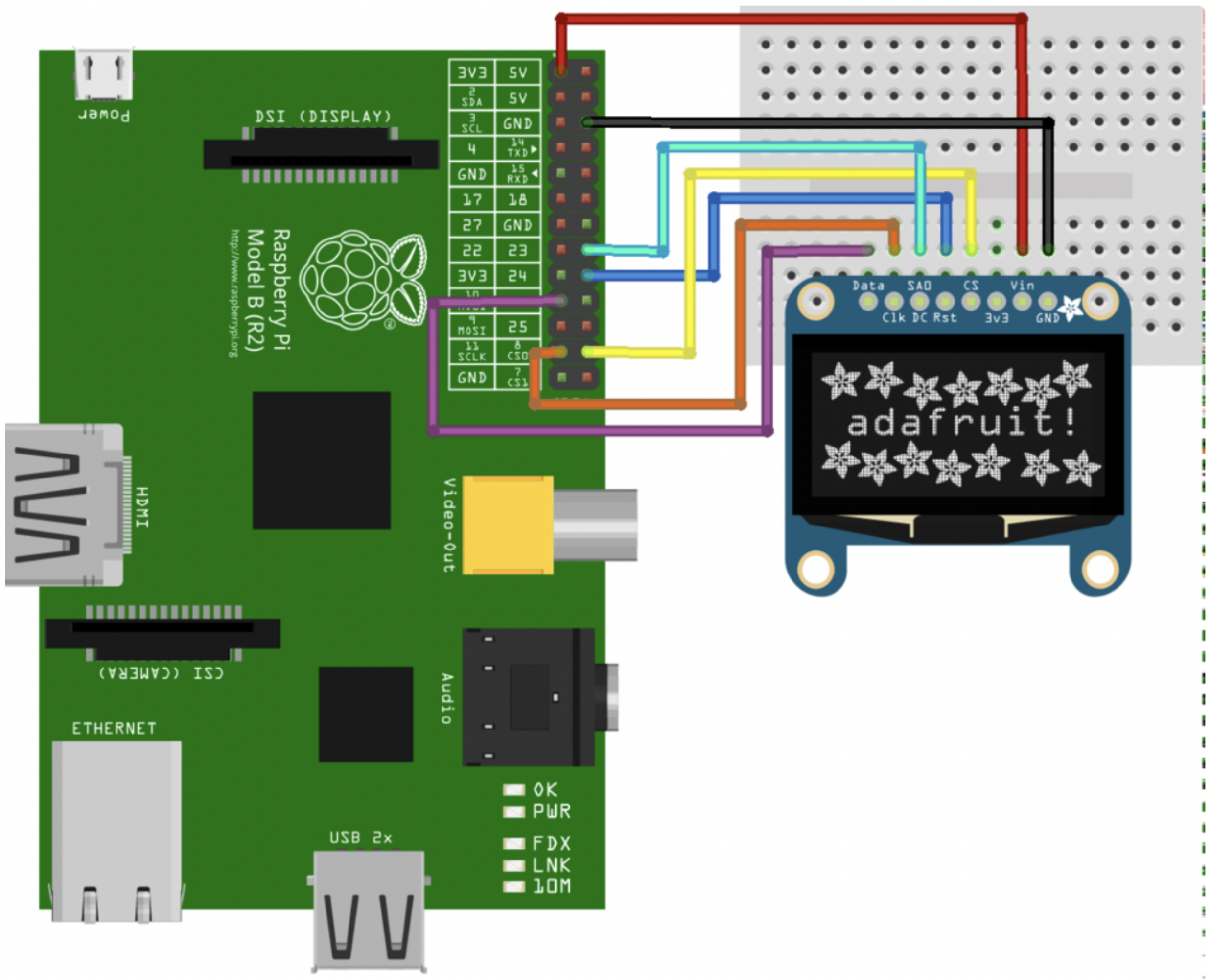
Project On Interfacing SSD1306 OLED using SPI Peripheral

