ASSIGNMENT 2

ON

SYSTEM SOFTWARE

TOPICS :- 1. DEVICE DRIVERS
2. WORKING OF DEVICE DRIVERS
3. DEVELOPMENT OF USB DEVICE DRIVER

REFERENCES :-

1. en.wikipedia.org/wiki/Device_drivers
2. D.M. Dhamdhere, Systems Programming and Operating Systems, Second Revised Edition
3. Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, Linux Device Drivers, Third Edition

SUBMISSION DATE :- 04-11-2019

Done By,
Harikrishnan.V
CS-5A
Roll No: 27

Device Drivers

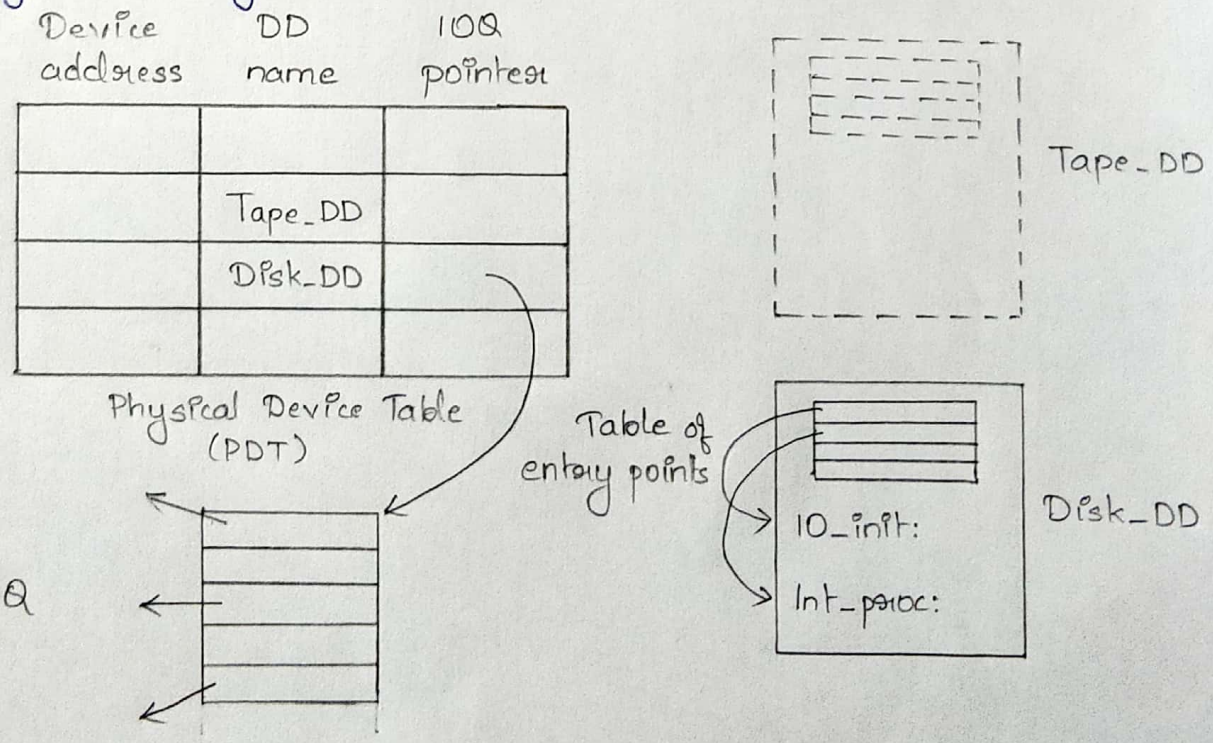
In computing, a device driver is a computer program that operates or controls a particular type of device that is attached to a computer. A driver provides a software interface to hardware devices, enabling operating systems & other computer programs to access hardware functions without needing to know precise details about the hardware being used.

A driver communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the driver receives the data back from the device, the driver may invoke routines in the original calling program. Drivers are hardware dependent & OS specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

The main purpose of device drivers is to provide abstraction by acting as a translator between a hardware device & the applications or operating systems that use it. Programmers can write higher-level application code independently of whatever specific hardware the end-user is using. At a lower level, a device driver implementing these functions would communicate to the particular serial port controller installed on a user's computer.

