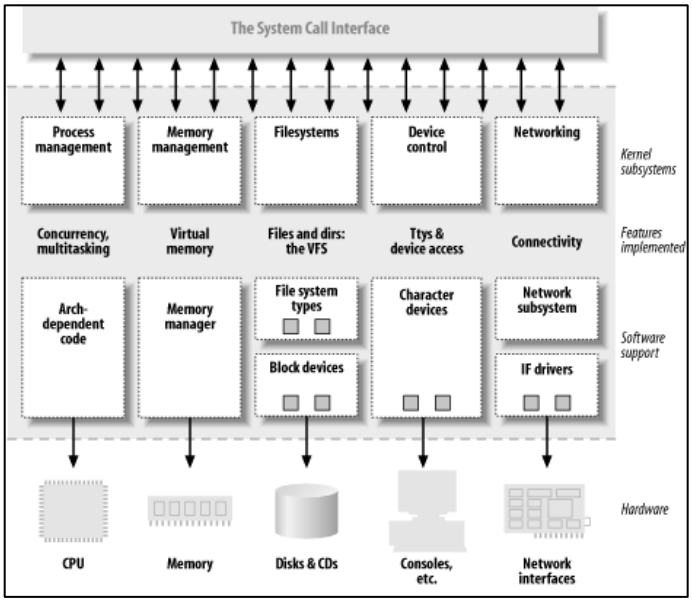

Figure 1.1 High-level architecture diagram of the Linux kernel

1.3 Objective

The primary objective of this project is to develop a generic Linux device driver for the MPU6050 sensor on a BeagleBone-based board using the I2C protocol. This involves creating a driver that can interface with the MPU6050 to gather motion data, process it, and relay it to

the user or application software. The driver should be efficient, robust, and reusable, allowing other developers to interface similar devices with the BeagleBone using minimal modifications.

1.4 Problem Statement

Interfacing custom hardware like the MPU6050 sensor with Linux-based embedded platforms such as the BeagleBone poses challenges due to the need for specific driver software. Without a dedicated driver, the MPU6050 cannot be accessed directly by Linux, limiting the potential applications of the BeagleBone in sensor-based environments. This project aims to bridge this gap by providing a reusable device driver that can simplify interaction with the MPU6050 and potentially other I2C devices.

1.5 Scope of the Project

The scope of this project is extensive and encompasses various aspects of device driver development, hardware-software interfacing, and system performance optimization, specifically focused on creating a generic Linux device driver for the MPU6050 sensor on a BeagleBone-based platform. The project aims to design a modular, reusable driver structure that not only communicates effectively with the MPU6050 via the I2C interface but can also serve as a template for interfacing similar I2C-based sensors or peripherals with minimal modifications. This adaptability is crucial in the embedded systems field, where a variety of sensors and modules often require integration, but existing drivers may not fully support custom hardware configurations.

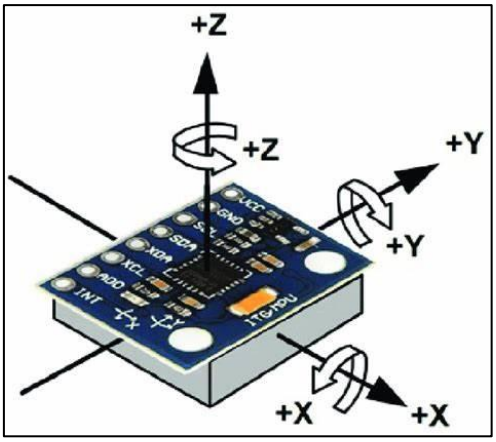

Figure 1.2 IMU Sensor Measurements

Initially, the project involves a deep dive into **understanding the MPU6050** sensor's **architecture**, its **register set**, and **data handling** capabilities. By thoroughly examining its functionalities, such as reading accelerometer and gyroscope data, we can design the driver to efficiently capture and transfer data to user applications. Furthermore, the project includes **configuring I2C communication**, which involves addressing potential challenges like bus arbitration, noise management, and ensuring reliable data transfer across the interface. Beyond the primary objective of establishing I2C communication and data access, this project will also explore techniques for error handling, synchronization, and data integrity within the driver. Implementing robust error-handling mechanisms is essential to ensure that data received from the sensor is accurate and consistent, particularly in applications where real-time or high-precision data is critical. Additionally, the driver will incorporate features to manage potential communication issues, such as handling bus errors or ensuring that data updates are synchronized across different layers of the system.

The project scope also extends to implementing a user-friendly method for accessing sensor data from the Linux user-space environment. By utilizing **sysfs or ioctl interfaces**, the driver will expose sensor data and configuration options to user-space applications, allowing developers to interact with the MPU6050 without requiring specialized knowledge of low-level hardware programming. This user-space access is key to broadening the driver's usability,

enabling easy integration with higher-level applications, and supporting rapid prototyping in sensor-based projects. Moreover, the project includes thorough testing and validation on the BeagleBone platform, where the driver's performance, reliability, and compatibility with the MPU6050 will be assessed under different operating conditions. This process will involve measuring data transmission rates, observing driver response times, and ensuring consistent behavior across multiple I2C read and write operations. Validating the driver in this way ensures that it performs optimally and reliably in real-world scenarios, contributing to a stable and efficient embedded Linux environment.

This project ultimately lays the groundwork for future enhancements and scalability. The modular design of the driver makes it easy to expand to support additional functionalities of the MPU6050, such as interrupt handling, low-power modes, and data filtering. Additionally, since the driver is generic, it can be adapted for other I2C sensors or devices with minimal changes, allowing developers to use it as a basis for more complex embedded projects involving multi-sensor systems. By creating a generic, robust, and reusable driver, this project significantly extends BeagleBone's capability as a versatile platform for various sensor applications in fields such as robotics, industrial automation, and the Internet of Things (IoT).