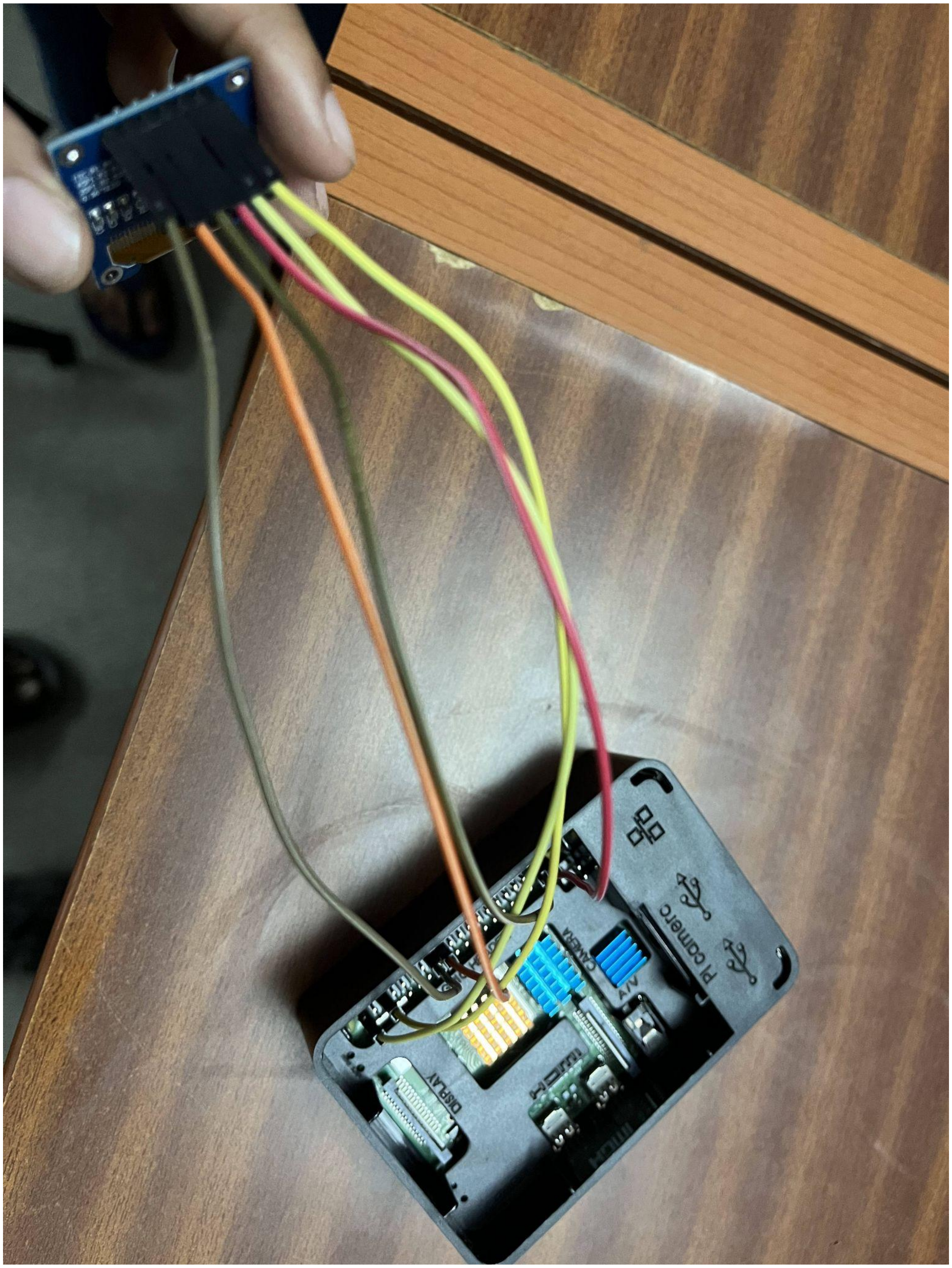
Anwesha Swain: **2020H1400172P**
Swati Purna Sahoo: **2020H1400177P**