Working of Device Drivers

The physical IOCS handles I/O initiation, I/O completion & error recovery for all classes of I/O devices within the system. Consequently, addition of a new class of I/O devices requires changes to the physical IOCS, which can be both complex & expensive because the physical IOCS may be a part of the kernel. Modern operating systems overcome this problem through a different arrangement. The physical IOCS provides only generic support for I/O operations & invokes a specialized device driver (DD) module for handling device-level details for a specific class of devices. This arrangement enables new classes of I/O devices to be added to the system without having to modify the physical IOCS. Device drivers are loaded by the system boot procedure depending on the classes of I/O devices connected to the computer. Alternatively, device drivers can be loaded whenever needed during operation of the OS. This feature is particularly useful for providing a plug-and-play capability.



The entry of a device in the physical device table (PDT) shows the name of its device driver in the DD name field. The Disk-DD, the device driver for the system disk, has been loaded at system boot time. The Tape-DD would be loaded on demand. A device driver contains functionalities of the 4 physical IOCS modules shown in fig, namely, I/O scheduler, I/O initiator, I/O completion handler, & error recovery. A table of entry points located at the start of its code contains start addresses of these functionalities.

When the physical IOCS is invoked for initiating an I/O operation, it locates the PDT entry of the device & performs the generic function of entering details of the I/O operation into the IOQ of the device. It now consults the DD name field of the PDT entry, obtains the identity of the device driver & loads the device driver in memory if it is not already in memory. It now obtains the address of the entry point for I/O initiator in the device driver by following the standard conventions & passes control to it. The device driver performs I/O initiation processing & returns control to the physical IOCS, which passes control to the process scheduler. When the physical IOCS is invoked implicitly at an I/O interrupt, it performs similar actions to identify the device driver entry point for handling interrupts & passes control to it. After servicing the interrupt, the device driver returns control to the physical IOCS, which passes it to the process scheduler.

Development of USB Device Drivers

The most basic form of USB communication is through something called an endpoint. A USB endpoint can carry data in only 1 direction, either from the host computer to the device (called an OUT endpoint) or from the device to the host computer (called an IN endpoint).

A USB endpoint can be 1 of 4 different types that describe how the data is transmitted:

(i) CONTROL

Control endpoints are used to allow access to different parts of the USB device. They are commonly used for configuring the device, retrieving information about the device, sending commands to the device, or retrieving status reports about the device.

(ii) INTERRUPT

Interrupt endpoints transfer small amounts of data at a fixed rate every time the USB host asks the device for data. These endpoints are the primary transport method for USB keyboards & mice.

(iii) BULK

Bulk endpoints transfer large amounts of data. These endpoints are usually much larger (they can hold more characters at once) than interrupt endpoints.

(iv) ISOCHRONOUS

Isochronous endpoints also transfer large amounts of data, but the data is not always guaranteed to make it through. These endpoints are used in devices that can handle loss of data, & rely more on keeping a constant stream of data flowing.

Control & bulk endpoints are used for asynchronous data transfer, whenever the driver decides to use them. Interrupt & isochronous endpoints are periodic.

USB endpoints are described in the kernel with the structure struct usb_host_endpoint. This structure contains the real endpoint information in another structure called struct usb_endpoint_descriptor. The fields of this structure that drivers care about are:

(i) bEndpointAddress

This is the USB address of this specific endpoint. Also included in this 8-bit value is the direction of the endpoint.

(ii) bmAttributes

This is the type of endpoint. The bitmask USB_ENDPOINT_XFERTYPE_MASK should be placed against this value in order to determine if the endpoint is of type USB_ENDPOINT_XFER_ISOC, USB_ENDPOINT_XFER_BULK, or of type USB_ENDPOINT_XFER_INT.

(iii) wMaxPacketSize

This is the maximum size in bytes that this endpoint can handle at once. For high-speed devices, this field can be used to support a high-bandwidth mode for the endpoint by using a few extra bits in the upper part of the value.

(iv) bInterval

If this endpoint is of type interrupt, this value is the interval setting for the endpoint - that is, the time between interrupt requests for the endpoint. The value is represented in ms.

USB endpoints are bundled up into interfaces. USB interfaces handle only 1 type of a USB logical connection, such as a mouse, a keyboard, or an audio stream.