1.6 Approach

The project approach is structured around understanding both the hardware and software requirements to facilitate communication between the BeagleBone board and the MPU6050 sensor. The initial step involves a thorough exploration of the Embedded Linux and its setup for BeagleBone Black. I2C protocol and how it can be implemented within the Linux kernel. From here, the development of the device driver will focus on configuring the MPU6050 sensor through specific initialization routines, followed by establishing the reading and processing functions to interpret sensor data.

Setting up the Host and Target system is the first step in the approach. The Host system involves downloading essential packages for the board, Boot Images, RFS, and **Cross-compiler and Toolchain** – arm-linux-gnueabihf for the x64 platform. We must set up and understand a few basic terminologies for the Target system. We initiate our target setup by setting up the serial debug provided at the expansion header of the board (UART0). We also need to understand the different components and boot sequences of the board. At last, we need to **prepare the SD Card** by copying the **boot images and RFS** into it and booting the board from the SD Image. After the board is booted, we need an internet connection for updating the modules and downloading the required files whenever necessary. As the board doesn't have WiFi hardware attached to it, we need to enable the **internet over USB** or use an ethernet connection for connecting the board to the internet.

sIn the second step, we studied the working and implementation of the **Linux Device Driver and Linux Kernel Modules** and implemented a pseudo-character device driver using the virtual file system. To understand the workings of the LKM, we first study the concept of user space and kernel space in OS. Secondly, we understood the writing fundamental syntax for writing an LKM and performed the registration of the device. For Implementation, we have built an LKM for a pseudo-character driver and performed various operations such as – read, write, seek, open and close. For compilation and testing of the driver, we need to develop a **Make File**.

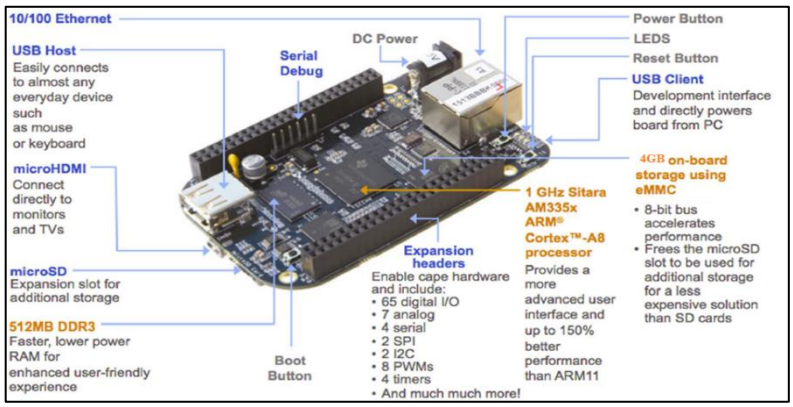

Figure 1.3 Various components on the board

For developing a device driver for MPU6050, we need to explore the I2C peripheral driver which already exists in the BeagleBone Black Linux kernel as MPU6050 works on I2C protocol. So, understanding the driver structure and its APIs is crucial for writing any device driver. The key approach here involves identifying how the **I2C adapter driver** interacts with the hardware controller and how the **I2C client driver** communicates with specific I2C devices. We first focus on the **I2C core**, which provides the framework and APIs that bridge adapters and clients. We study key data structures like i2c_adapter and i2c_client to grasp their roles in representing the bus and devices. Next, we look into the driver registration processes, such as i2c_add_adapter () and i2c_add_driver(), to see how drivers integrate into the kernel. Finally, we examine device tree bindings or platform data, which define device connections and configurations, enabling proper initialization. By following this structured, modular approach, we can systematically understand the interaction between the core, adapter, and client components. Finally, we would understand the workings of the IMU sensor and its internal register and develop a device driver for MPU6050 using the I2C device driver

1.7 Gantt Chart

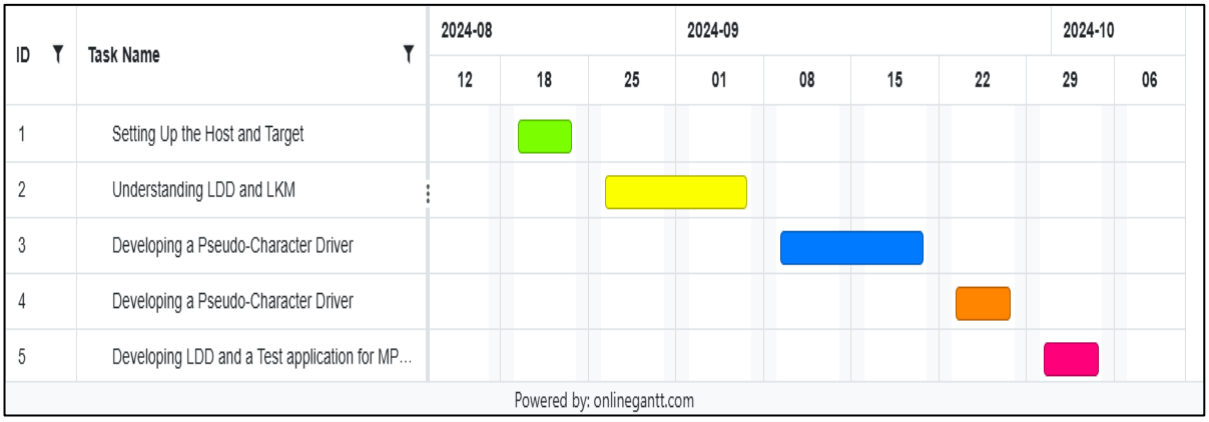

Figure 1.4 Working Timeline for Project

2 Literature Review

2.1 Description of Existing Technology and Current Trends

Device drivers are critical software components that facilitate communication between the operating system and hardware devices. They act as an abstraction layer, ensuring the operating system interacts with hardware without needing to know the hardware's internal workings. Linux, an open-source operating system, offers a robust framework for device driver development, particularly through its modular and layered architecture. The Linux kernel supports a wide array of devices, ranging from generic peripherals to specialized hardware. Among the hardware platforms, BeagleBone-based boards, which are versatile single-board computers, have gained prominence in embedded systems development due to their affordability, community support, and rich feature set.

