iJRASET

International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 13 Issue: III Month of publication: March 2025

DOI: https://doi.org/10.22214/ijraset.2025.67488

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue III Mar 2025- Available at www.ijraset.com

Implementation of I2C Device Driver in Linux

Shrinidhi V[1], Geetishree Mishra[2]
Dept of Electronics and Communication, BMS College of Engineering, Bangalore, Karnataka, India

Abstract: This project focuses on the implementation of an I2C device driver in Linux for enhancing the communication between interconnected hardware devices. The project's objective is to gain a deep understanding of the Linux kernel, different types of device drivers and the working of the I2C protocol. The methodology involves understanding the I2C protocol by exploring the I2C architecture in Linux, setting up the development environment, and creating an OLED device driver file. By testing and verifying the created device driver's functionality using a Raspberry Pi 3B+ model and an SSD1306 OLED display, the project demonstrates the successful implementation of the I2C device driver for SSD1306 OLED. The ultimate aim is to enhance system functionality by improving communication efficiency between hardware devices through the optimized I2C protocol.
Keywords: I2C – inter integrated circuit, OLED – organic light emitting diode

I. INTRODUCTION

A. Linux Kernels

The Linux kernel serves as the heart of the Linux operating system. It manages essential hardware resources, including the CPU, memory, and input/output devices. Additionally, the Linux kernel offers a collection of APIs that enable users to access and manage these hardware components effectively.

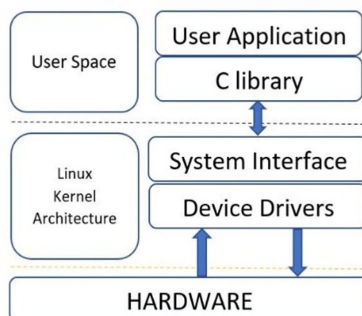

Figure 1: Representation of kernel space and User space

B. Device Drivers

A software application known as a device driver facilitates communication between a particular hardware device and the operating system of a computer. By giving the operating system the instructions it needs to manage and communicate with the hardware, it serves as a translator. The operating system can identify and make use of the features and capabilities of a variety of hardware components, including network adapters, printers, graphics cards, sound cards, and more, thanks to device drivers. [5]

C. Device Driver Types

There are three types of devices in the conventional classification:

- Character Device: Character Device: A hardware file that reads and writes data character by character is called a char file. Keyboards, mice, and serial printers are a few traditional examples. If a user writes data in a char file, no other user can write data in the same char file, preventing another user from accessing it. Character files are used for communication and cannot be mounted. They write data using a synchronized technique. [7]
- Block Device: A block file is a type of hardware file that reads and writes data in blocks rather than individual characters. When writing or reading data in bulk, this kind of file is quite helpful. All of the disks, including CD-ROM, USB, and HDD, are block devices. Data writing is a CPU-intensive process that is carried out asynchronously. Data on actual hardware is stored in these device files, which can be mounted to enable access to the data they contain. [7]

- Network Device: According to Linux's network subsystem, a network device is an object that transmits and receives data packets. Usually, this is a tangible item, like an Ethernet card. However, some network devices—like the loopback device, which is used to relay data to yourself—are software-only. [6]

D. I2C Protocol

A data line (SDA) and a clock line (SCL) are the two wires that various devices can use to interact with one another via the serial communication protocol known as I2C. It was created by Philips Semiconductors, which is now known as NXP Semiconductors, and is currently a commonly used standard for integrated circuit communication. [10]

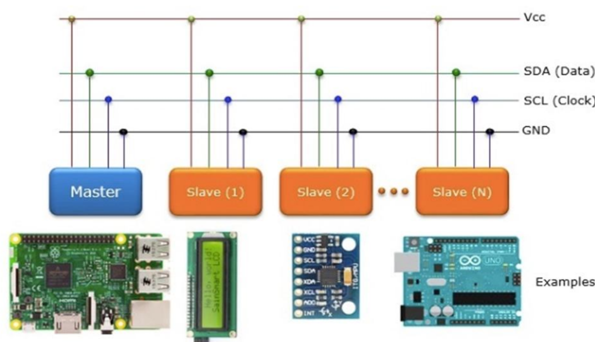

Figure 2: I2C Protocol

One bus master device, like a microcontroller or microprocessor, and one or more slave devices, including sensors, EEPROMs, displays, etc., are usually linked to the bus in an I2C system.