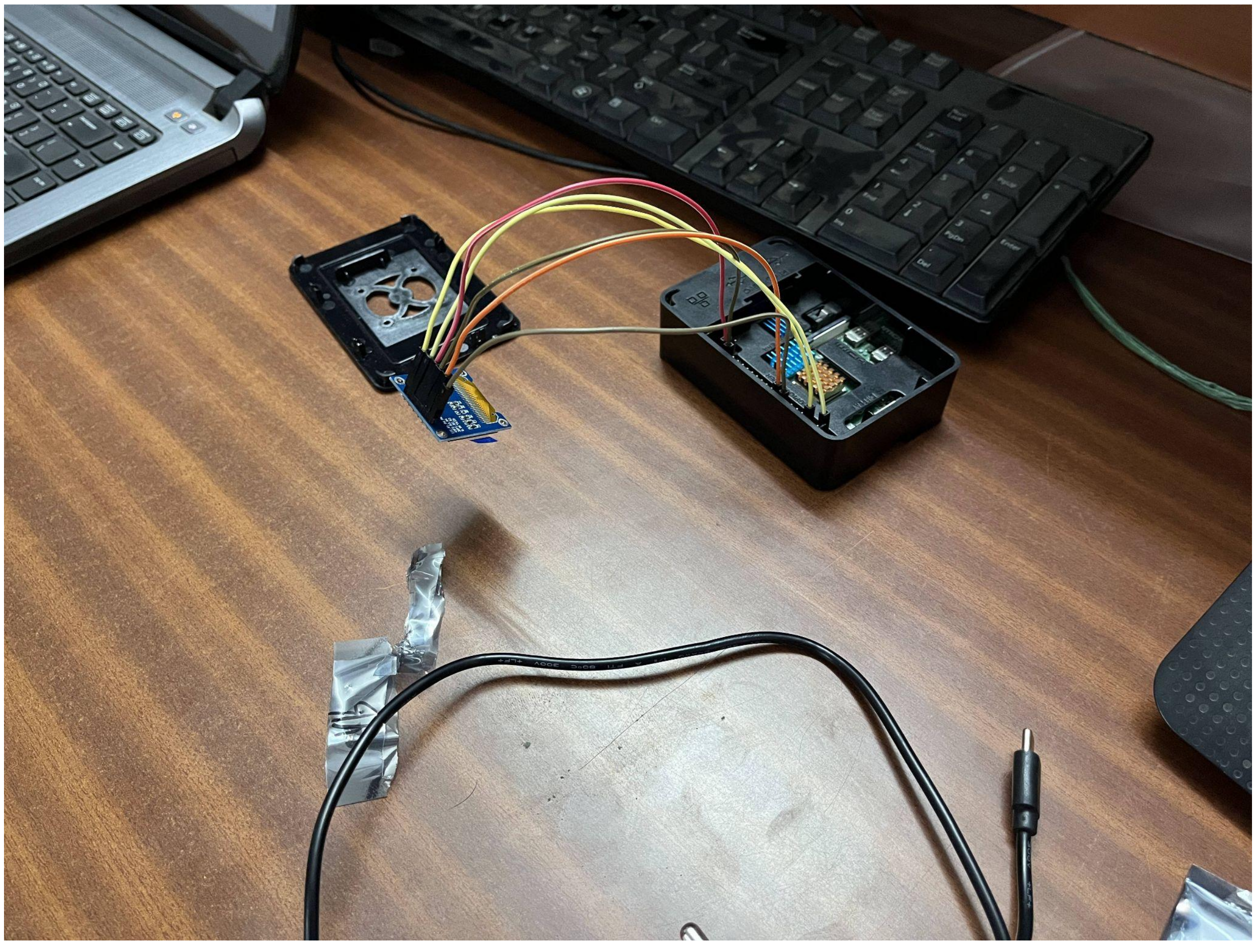
SUMMARY:

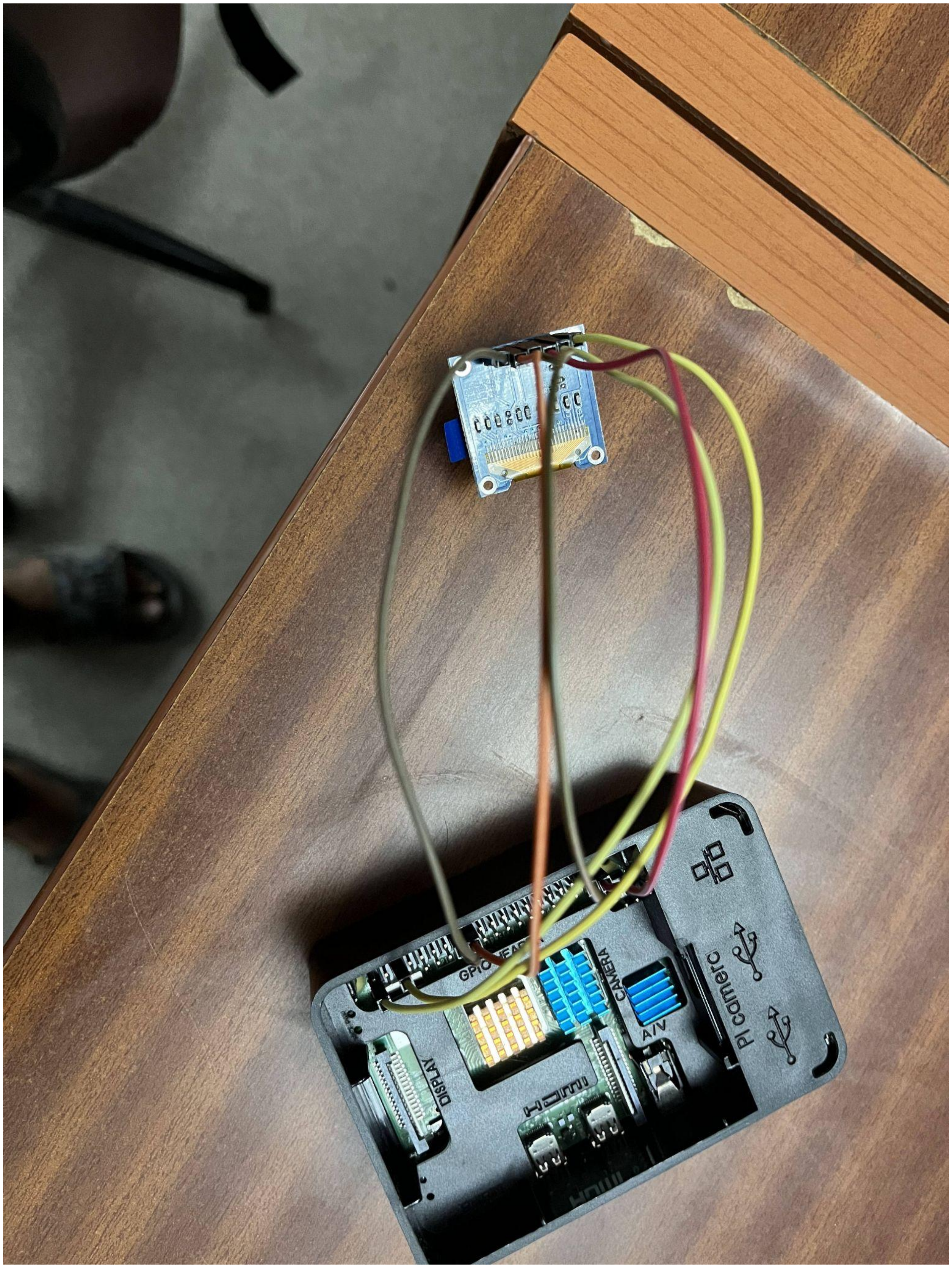
The project aims at interfacing SSD1306 oled display module with raspberry pi using the SPI peripheral.We will write driver level code and interface the SSD module to display the characters

INTERFACE DIAGRAM OF SSD1306 WITH RASPBERRY PI USING SPI PERIPHERAL:




DISPLAY
CAMERA
Pi camera
A/V




GPIO
CAMERA
A/V
DISPLAY
Pi camera

SPI Subsystem in Linux

The SPI device driver in Linux is mainly managed by the SPI subsystem, and it is divided into 3 sections.

- SPI Core
- SPI Controller Driver
- SPI Protocol Driver

SPI Core

The SPI core provides APIs for the definition of core data structures, registration, cancellation management of SPI controller drivers and device drivers. It is a hardware platform-independent layer that shields the differences of the physical bus controller downwards and defines a unified access strategy and interface. It provides a unified interface upwards. So, the SPI device driver can send and receive data through the SPI bus controller.

SPI Controller Driver

The SPI Controller driver is the platform-specific driver. So, each SoC manufacturer has to write this driver for their platform or MCU. controllers These SPI controller drivers may be built into System-On-Chip processors, and often support both Master and Slave roles. These drivers touch hardware registers and may use DMA or they can be GPIO bit bangers, needing just GPIO pins. Its responsibility is to implement a corresponding read and write method for each SPI bus in the system. Physically, each SPI controller can connect several SPI slave devices. When the system is turned on, the SPI controller driver is loaded first. A controller driver is used to support the reading and writing of a specific SPI bus.

SPI Protocol Driver in Linux Kernel

Steps that involve writing the SPI protocol device driver are given below.

1. Get the SPI Controller driver
2. Add the slave device to the SPI Controller
3. Configure the SPI
4. Now the driver is ready. So you can transfer the data between master and slave.
5. Once you are done, then remove the device.

Get the SPI Controller driver

First, we need to get the controller driver. To do that, we can use the below API. Before using that API, we need to know which SPI bus we are using.

struct spi_controller * spi_busnum_to_master(u16 bus_num)

It returns a pointer to the relevant spi_controller (which the caller must release), or NULL if there is no such master registered.

Add the slave device to the SPI Controller

In this step, you need to create a **spi_board_info** structure that has the slave device's details.

Once we have filled the above structure, then we can use the below API to add the slave device to the SPI controller driver.

struct spi_device * spi_new_device(struct spi_controller *ctlr, struct spi_board_info *chip)

It returns the new device, or NULL.

Configure the SPI

Till now, we have added the slave device to the controller driver. If we change any mode or clock frequency, then we have to call this below API to take effect on the bus.

int spi_setup(struct spi_device *spi)

Return
It returns zero on success, else a negative error code.