BeagleBone boards, powered by the ARM architecture, are widely used for prototyping and deploying embedded systems. These boards support Linux-based operating systems, making them ideal for applications that require real-time control, automation, or custom hardware integration. Device driver development for BeagleBone boards typically involves interaction with the Linux kernel through subsystems such as character drivers, block drivers, or network drivers. Each subsystem has unique challenges, requiring developers to deeply understand kernel APIs, hardware specifications, and debugging tools. However, the existing approach often leads to hardware-specific drivers, limiting the reusability and maintainability of the code across different hardware platforms.

Current technologies for device driver development, while robust, face several limitations. One significant drawback is the lack of generic frameworks that enable seamless driver integration across various hardware platforms. Most device drivers are developed with specific hardware in mind, leading to fragmentation in driver ecosystems. This hardware specificity necessitates frequent rewriting or extensive modifications when migrating to newer hardware. Additionally, debugging drivers remains a challenging task, with tools like Minicom and Termius primarily catering to low-level debugging rather than providing intuitive, high-level abstractions. Furthermore, the onboarding process for new developers in this field remains steep due to the lack of comprehensive documentation and examples for modern driver development.

To address these challenges, advancements can focus on creating more generic and modular driver frameworks. Such frameworks would abstract hardware-specific details and provide unified interfaces for common functionalities, significantly reducing development overhead. Incorporating machine learning techniques for driver debugging and performance analysis is another promising direction. By analyzing patterns in kernel logs and system behavior, ML models can predict potential issues or suggest optimizations. Additionally, enhancing development environments with better debugging and testing tools integrated into IDEs, such as Eclipse, would streamline the workflow for developers.

2.2 Tools and Technologies Used

We utilized several tools to streamline the development and debugging process. For preparing SD cards for booting the target board, we relied on the **GParted** application to partition and format storage devices efficiently. For debugging, we used **Minicom** to establish serial communication and monitor logs from the target system, along with **Termius**, which provided

a robust SSH-based debugging interface. To write and manage code, we adopted **Visual Studio Code (VS Code)** as our primary text editor, leveraging its extensions for C/C++ development. Additionally, I2C tools were employed for initial I2C communication testing in user space, and **dmesg** was invaluable for analyzing kernel logs during driver development.

Our exploration involved a range of Embedded Linux technologies. We worked with the Linux kernel source code, focusing on I2C subsystems and device driver frameworks. Using device tree configurations, we described hardware details like the MPU6050's I2C connection to the system. Cross-compilation was essential for building the kernel and driver for the target board architecture. We also explored **Buildroot** for setting up a **custom Linux image** and used I2C protocol analysis to ensure proper communication with the MPU6050 sensor. These technologies collectively deepened our understanding of the Embedded Linux environment and kernel driver development.

3 Software Design

3.1 Hosts and Target Setup

3.1.1 Host Setup: Desktop/PC

For the development of a Linux device driver for BeagleBone-based boards, we chose to use a host platform running the latest version of Ubuntu, preferably the 64-bit version. While the 32-bit version of Ubuntu is also acceptable, the 64-bit version offers better compatibility and performance for our development tasks.

To set up our host machine, the first step is to ensure that the system is up to date. We begin by opening the terminal on our Ubuntu system and running the following command to update the package repository. Once the system is updated, we need to install several essential software packages to ensure smooth development and avoid potential compilation or runtime issues. These packages include compilers, libraries, and tools required for cross-compiling and interacting with the BeagleBone board.

```
~$ sudo apt-get update
~$ sudo apt-get install lzop u-boot-tools net-tools bison flex libssl-dev libncurses5-dev
libncursesw5-dev unzip chrpath xz-utils minicom wget git-core
```

By installing these essential packages, we prepare our host machine to handle the cross-compilation and debugging tasks required for developing a generic Linux device driver for BeagleBone-based boards. After setting up the host machine, the next step is preparing the SD card for booting. We need to download the appropriate boot images and root file system for the BeagleBone Black, which can be obtained from official sources. These images will be written onto the SD card using tools like dd or Etcher.

Since the BeagleBone Black uses the AM335x SoC with an ARM Cortex-A8 processor (ARMv7 architecture), we cannot compile directly on the board. Therefore, we need a cross-compiler toolchain. We will download and set up an ARM cross-compiler (such as Linaro) to compile the Linux source code, kernel modules, and applications on our host machine, which will then run on the BeagleBone Black.