1) Start Condition: When the bus master pulls the SDA line low while keeping the SCL line high, creating a start condition, communication starts.
2) Addressing: The address of the slave device with which the bus master wishes to communicate is sent. On the bus, every slave device has a distinct address. Seven bits make up the address, and a direction bit that indicates whether the operation is read or write comes next.
3) Data Transfer: Data can be transferred to or from a specific slave device once the bus master has addressed it. Eight-bit blocks of data are sent and received, and the recipient's acknowledgment bit (ACK) comes after.

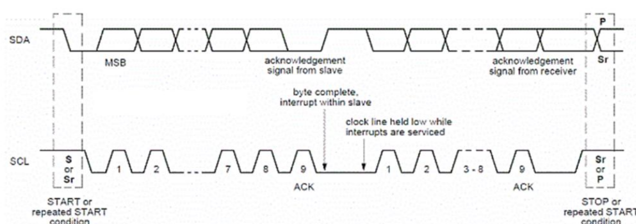

Figure 3: I2C data Transfer

4) Clock Synchronization: The data flow is synchronized using the clock line (SCL). To read and write data on the data line (SDA), the slave and master devices both adhere to the clock edges.
5) Stop Condition: The bus master pulls the SDA line high while keeping the SCL line high to provide a stop condition after the communication is finished. This signifies that the conversation is over.
6) It is significant to remember that the I2C protocol allows for multi-master configuration, which allows the bus to be accessed by various bus master devices. Apart from that, I2C may send data at different speeds; the most used modes are standard (100 kbps) and fast (400 kbps). [3]

E. Working of I2C in Linux

The Linux operating system's support and capability for communicating with I2C devices is referred to as I2C. Developers may interact with I2C devices from user-space applications thanks to Linux's I2C subsystem, a stable and adaptable foundation.

Working of I2C protocol in Linux Kernel:

1) Kernel Configuration: Linux requires the I2C subsystem and related I2C bus drivers to be compiled as modules or into the kernel in order to provide I2C support. The kernel setup stage is when this can be accomplished.
2) Kernel I2C Subsystem: A collection of APIs and tools for controlling I2C buses, devices, and communication are offered by the Linux kernel's I2C subsystem. The I2C bus protocol's hardware-specific features are implemented by bus drivers, while the core I2C subsystem manages the low-level functions.
3) I2C Bus Detection: The I2C bus drivers search for I2C buses on the Linux system as it boots up. Bus numbers like "i2c-0" and "i2c-1" are given to each discovered bus.
4) Device Drivers: Linux offers device drivers that connect to certain I2C devices in addition to bus drivers. These drivers specify the features and actions unique to each device. The appropriate device driver is loaded and linked to the compatible I2C device when it is found on the bus.
5) I2C Tools: Linux provides a number of command-line tools to facilitate communication with I2C devices. The utilities i2c-tools and i2c-dev are frequently utilized. Tools like i2cdetect, which can identify I2C devices on a bus, and i2cget/i2cset, which can read and write register values on I2C devices, are available from i2c-tools. Through the /dev filesystem, user-space applications can access I2C devices as regular files thanks to the i2c-dev driver.
6) User-Space Interaction: By gaining access to the relevant device file supplied by the i2c-dev driver, user-space applications can interact with I2C devices via the I2C protocol. This enables developers to use common file operations, such as open, read, write, etc., to read from and write to I2C devices.

Overall, the developers may more easily engage with sensors, EEPROMs, displays, and other I2C-based peripherals in their applications thanks to Linux's I2C subsystem, which offers a standardized and effective means of communicating with I2C devices.