USB interfaces are described in the kernel with the struct_usb_interface structure. This structure is what the USB core passes to USB drivers & is what the USB driver then is in charge of controlling. The important fields in this structure are:

(i) struct usb_host_interface *altsetting

An array of interface structures containing all of the alternate settings that may be selected for this interface.

(ii) unsigned num_altsetting

The number of alternate settings pointed to by the altsetting pointer.

(iii) struct usb_host_interface *cur_altsetting

A pointer into the array altsetting, denoting the currently active setting for this interface.

(iv) int minor

If the USB driver bound to this interface uses the USB major number, this variable contains the minor number assigned by the USB core to the interface.

USB interfaces are themselves bundled up into configurations. A USB device can have multiple configurations & might switch between them in order to change the state of the device.

A USB device driver commonly has to convert data from a given struct usb_interface structure into a struct usb_device structure that the USB core needs for a wide range of function calls. To do this, the function interface_to_usbdev is provided.

Due to the complexity of a single USB physical device, the representation of that device in sysfs is also quite complex. Both the physical USB device (as represented by a struct usb_device) and the individual USB interfaces (as represented by a struct usb_interface) are shown in sysfs as individual devices.

The approach to writing a USB device driver is similar to a pci_driver: the driver registers its driver object with the USB subsystem & later uses vendor & device identifiers to tell if its hardware has been installed.

The struct usb_device_id structure provides a list of different types of USB devices that this driver supports. This list is used by the USB core to decide which driver to give a device to. The struct usb_device_id structure is defined with the following fields:

(i) __u16 match_flags

Determines which of the following fields in the structure the device should be matched against.

(ii) __u16 idVendor

The USB vendor ID for the device.

(iii) __u16 idProduct

The USB product ID for the device.

(iv) __u16 bcdDevice_lo

__u16 bcdDevice_hi

Define the low & high ends of the range of the vendor-assigned product version number.

(v) __u8 bDeviceClass

__u8 bDeviceSubClass

__u8 bDeviceProtocol

Define the class, subclass, & protocol of the device, respectively.

(vi) __u8 bInterfaceClass

__u8 bInterfaceSubClass

__u8 bInterfaceProtocol

These define the class, subclass, & protocol of the individual interface, respectively.

(vii) kernel_ulong_t driver_info

It holds information that the driver can use to differentiate the different devices from each other in the probe call-back function to the USB driver.

There are a number of macros that are used to initialize this structure:

(ii) USB_DEVICE(vendor, product)

Creates a struct usb_device_id that can be used to match only the specified vendor & product ID values.

(iii) USB_DEVICE_VER(vendor, product, lo, hi)

Creates a struct usb_device_id that can be used to match only the specified vendor & product ID values within a version range.

(iii) USB_DEVICE_INFO(class, subclass, protocol)

Creates a struct usb_device_id that can be used to match a specific class of USB interfaces.

So, for a simple USB device driver that controls only a single USB device from a single vendor, the struct usb_device_id table would be defined as:

```
/* table of devices that work with this driver */
static struct usb_device_id skel_table [] = {
    { USB_DEVICE (USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
    { }                         /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, skel_table);
```

The main structure that all USB drivers must create is a struct usb_driver. It must be filled out by the USB driver & contains a number of function callbacks & variables that describe the USB driver to the USB core code:

(i) struct module *owner

(ii) const char *name

(iii) const struct usb_device_id *id_table

(iv) int (*probe) (struct usb_interface *intf, const struct usb_device_id *id)

(v) void (*disconnect) (struct usb_interface *intf)

So, to create a value struct usb_driver structure, only 5 fields need to be initialized:

```
static struct usb_driver skel_driver = {
        .owner = THIS_MODULE,
        .name = "skeleton",
        .id_table = skel_table,
        .probe = skel_probe,
        .disconnect = skel_disconnect,
};
```

When the USB driver is to be unloaded, the struct usb_driver needs to be unregistered from the kernel. This is done with a call to usb_deregister_driver. When this call happens, any USB interfaces that were currently bound to this driver are disconnected, & the disconnect function is called for them.

```
static void __exit usb_skel_exit (void)
{
    /* deregister this driver with the USB subsystem */
    usb_deregister (& skel_driver);
}
```