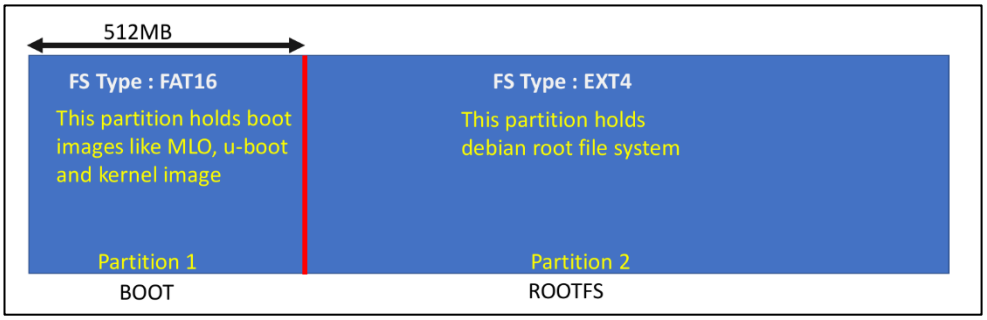

Figure 3.1 MicroSD card partitions

3.1.2 Target setup: BeagleBone Black

The **BeagleBone Black** (BBB) is a powerful, low-cost development board equipped with an **AM335x** SoC, featuring an **ARM Cortex-A8** processor. It is commonly used for embedded system development and supports various boot options. One key feature of the BeagleBone Black is its embedded MMC (eMMC), which provides additional storage for the board, allowing us to load the operating system and other files. The eMMC can be used in conjunction with an SD card for booting the board, offering flexibility in system setup.

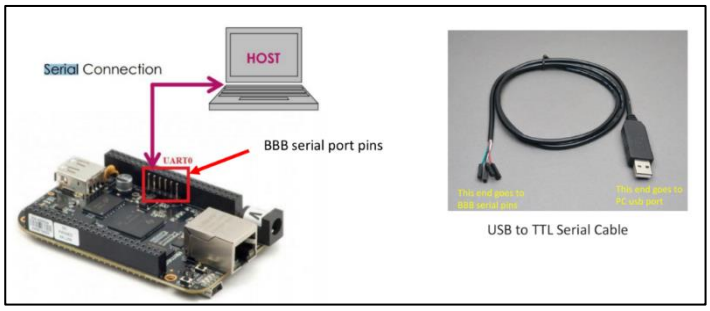

Figure 3.2 Serial debug port connection

For debugging, the BeagleBone Black provides a serial debug port. When Linux boots on the board, it emits early boot messages via the serial pins. These pins are exposed on the board, and by connecting a serial cable to the appropriate ports, we can capture these debug messages on our computer. The board offers multiple boot options, including booting from eMMC, SD card, USB, Ethernet, or SPI interfaces. This flexibility allows us to choose the most suitable boot method for our specific needs. This serial connection is essential for monitoring the boot process, troubleshooting issues, and ensuring that the system is booting correctly before moving on to further development.

3.2 Booting BBB via SD card

The BeagleBone Black supports multiple boot options, including eMMC, SD card, USB, and others. By default, it boots from the onboard eMMC where Debian OS is pre-installed. If we want to boot from an SD card, we need to press the boot button (S2) during power-up. This changes the boot sequence to check the MMC0 (MicroSD card) interface first.

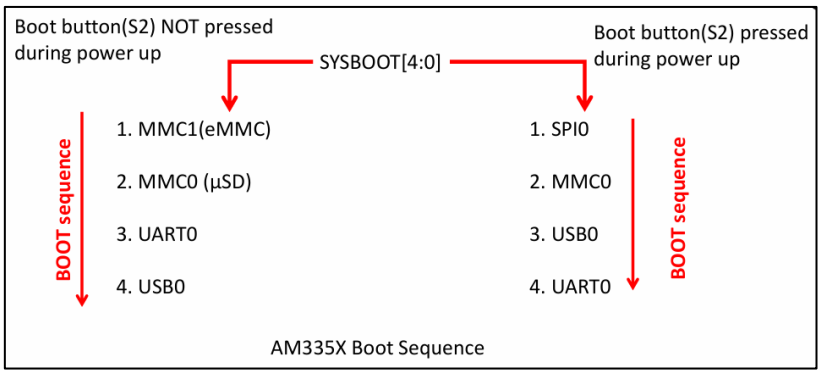

Figure 3.3 BeagleBone Black Booting Sequence

If the boot button is pressed, the board will attempt to boot from the SD card, provided valid boot images and partitions are present. We can avoid pressing the boot button every time by configuring the board to boot from the SD card permanently, which will be covered later.

When the boot button is pressed, the first boot source becomes SPI (which is usually invalid in our case). The board will then attempt to boot from the MMC0 interface, which corresponds to the MicroSD slot. To make this work, we must ensure that valid boot images and partitions are present on the SD card. Once the boot button is pressed during power-up, the board will attempt to boot from the SD card instead of the eMMC, following the new boot sequence. After setting this up, we won't need to press the boot button every time. There's a way to make this change permanent, which will be covered in a future article.

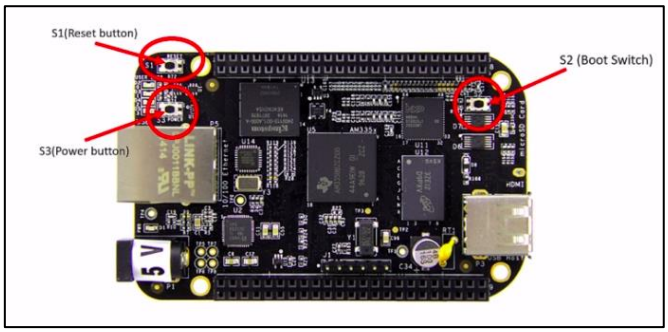

Figure 3.4 Boot Power and Reset button

The boot button (S2) directly affects a register called SYSBOOT, which controls the boot sequence. The state of the SYSBOOT register determines whether the board boots from the eMMC, SD card, or other sources. Additionally, the board has other important buttons:

3.3 Linux Device Driver and Kernel Modules

A device driver is a specialized software program that acts as a bridge between the operating system and hardware devices. It provides a standardized interface, enabling the OS and applications to communicate with hardware without needing to know the hardware's specifics.