F. Hardware And Software Requirements

1) Raspberry Pi

Using a conventional keyboard and mouse, the Raspberry Pi is a low-cost, credit card-sized computer that can be plugged into a TV or computer monitor. A wide range of operating systems and apps can be run on the Raspberry Pi, a general-purpose CPU. It is not made to manage particular systems or gadgets. It has all the features you would expect from a desktop computer, including word processing, spreadsheet creation, gaming, and high definition video and internet browsing.

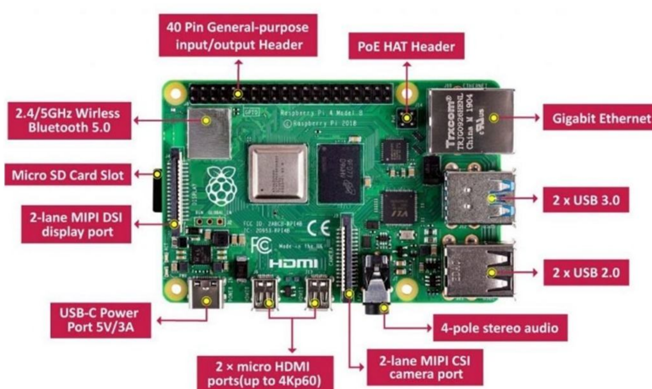

Figure 4: Specification of Raspberry pi

2) OLED I2C Display (SSD1306)

A controller and a single-chip CMOS OLED/PLED driver for organic or polymer light-emitting diode dot-matrix graphic display systems is the SSD1306. It has 64 commons and 128 segments. By integrating oscillator, display RAM, and contrast management, the SSD1306 lowers power consumption and the number of external components. The generic MCU transmits data and commands via the Serial Peripheral Interface, I2C interface, or hardware-selectable Parallel Interface compatible with the 6800/8000 series. It is appropriate for a wide range of small, portable devices, including calculators, MP3 players, and mobile phone sub-displays.

The SSD1306 display has four pins:
- VCC: Power supply (3.3V - 5V),
- GND: Ground,
- SCL: I2C clock signal,
- SDA: I2C data signal

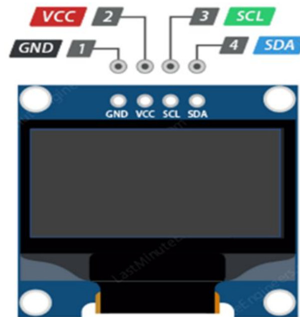

Figure 5: OLED (SSD 1306)

The OLED screen needs a supply voltage of 7V to 15V, whereas the SSD1306 controller operates between 1.65V and 3.3V. Internal charge pump circuitry is used to provide all of these various power needs. This eliminates the need for a logic level converter and enables it to be readily connected to any 5V logic microcontroller.

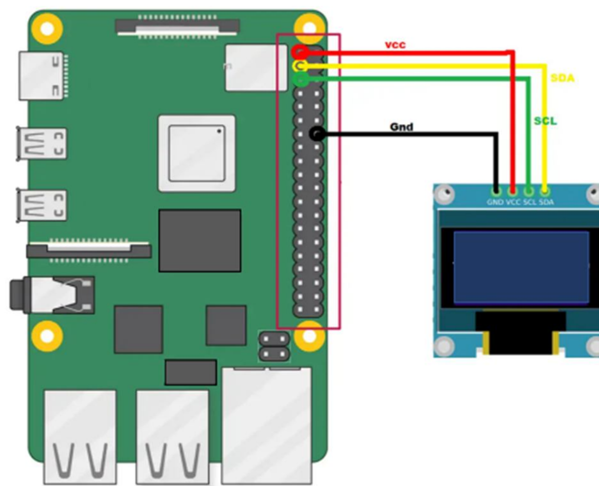

Figure 6: Interfacing OLED with Raspberry pi

3) Software Requirements

The Raspberry Pi 3B+ uses Raspberry Pi OS, which is a Debian-based operating system specifically designed for the Raspberry Pi. The latest kernel version for the Raspberry Pi 3B+ is 6.6.