Message Transfer

We have done our initialization. Now we can transfer the data using the below APIs.

This function will be blocked until the master completes the transfer.
int spi_sync_transfer(struct spi_device *spi, struct spi_transfer *xfers, unsigned int num_xfers)
It returns zero on success, else a negative error code.

This API is used to transfer the data asynchronously. This call may be used **in_irq** and other contexts which can't sleep,

int spi_async(struct spi_device *spi, struct spi_message *message)

It returns zero on success, else a negative error code.

This API is used to write the data and followed by a read. This is synchronous.
int spi_write_then_read(struct spi_device * spi, const void * txbuf, unsigned n_tx, void * rxbuf, unsigned n_rx)

Remove the device

Using the below API, you can unregister the slave device.

void spi_unregister_device(struct spi_device *spi)

Loading and Unloading Modules

After the module is built, the next step is loading it into the kernel. As we've already pointed out, *insmod* does the job for you. The program loads the module code and data into the kernel, which, in turn, performs a function similar to that of *ld*, in that it links any unresolved symbol in the module to the symbol table of the kernel. Unlike the linker, however, the kernel doesn't modify the module's disk file, but rather an in-memory copy. *insmod* accepts a number of command-line options (for details, see the manpage), and it can assign values to parameters in your module before linking it to the current kernel. Thus, if a module is correctly designed, it can be configured at load time; load-time configuration gives the user more flexibility than compile-time configuration.

Device drivers in user space

Traditionally, packet-processing or data-path applications in Linux have run in the kernel space due to the infrastructure provided by the Linux network stack. Frameworks such as netdevice drivers and netfilters have provided means for applications to directly hook into the packet-processing path within the kernel.

However, a shift toward running data-path applications in the user-space context is now occurring. The Linux user space provides several advantages for applications, including more robust and flexible process management, standardized system-call interface, simpler resource management, a large number of libraries for XML, and regular expression parsing, among others. It also makes applications more straightforward to debug by providing memory isolation and independent restart. At the same time, while kernel-space applications need to conform to General Public License guidelines, user-space applications are not bound by such restrictions.

User-space data-path processing comes with its own overheads. Since the network drivers run in kernel context and use kernel-space memory for packet storage, there is an overhead of copying the packet data from user-space to kernel-space memory and vice versa. Also, user/kernel-mode transitions usually impose a considerable performance overhead, thereby violating the low-latency and high-throughput requirements of data-path applications.

UIO drivers

Linux provides a standard UIO (User I/O) framework for developing user-space-based device drivers. The UIO framework defines a small kernel-space component that performs two key tasks:

a. Indicate device memory regions to user space.

b. Register for device interrupts and provide interrupt indication to user space.

The kernel-space UIO component then exposes the device via a set of sysfs entries like `/dev/uioXX` . The user-space component searches for these entries, reads the device address ranges and maps them to user space memory.

The user-space component can perform all device-management tasks including I/O from the device. For interrupts however, it needs to perform a blocking `read()` on the device entry, which results in the kernel component putting the user-space application to sleep and waking it up once an interrupt is received.

GENERAL DESCRIPTION of SSD1306

SSD1306 is a single-chip CMOS OLED/PLED driver with controller for organic / polymer light emitting diode dot-matrix graphic display system. It consists of 128 segments and 64commons. This IC is designed for Common Cathode type OLED panel. The SSD1306 embeds with contrast control, display RAM and oscillator, which reduces the number of external components and power consumption. It has 256-step brightness control. Data/Commands are sent from general MCU through the hardware selectable 6800/8000 series compatible Parallel Interface, I 2 C interface or Serial Peripheral Interface. It is suitable for many compact portable applications, such as mobile phone sub-display, MP3 player and calculator, etc.

FEATURES

• Resolution: 128 x 64 dot matrix panel

• Power supply o VDD = 1.65V to 3.3V for IC logic o VCC = 7V to 15V for Panel driving

• For matrix display

- o OLED driving output voltage, 15V maximum
- o Segment maximum source current: 100uA
- o Common maximum sink current: 15mA
- o 256 step contrast brightness current control

• Embedded 128 x 64 bit SRAM display buffer

• Pin selectable MCU Interfaces:

- o 8-bit 6800/8080-series parallel interface
- o 3 /4 wire Serial Peripheral Interface
- o I 2 C Interface

• Screen saving continuous scrolling function in both horizontal and vertical direction

• RAM write synchronization signal

• Programmable Frame Rate and Multiplexing Ratio

• Row Re-mapping and Column Re-mapping

• On-Chip Oscillator

• Chip layout for COG & COF

• Wide range of operating temperature: -40°C to 85°C