This abstraction ensures that user applications can perform hardware-related operations through simple system calls while the driver handles the complex, low-level details.

Types of Device Drivers:

- **Character Drivers**: Manage devices that can be accessed character by character, such as serial ports and keyboards.
- **Block Drivers**: Handle devices that store and retrieve data in fixed-size blocks, such as hard drives and SD cards.
- **Network Drivers:** Enable communication over a network by handling packets for devices like Ethernet cards or Wi-Fi adapters.

Linux supports two types of kernel modules: **static modules** and **dynamic modules**. Static modules are integrated into the kernel during compilation, while dynamic modules, or Loadable Kernel Modules (LKMs), can be inserted or removed from the running kernel as needed. LKMs allow flexibility and modularity in managing hardware.

Writing a kernel module involves using specific syntax and APIs provided by Linux. Key steps include creating the module's initialization and cleanup functions (module_init() and module_exit()) and registering the device with the kernel using functions like register_chrdev() for character devices. To build a kernel module, we write a **Makefile**, which automates the compilation process by specifying the module name and source files, along with flags for the kernel build system.

After writing the module code, we compile it using the make command. Testing involves loading the module with insmod, verifying functionality, and removing it with rmmod. Tools like dmesg and /proc interfaces help debug and confirm the module's behavior. This structured process ensures compatibility and stability when adding new hardware support to Linux systems.

- **obj-m**: This is a special variable used for building loadable kernel modules (LKMs). It lists the target module object file(s) (my_module.o) that need to be compiled.
- **+=:** Allows you to append multiple modules if needed, e.g., obj-m += module1.o module2.o.
- **all**: This is a target that defines what happens when make is run without arguments.
- make -C: Specifies the directory containing the kernel source tree (/lib/modules/$(shell uname -r)/build).
- **$(shell uname -r):** Dynamically fetches the currently running kernel version.
- **M=$(PWD):** Points to the current directory (where the module source code is located).
- **modules**: Instructs the kernel build system to compile the module(s) defined in obj-m.
- **clean:** This target is used to clean up files generated during the build process.

The command make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules is a key aspect of integrating with the Linux kernel's build system. It ensures that the module being compiled is compatible with the currently running kernel version. The -C flag directs the build process to the kernel source directory located at /lib/modules/$(shell uname -r)/build, which contains the kernel's build system. By using $(shell uname -r), the Makefile dynamically identifies the kernel version, ensuring the module is built for the exact kernel currently in use.

This approach also supports modularity by enabling the definition of multiple modules in the Makefile. This is achieved by appending additional entries to the obj-m variable, where each entry corresponds to a module that needs to be built. For instance, obj-m += module1.o module2.o allows for the simultaneous compilation of multiple modules within the same process. Additionally, the Makefile includes a clean target, which is crucial for maintaining a fresh build environment. The clean command removes all intermediate and temporary files generated during the build process. This ensures that the module can be rebuilt from scratch without leftover artifacts, providing reliability and consistency in the development and testing phases.

3.4 Exploration of I2C Peripheral Driver in Linux

The I2C (Inter-Integrated Circuit) protocol is widely used for communication between peripherals and microcontrollers due to its simplicity and efficiency. In Linux, the I2C subsystem provides an abstraction layer to interact with I2C hardware and devices. The Linux kernel includes a standard I2C driver framework that simplifies the development of I2C drivers by exposing a set of APIs and structures. This allows developers to easily manage I2C communication, register devices, and handle data transfer.

Key Structures:

- **struct i2c_adapter**: Represents the I2C bus controller. It includes information about the hardware and function pointers for bus operations.
- **struct i2c_client**: Represents an I2C device connected to the bus. It holds device-specific information such as the I2C address.
- **struct i2c_driver:** Represents the I2C driver, defining callbacks for probing, removing, and managing devices.

Key APIs:

- **i2c_add_adapter():** Registers an I2C adapter with the kernel.
- **i2c_add_driver():** Registers an I2C driver and binds it to compatible devices.
- **i2c_transfer():** Performs read or write operations on the I2C bus.
- **i2c_master_send() and i2c_master_recv():** Simplified APIs for sending and receiving data over the I2C bus.
- **i2c_unregister_adapter() and i2c_del_driver()**: Remove an adapter or driver from the kernel.

Workflow:

- **Bus Registration:** Use i2c_add_adapter() to register the I2C bus controller.
- **Device Interaction:** Create an i2c_client to represent the device and use APIs like i2c_transfer() for communication.
- **Driver Development:** Implement an i2c_driver to manage devices, using callbacks like probe and remove.

3.5 Development of Device Driver for MPU6050

The MPU6050 is a popular MEMS sensor that integrates a 3-axis accelerometer and 3-axis gyroscope. It communicates over I2C or SPI interfaces, and to develop a device driver for it,

we need to understand the MPU6050's register set. The registers control various features like power management, sensor data, configuration, and more.

Key Registers of MPU6050:

- **Power Management Registers** (0x6B)**:** Control the power modes and reset the functionality of the sensor. Example: To wake up the sensor, set bit 6 of this register to 0.
- **Accelerometer and Gyroscope Registers** (0x3B to 0x43 for accelerometer data and 0x44 to 0x47 for gyroscope data)**:** These registers hold the sensor data values (accelerometer and gyroscope).
- **Configuration Registers** (0x1A): Used to configure the sampling rate and filter settings for the accelerometer and gyroscope.

Device Driver File for MPU6050: The device driver interacts with the MPU6050 via the Linux I2C subsystem. It initializes the device, reads data from registers, and provides this data to the user space application.