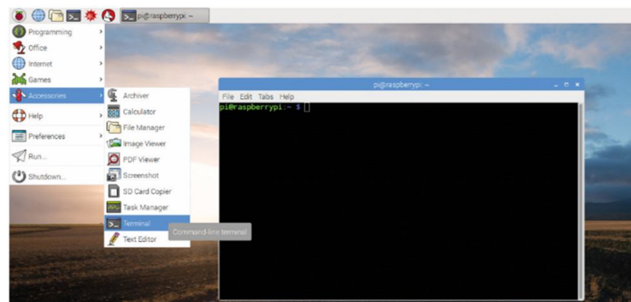

Figure 7: Linux Terminal in Raspberry pi

The following are the software requirements on the Raspberry Pi:

- Raspberry Pi OS with desktop
- C compiler (gcc)
- Make utility
- Kernel headers

II. LITRATURE REVIEW

The authors have explained tasks like first porting the OS to the Pi board, and how the device driver handled communication between external hardware and system hardware. The operation of the Bluetooth and Wi-Fi modules was then explained. Yocto project usage was also covered in detail. And concluded that the Raspberry Pi boards can be utilized in the future for applications like home automation, such as smart TVs, security, etc. [1]

This paper deals with design, implementation and testing of character device driver for the GPIO pins of ARM Cortex based platform. Device driver is a piece of code written in C language which is responsible for controlling the hardware and part of the kernel. ARM based platform Raspberry Pi is used in this work, supports the Linux OS. Due to missing of dependencies and patches, the option of cross compilation is available such as tool chain which contains GCC compiler, assembler, linker and debugger. [2]

This paper describes the serial communication protocols that are widely used in electronics in order to transfer data. The interface between the FPGA (Basys 2 Spartan 3E) and the DC motor has been achieved by using Dual Master I2C bus controller. To implement dual master configuration, first of all single master slave I2C bus controller is developed and then using arbitration technique, this design has been extended to two master devices. Thus, dual master I2C bus technique has been implemented by using arbitration technique. Finally, the synthesized design, obtained on FPGA, is interfaced with DC motor, which acts as a slave device for the masters. The direction of rotation of the DC motor has been controlled by the two masters. [3]

The article reports on the suggested algorithm of voltage supply management from user space programs. Much attention is given to details of potentiometer driver implementation. Reader’s attention is drawn to the driver initialization and device instantiating methods for ISL22317 and peculiarities of interaction between potentiometers and PCA9536 chip. Suggested software support makes possible to manage up to four potentiometers bound to common I2C bus. Article also informs on algorithm of management up to 4 power sources bound to the same I2C bus. Proposed algorithm is based on I2C-bus multiplexing by Linear LTC4306 chip. [4]

III. METHODOLOGY

Creating an I2C device driver involves several steps:

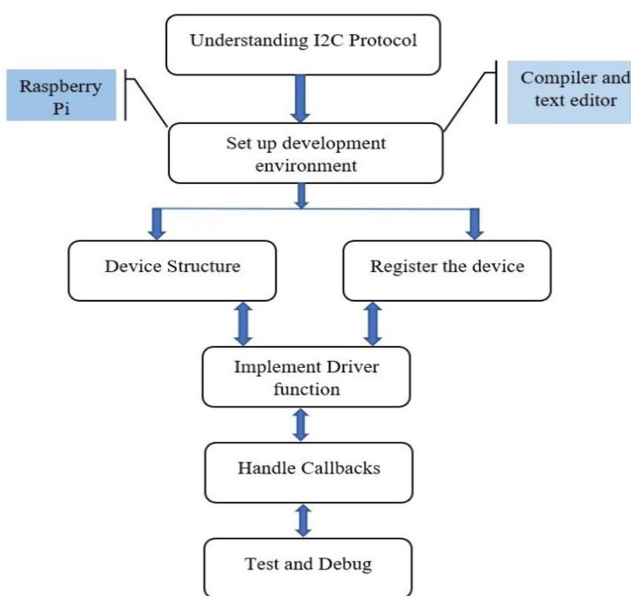

Figure 8: Flow graph of creating I2C Device Driver

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue III Mar 2025- Available at www.ijraset.com