- **Initialization:**
  - Probe function checks for the device and configures registers like PWR_MGMT_1.
  - Set sample rate and enable accelerometer/gyroscope data streams.
- **Register Interaction**:
  - Use APIs like i2c_smbus_read_byte_data() to read data from specific registers. Combine high and low bytes for 16-bit data values.
- **File Operations:**
  - Implement read and write functions to interface with user-space applications.
  - Use ioctl for specific configurations, if required.
- **Module Integration**:
  - Register the driver using i2c_add_driver().
  - Clean up during removal using i2c_del_driver().

4 Results

The MPU6050 kernel module successfully integrates with the Linux I2C subsystem, allowing seamless interaction with the sensor. After loading the driver using insmod, the device is detected and initialized correctly, as confirmed by log messages in dmesg. A user-space application reads data from the device file /dev/mpu6050 and processes it to display sensor readings.

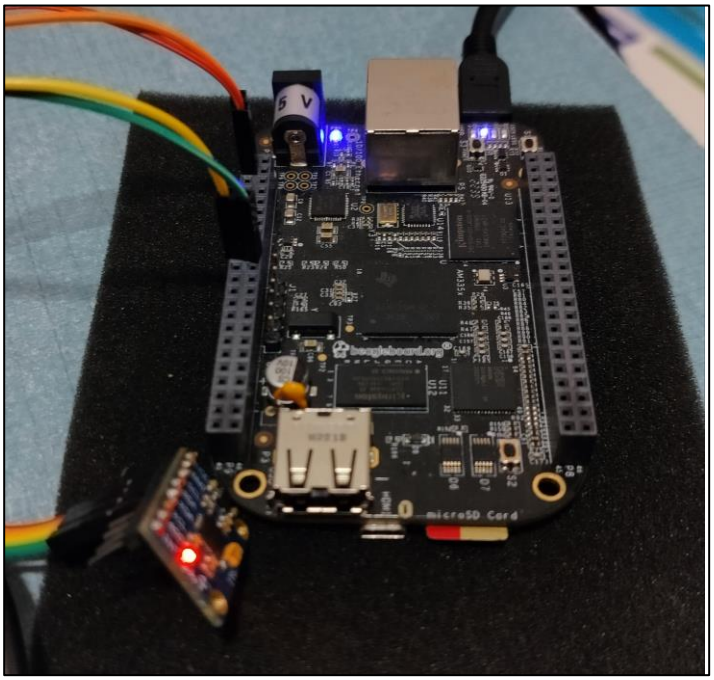

Figure 4.1 Interfacing MPU6050 with BeagleBone AI

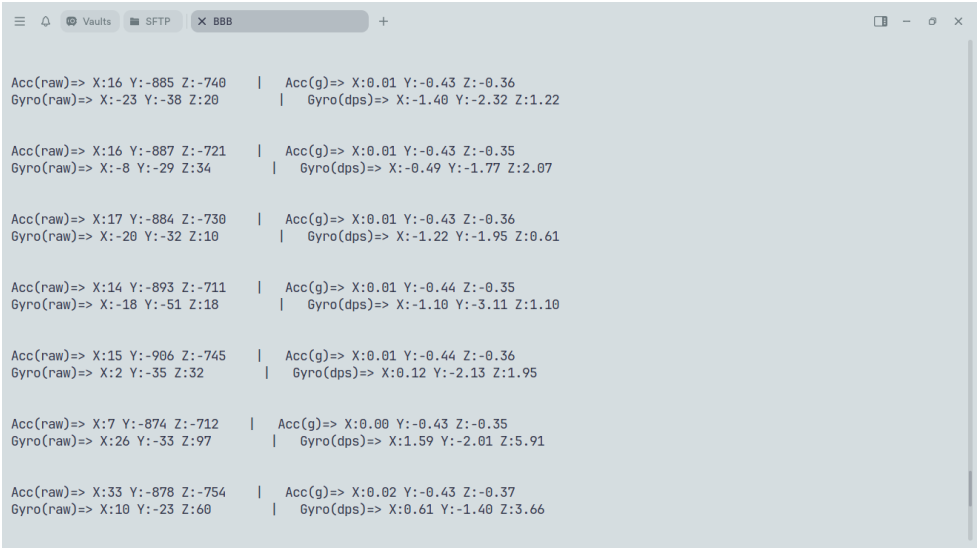

Figure 4.2 Output Readings of MPU6050

5 Conclusion and Future Scope

The development of device drivers for Embedded Linux, particularly for I2C-based peripherals like the MPU6050, is a crucial area in embedded systems. Through a structured understanding of the Linux kernel's subsystems, device tree configurations, and debugging tools, developers can bridge the gap between hardware and software. While tools like Buildroot, Yocto, and GParted simplify kernel customization and system setup, debugging solutions such as Minicom, Termius, and logic analyzers support iterative development. However, the process remains complex, requiring a deep understanding of hardware protocols, kernel-level programming, and cross-compilation techniques. Current advancements provide a strong foundation but fall short in areas like intuitive debugging, seamless integration, and pre-deployment testing for real-time applications.

The future of Embedded Linux and driver development lies in enhanced automation and user-friendly tools. Automated driver generation based on hardware descriptors or device tree files could drastically reduce development time and minimize errors. AI-driven solutions could provide real-time insights into driver performance and detect anomalies in sensor communication. The integration of virtualized environments with complete peripheral simulation, including virtual I2C devices, would allow developers to prototype and debug drivers without physical hardware. Additionally, improvements in real-time debugging tools that offer visual representations of the kernel and I2C interactions could make the development process more accessible. These advancements have the potential to revolutionize how device drivers are developed, tested, and deployed, accelerating innovation in embedded systems and IoT applications.