1) Understand the I2C protocol: Learn about the fundamentals of the I2C protocol and how it works. Discover the many I2C communication signals, addressing modes, and data transmission techniques.
2) Set up development environment: Install the software and tools required for driver development, including a text editor, compiler, kernel headers, and a Linux distribution.
3) Define the device structure: Determine whatever hardware device you wish to interface with and specify the required data structures, including register maps, device IDs, and addresses. The device will be able to communicate with these buildings.
4) Register the device: Register the hardware device with the Linux kernel and set aside resources for the I2C bus. This step makes it possible for the kernel to identify and control the device.
5) Implement driver functions: Write the read, write, initialization, and control actions that the device driver requires. The required setups, data transport, and hardware device interactions will be managed by these services.
6) Handle callbacks: Use callback functions to control operations that are triggered by interruptions. When particular data or events are received from the hardware device, these callbacks will be activated.
7) Test and debug: After compilation, install the driver on the computer. Verify the driver's functionality using the proper testing techniques, such as unit and integration testing. Troubleshoot any problems that might come up while testing.
8) Documentation and finalization: Keep track of the driver's restrictions, usage, and any special hardware setups that are needed. Using the appropriate packaging and licensing instructions, get the driver ready for final deployment or distribution.

Note that the particular procedures and implementation details may change based on the hardware device characteristics, kernel settings, and target platform. Furthermore, consulting development materials and documentation tailored to the Linux kernel version in use might offer helpful direction when creating drivers.

A. *Device File Creation*

Linux device files, such as those for Raspberry Pi, provide as a transparent channel of communication between hardware and user-space applications. They look just like any other file that can be mmaped, read from, or written to. Interacting with a device file causes the kernel to identify the I/O request and forward it to the appropriate device driver, which then carries out the required tasks, including sending data to hardware or reading data from a serial port. You can use the mknod command to manually construct a device file. Here is the syntax:

- sudo mknod -m <permissions> <name> <device type> <major> <minor>
- <name> is the full path of the device file (e.g., /dev/my_device).
- <device type> specifies whether it's a character device (c) or block device (b).
- <major> and <minor> represent the major and minor device numbers, respectively.

The -m flag can be used to optionally set the device file's permissions. After it is created, the device file enables the relevant device or device driver to be used for a variety of purposes. It's crucial to remember that anyone can create device files manually, even before the driver is loaded.

File Operations: The operations that can be carried out on a character device file are specified by the File Operations structure, also referred to as the struct file_operations structure. This structure implements each operation as a function pointer.
Some commonly used file operations in the struct file_operations structure are:

- int (*open)(struct inode *inode, struct file *file): Called when a device file is opened.
- int (*release)(struct inode *inode, struct file *file): Called when a device file is closed.
- ssize_t (*read)(struct file *filp, char __user *buf, size_t len, loff_t *off): Called when data is read from the device file.
- ssize_t (*write)(struct file *filp, const char __user *buf, size_t len, loff_t *off): Called when data is written to the device file.

A character device driver is linked to the File Operations structure by assigning it to the struct cdev structure's ops field. The character driver can manage open, shut, read, and write activities on the device file thanks to these structures and operations.

B. *Communication Between User Space And Kernel Space*

In Linux everything is a File. For a device driver two separate application programs has to be developed:

- User Space application (User program)
- Kernel Space program (Driver)

The device file will be used by the user software to interface with the kernel space program.

Major and minor numbers, device files, and file operations are all part of the Kernel Space Program, commonly referred to as the Device Driver. Other file operations in the device driver include the following functions:

1) Open driver: This function is in charge of managing the device driver's opening. When a user asks to use the device, it gets activated.
2) Write driver: The user can transmit data or strings to the kernel device driver using the write driver function. The kernel space is then used to store this data.
3) Read driver: When the user want to retrieve data from the device driver, they use the read driver function. It sends the data to the user-space program after reading it from the kernel area.
4) Close driver: The close driver function is invoked to manage the closing procedure after the user has completed using the device driver.

Along with these features, this driver also makes use of the following ideas and capabilities:

- kmalloc(): This function is used to dynamically allocate memory in the kernel space.
- kfree(): It is used to deallocate the dynamically allocated memory in the kernel space.
- copy_from_user(): This function copies data from the user space to the kernel space while ensuring proper memory access and security.
- copy_to_user(): Conversely, this function copies data from the kernel space to the user space while maintaining appropriate memory permissions.

These features are essential for memory management and facilitating safe data transmission between the device driver's kernel and user areas.

C. IOCTL in Linux

Input and Output Control, or IOCTL, is a protocol used to communicate with device drivers. The majority of driver categories support this system call. This is primarily used when managing certain device actions for which the kernel lacks a system call by default.

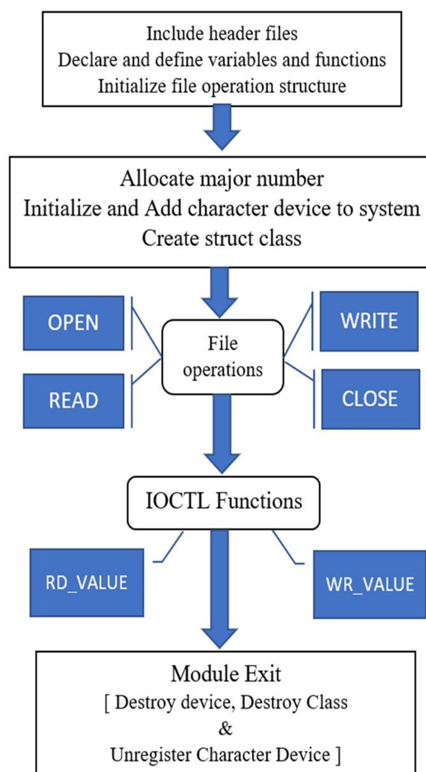

Figure 9: IOCTL Flow

D. I2C Linux Device Drivers

The I2C subsystem was separated by the kernel into Buses and Devices. Once more, the buses are separated into adapters and algorithms. Once more, the devices are separated into Drivers and Clients. You will gain some knowledge from the graphic below.

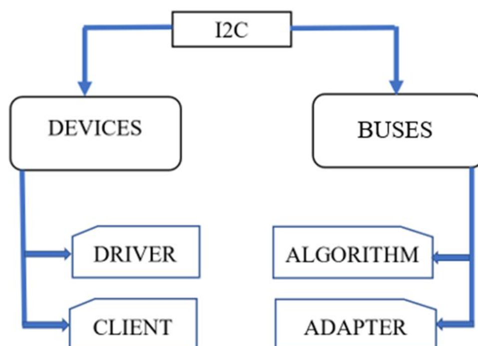

Figure 10: I2C subsystem

1) Algorithm: A universal code included in an algorithm driver can be applied to an entire class of I2C adapters.
2) Adapters: A bus is essentially represented by an adapter, which is used to link a specific I2C to a bus number and algorithm. Every individual adapter driver either has its own implementation or is dependent on a single algorithm driver.
3) Clients: On the I2C, a client is a chip (slave).
4) Drivers: This is the driver that is being written for the client.

Algorithm and Adapter are typically less integrated than Driver and Client. Therefore, an I2C bus requires a driver, and I2C devices require equivalent drivers (often one driver per device).