6 References

[1] ‘Linux Device Driver Programming Tutorial | FastBit EBA --- fastbitlab.com’. [Online]. Available: https://fastbitlab.com/tag/linux-device-driver-programming/.

[2] ‘I2C Bus Drivers — The Linux Kernel documentation --- docs.kernel.org’. [Online]. Available: https://docs.kernel.org/6.8/i2c/busses/index.html.

[3] Admin, ‘I2C Bus Driver Dummy Linux Device Driver – Linux Device Driver Tutorial - Part 38 --- embetronicx.com’. [Online]. Available: https://embetronicx.com/tutorials/linux/device-drivers/i2c-bus-driver-dummy-linux-device-driver-using-raspberry-pi/.

[4] ‘Linux Device Drivers 101 --- engineers.inpyjama.com’. [Online]. Available: https://engineers.inpyjama.com/learn/ldd-101.

Appendix A

I. Basic Skeleton of a Loadable Kernel Module (LKM)

```
#include <linux/init.h>    // Required for init and exit macros
#include <linux/module.h>  // Required for all kernel modules

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A Simple Linux Kernel Module");

// Initialization function
static int __init my_module_init(void) {
    printk(KERN_INFO "My Module: Initialization successful\n");
    return 0;
}

// Cleanup function
static void __exit my_module_exit(void) {
    printk(KERN_INFO "My Module: Cleanup successful\n");
}

module_init(my_module_init);
module_exit(my_module_exit);
```

II. Makefile for Kernel Module

```
obj-m += my_module.o  # Replace 'my_module' with your source file name (without .c)

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

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

III. Commands for Compilation and Testing

```
# Compile the module
make
# Insert the module
sudo insmod my_module.ko
# Check kernel logs for messages
dmesg | tail
# Remove the module
sudo rmmod my_module
# Clean the build
make clean
```

Appendix B

I. Device Registration Example (Character Device)

```
#include <linux/module.h>
#include <linux/fs.h>

#define DEVICE_NAME "my_char_device"
static int major_number;

// File operation functions
static int dev_open(struct inode *inodep, struct file *filep) {
    printk(KERN_INFO "Device opened\n");
    return 0;
}

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

static struct file_operations fops = {
    .open = dev_open,
    .release = dev_release,
};

static int __init my_device_init(void) {
    major_number = register_chrdev(0, DEVICE_NAME, &fops);
    if (major_number < 0) {
        printk(KERN_ALERT "Device registration failed\n");
        return major_number;
    }
    printk(KERN_INFO "Device registered with major number %d\n", major_number);
    return 0;
}

static void __exit my_device_exit(void) {
    unregister_chrdev(major_number, DEVICE_NAME);
    printk(KERN_INFO "Device unregistered\n");
}

module_init(my_device_init);
module_exit(my_device_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A Simple Character Device Driver");
```

II. Device Driver for MPU6050

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/i2c.h>
#include <linux/device.h>
#include <linux/fs.h>
#include<linux/cdev.h>
#include<linux/kdev_t.h>
#include<linux/uaccess.h>

#define DEV_MEM_SIZE 512

#define MPU6050_WHO_AM_I        0x75
#define MPU6050_PWR_MGMT_1      0x6B
#define MPU6050_ACCEL_XOUT_H    0x3B
#define MPU6050_ACCEL_YOUT_H    0x3D
#define MPU6050_ACCEL_ZOUT_H    0x3F
#define MPU6050_GYRO_XOUT_H     0x43
#define MPU6050_GYRO_YOUT_H     0x45
#define MPU6050_GYRO_ZOUT_H     0x47

#define DEVICE_NAME "mpu6050"
#define CLASS_NAME "imu"

struct mpu6050_data {
   struct i2c_client *client;
   int16_t accel_x;
   int16_t accel_y;
   int16_t accel_z;
   int16_t gyro_x;
   int16_t gyro_y;
   int16_t gyro_z;
};


static ssize_t mpu6050_read(struct file *filp, char __user *buff, size_t count, loff_t *f_pos);
//static ssize_t mpu6050_write(struct file *filp, const char __user *buffer, size_t count,
loff_t *f_pos);
static int mpu6050_open(struct inode *inode, struct file *filp);
static int mpu6050_write_byte(struct i2c_client *client, u8 reg, u8 value);

#undef pr_fmt
#define pr_fmt(fmt) "%s: " fmt, __func__

static dev_t device_number;

static int mpu6050_probe(struct i2c_client *client);
static void mpu6050_remove(struct i2c_client *client);
```

```
static const struct i2c_device_id mpu6050_id[] = {
    { "mpu6050", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, mpu6050_id);

static const struct of_device_id mpu6050_of_match[] = {
    { .compatible = "invensense,mpu6050" },
    { }
};
MODULE_DEVICE_TABLE(of, mpu6050_of_match);

static struct i2c_driver mpu6050_driver = {
    .driver = {
        .name = "mpu6050",
        .of_match_table = mpu6050_of_match,
    },
    .probe = mpu6050_probe,
    .remove = mpu6050_remove,
    .id_table = mpu6050_id,
};


struct class *class_mpu;
struct device *device_mpu;
struct cdev cdev_mpu;
static struct mpu6050_data *mpu_data;

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .read = mpu6050_read,
    .open = mpu6050_open,

};

/* Read sensor data */
static int mpu6050_read_word(struct i2c_client *client, u8 reg)
{
    u8 buf[2];
    int ret;

    ret = i2c_master_send(client, &reg, 1);
    if (ret < 0)
        return ret;

    ret = i2c_master_recv(client, buf, 2);
    if (ret < 0)
        return ret;

    return (buf[0] << 8) | buf[1];
}
```

```
static int mpu6050_read_sensor_data(struct mpu6050_data *mpu)
{
    struct i2c_client *client = mpu->client;

    mpu->accel_x = mpu6050_read_word(client, MPU6050_ACCEL_XOUT_H);
    mpu->accel_y = mpu6050_read_word(client, MPU6050_ACCEL_YOUT_H);
    mpu->accel_z = mpu6050_read_word(client, MPU6050_ACCEL_ZOUT_H);
    mpu->gyro_x = mpu6050_read_word(client, MPU6050_GYRO_XOUT_H);
    mpu->gyro_y = mpu6050_read_word(client, MPU6050_GYRO_YOUT_H);
    mpu->gyro_z = mpu6050_read_word(client, MPU6050_GYRO_ZOUT_H);

    return 0;
}

/* File operations */
static ssize_t mpu6050_read(struct file *file, char __user *buffer, size_t len, loff_t *offset)
{
    int ret;
    struct mpu6050_data data;

    if (!mpu_data)
        return -EFAULT;

    ret = mpu6050_read_sensor_data(mpu_data);
    if (ret < 0)
        return ret;

    data = *mpu_data;

    if (copy_to_user(buffer, &data, sizeof(data)))
        return -EFAULT;

    return sizeof(data);
}

static int mpu6050_open(struct inode *inode, struct file *file)
{
    return 0;
}


static int mpu6050_probe(struct i2c_client *client)
{
    int ret;

    mpu_data = devm_kzalloc(&client->dev, sizeof(struct mpu6050_data), GFP_KERNEL);
    if (!mpu_data)
        return -ENOMEM;
mpu_data->client = client;
```

```
    ret = mpu6050_read_word(client, MPU6050_WHO_AM_I);
    if (ret < 0 || (ret & 0xFF) != 0x68) {
        dev_err(&client->dev, "WHO_AM_I check failed: 0x%x\n", ret);
        return -ENODEV;
    }

    ret = mpu6050_write_byte(client, MPU6050_PWR_MGMT_1, 0x00);
    if (ret < 0)
        return ret;

    dev_info(&client->dev, "MPU6050 driver initialized successfully\n");
    return 0;

}

static int mpu6050_write_byte(struct i2c_client *client, u8 reg, u8 value) {
    struct i2c_msg msg;
    u8 data[2] = {reg, value};

    msg.addr = client->addr;
    msg.flags = 0;
    msg.len = sizeof(data);
    msg.buf = data;

    return i2c_transfer(client->adapter, &msg, 1);
}


static void mpu6050_remove(struct i2c_client *client)
{
    device_destroy(class_mpu, device_number);
    class_destroy(class_mpu);
    cdev_del(&cdev_mpu);
    unregister_chrdev_region(device_number, 1);
    dev_info(&client->dev, "MPU6050 driver removed\n");
    pr_info("exit with clean up successfully\n");
}

static int __init mpu6050_init(void)
{
    int ret;

    printk(KERN_INFO "MPU6050 I2C driver initializing...\n");

    ret = i2c_add_driver(&mpu6050_driver);
    if (ret) {
        printk(KERN_ERR "Failed to add I2C driver\n");
        return ret;
    }
```

```
        ret = alloc_chrdev_region(&device_number, 0, 2, "MPU6050_DEVICE");
            if (ret < 0) {
                    pr_err("Error While register device number!!\n");
                    goto out;
            }
    pr_info("Device number - <major>:<minor> = %d:%d", MAJOR(device_number),
MINOR(device_number));


    cdev_init(&cdev_mpu, &fops);
    cdev_mpu.owner = THIS_MODULE;


    ret = cdev_add(&cdev_mpu, device_number, 1);
            if (ret < 0) {
                    pr_err("error while cdev_add!!\n");
                    goto unreg_chrdev;
            }


    class_mpu = class_create("mpu6050_class");

    if (IS_ERR(class_mpu)) {
        ret = PTR_ERR(class_mpu);
        goto cdev_del;
    }

    device_mpu = device_create(class_mpu, NULL, device_number, NULL, "mpu6050");
    if (IS_ERR(device_mpu)) {
        ret = PTR_ERR(device_mpu);
        goto class_destroy;
    }


class_destroy:
    class_destroy(class_mpu);
cdev_del:
    cdev_del(&cdev_mpu);
unreg_chrdev:
    unregister_chrdev_region(device_number, 1);
    return ret;
out:
            pr_info("Module insertion failed!!\n");
            return ret;

    printk(KERN_INFO "MPU6050 driver initialized\n");
    return 0;
}
static void __exit mpu6050_exit(void)
{
```

```
    printk(KERN_INFO "MPU6050 I2C driver exiting...\n");

    i2c_del_driver(&mpu6050_driver);

    device_destroy(class_mpu, device_number);
    class_destroy(class_mpu);
    cdev_del(&cdev_mpu);
    unregister_chrdev_region(device_number, 1);
    pr_info("exit with clean up successfully\n");


}

// Register the initialization and exit functions
module_init(mpu6050_init);
module_exit(mpu6050_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("MPU6050 I2C Driver");
```

III. Application Code for MPU6050

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

#define DEVICE "/dev/mpu6050"

int main() {
    int fd = open(DEVICE, O_RDONLY);
    if (fd < 0) {
        perror("Failed to open the device");
        return -1;
    }

    int16_t data[6]; // Accelerometer and Gyroscope data
    if (read(fd, data, sizeof(data)) < 0) {
        perror("Failed to read data");
        close(fd);
        return -1;
    }

    printf("Accel X: %d, Y: %d, Z: %d\n", data[0], data[1], data[2]);
    printf("Gyro X: %d, Y: %d, Z: %d\n", data[3], data[4], data[5]);

    close(fd);
    return 0;
}
```