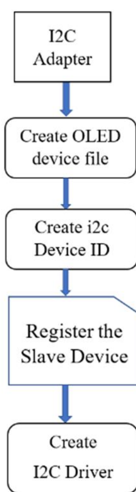

Figure 11: Creating I2C Driver files

Get the I2C adapter

The Raspberry Pi 3B+ already has the i2c-1 bus available. The command ls -al /sys/bus/i2c/devices/ can be used to confirm it. To obtain the adapter structure for this I2C bus, using the API listed below.

```
struct i2c_adapter *i2c_get_adapter(int nr);
```

Create Device File

Now board info structure is ready. Instantiate the device from that I2C bus.

```
struct i2c_client * i2c_new_device ( struct i2c_adapter * adap, struct i2c_board_info const * info);
```

Create the device id and register

```
#define I2C_BUS_AVAILABLE   (          5 )
#define SLAVE_DEVICE_NAME   ( "ETX_OLED" )
#define SSD1306_SLAVE_ADDR  (       0x3C )

static struct i2c_adapter *etx_i2c_adapter     = NULL;
static struct i2c_client  *etx_i2c_client_oled = NULL;
```

Figure 12: I2C Driver structure

Add the I2C driver to the I2C subsystem

i2c_add_driver(struct i2c_driver *i2c_drive);

E. Creating a Client Driver

Declaring parameters and initializing client structures are among the protocols that must be implemented by the Client driver file. Writing the structs required to read and write data from the master and the bus is another aspect of it. Additionally, it involves creating an I2C driver structure file that needs to be registered in Linux.

```
struct i2c_driver {
  unsigned int class;
  int (* attach_adapter) (struct i2c_adapter *);
  int (* probe) (struct i2c_client *, const struct i2c_device_id *);
  int (* remove) (struct i2c_client *);
  void (* shutdown) (struct i2c_client *);
  void (* alert) (struct i2c_client *, unsigned int data);
  int (* command) (struct i2c_client *client, unsigned int cmd, void *arg);
  struct device_driver driver;
  const struct i2c_device_id * id_table;
  int (* detect) (struct i2c_client *, struct i2c_board_info *);
  const unsigned short * address_list;
  struct list_head clients;
};
```

Figure 13: declaration of client driver

The acknowledgement is transmitted before to the client receiving the data, and this will start the start condition with the slave address as the R/W bit.

```
static int I2C_Read(unsigned char *out_buf, unsigned int len)
{
    /*
    ** Sending Start condition, Slave address with R/W bit,
    ** ACK/NACK and Stop condtions will be handled internally.
    */
    int ret = i2c_master_recv(etx_i2c_client_oled, out_buf, len);

    return ret;
}
```

Figure 14: Start condition in client driver

The function that receives the data from the output buffer is called I2C Read. It also sends an acknowledgement for the stop condition in accordance with protocol.

```
static int I2C_Write(unsigned char *buf, unsigned int len)
{
    /*
    ** Sending Start condition, Slave address with R/W bit,
    ** ACK/NACK and Stop condtions will be handled internally.
    */
    int ret = i2c_master_send(etx_i2c_client_oled, buf, len);

    return ret;
}
```

Figure 15: i2c read function in client

IV. RESULTS AND DISCUSSION

Driver is written in C and has been inserted into the kernel successfully using “sudo insmod driver.ko” and the “dmesg” command gives the information about the driver.

Figure 16: Inserting the driver into kernel

Major and Minor number allocation: In Linux device drivers, the allocation of major and minor numbers is important for managing device nodes. Insert the module “sudo insmod driver.ko” and “cat /proc/devices”

Figure 17: Major and Minor number allocation

A. *Creating an I2C driver file, client file and bus file*

After inserting the appropriate driver, client, and bus modules into the kernel, use "tree/sys/bus/i2c/" to read the available buses.
Create an OLED device driver file integrating all the functions
Testing the driver:

- Build the driver by using Makefile (sudo make)
- Load the driver using sudo insmod driver.ko
- The display shows “Welcome to My MINI Project SHRINIDHI_1BM22LEL07”
- Unload the driver using sudo rmmod driver
- “Bye and Thank YoU!!!” is printed on the display and it is cleared after 1 second.

Figure 18: Created i2c-11 bus in i2c tree

```
pi@pi ~/Desktop/Programming/I2C_DeviceDriver_OLED
File  Edit  Tabs  Help
pi@pi:~ $ cd Desktop/
pi@pi:~/Desktop $ cd Programming/
pi@pi:~/Desktop/Programming $ cd I2C_DeviceDriver_OLED/
pi@pi:~/Desktop/Programming/I2C_DeviceDriver_OLED $ sudo insmod driver.ko
pi@pi:~/Desktop/Programming/I2C_DeviceDriver_OLED $
```

Figure 19: Inserting a driver module

Figure 20: Output is printed on OLED display

As the module is disconnected or removed the new statement is printed on the display.

```
pi@pi ~/Desktop/Programming/I2C_DeviceDriver_OLED
File  Edit  Tabs  Help
pi@pi:~ $ cd Desktop/
pi@pi:~/Desktop $ cd Programming/
pi@pi:~/Desktop/Programming $ cd I2C_DeviceDriver_OLED/
pi@pi:~/Desktop/Programming/I2C_DeviceDriver_OLED $ sudo insmod driver.ko
pi@pi:~/Desktop/Programming/I2C_DeviceDriver_OLED $ sudo rmmod driver
pi@pi:~/Desktop/Programming/I2C_DeviceDriver_OLED $ dmesg
```

Figure 21: Removing the module

Figure 22: New string is added to the output display

After executing the “sudo rmmod driver.ko” command, the oled display instantly displays the message "Bye and Thank YoU!!!". Approximately one second later, the screen turns off.

V. CONCLUSION

The implementation of the device driver for the I2C protocol in Linux involved the creation of essential files such as the I2C bus driver, I2C adapter, and I2C client driver. By effectively interfacing the Raspberry Pi with the SSD1306 OLED, the device driver was successfully executed. As a result, the master was able to reliably transmit a series of characters to the slave OLED device using the newly created device driver. This project demonstrates the successful integration of custom driver functionalities and the communication between components in the I2C communication.

REFERENCES

[1] R S Sandhya Devi, Savitha Sri N, P. Sivakumar "Developing Device Driver for Raspberry Pi" 2022 International Conference on Smart Generation Computing, Communication and Networking, Karnataka 2022.
[2] Bushra Rahman Ansari, Manjit Kaur, " Design and Implementation of Character Device Driver for Customized Kernel of ARM based Platform", International Conference on Current Trends in Computer, Electrical, Electronics and Communication IEEE (ICCTCEEC-2017).
[3] DEEPIKA, Neetika Yadav, "Design of Dual Master I2C Bus Controller and Interfacing it with Dc Motor" International Conference on Advances in Computing, Communication Control and Networking, ISBN: 978-1-5386-4119-4/18/$31.00, IEEE 2018.
[4] Evgeniy Kravtsunov, Andrey Kuyan, Sergey Radchenko, "Linux Kernel Drivers for I2C-manageable High Precision Power Source Based on ISL22317 and PCA9536 Chips" Proceeding of the 12th Conference Of Fruct Association, ISSN 2305-7254.
[5] Mary Grace Legaspi, Eric Peňa "I2C Communication Protocol: Understanding I 2 C Primer, PMBus, and SMBus" Vol 55, No 4—November 202.
[6] Device Driver basics: https://embetronicx.com/linux-device-driver-tutorials
[7] https://www.kernel.org/doc/html/latest/driver-api/
[8] Linux Device Drivers: https://lwn.net/Kernel/LDD3/
[9] https://www.tldp.org/LDP/lkmpg/2.6/html/
[10] I2C protocol: https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/

10.22214/IJRASET

45.98

IMPACT FACTOR:
7.129

IMPACT FACTOR:
7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY