Linux Driver, Kernel Programming.

AMD
together we advance_

Agenda

1. What is a Driver
2. Linux Driver Architecture and APIs
3. Linux Kernel Module
4. Transferring Data Within the Kernel
5. Reference

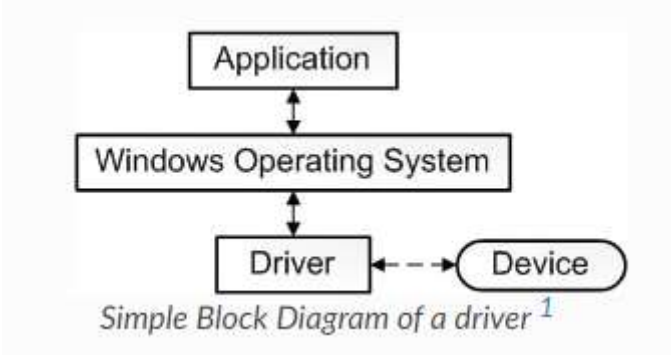
What is a Driver

Software drivers play a critical role in how we use computers and electronics on a daily basis, yet most users never consider the complexity of driver development. As a very brief introduction, we will conceptualize a driver as any software component that lets the operating system or OS (such as Linux or Windows) communicate with an external device, like a keyboard. These devices can represent either physical hardware or other software tools. A driver allows user applications to interact and exchange data with other devices through the OS.

Device drivers are parts of the operating system that facilitate the usage of hardware devices via certain programming interfaces so that software applications can control and operate the devices. As each driver is specific to a particular operating system, you need separate Linux, Windows, or Unix device drivers to enable the use of your device on different computers (this is why a career in driver development and embedded systems is often lucrative).

The first step in driver development is to understand the differences in the way each operating system handles its drivers, underlying driver model, and architecture it uses, as well as available development tools. For example, the Linux driver model is very different from the Windows one. While Windows emphasizes abstraction and separation between drivers and the host OS, Linux device drivers are often embedded within the OS kernel itself, as they are not built off a stable API. Each of these models has its own set of advantages and drawbacks, which is important to keep in mind while writing and analyzing drivers for each major OS.

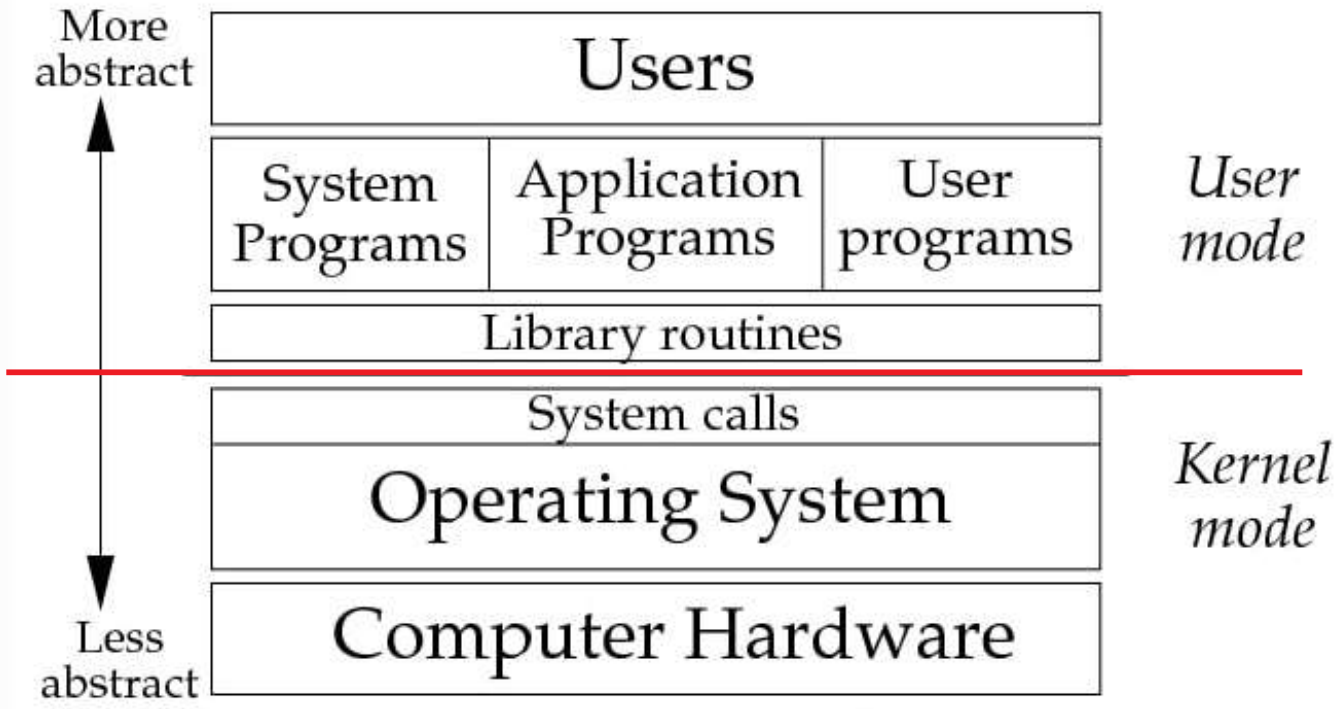
Xilinx's DMA PCIe Drivers are available for both Windows and Linux. However, throughout this article and subsequent software tutorials, **we will focus on the Linux OS and similar Unix distributions for its versatility and open-source nature.**



Simple Block Diagram of a driver [1]

Throughout this article, we will constantly reference the Linux kernel, kernel mode, and virtual memory. More information about these topics can be found in this article.

Linux Driver Architecture and APIs



Abstracted BD of a typical OS [2]

Linux device drivers support three kinds of devices:

- Character devices that implement a byte stream interface
- Block devices that host filesystems and perform IO with multibyte blocks of data
- Network interfaces are used for transferring data packets through the network

An important aspect of a driver is the API or Application Programming Interface it is built upon. An API is a software intermediary that allows two applications to communication with each other. Essentially, an API is a messenger that both delivers requests and subsequent responses between two applications. In the block diagram above, each layer provides an API as a set of functions/commands that the layer itself provides. We are mentioning APIs to create the distinction between drivers and APIs: drivers are low-level sections of code that run within the OS kernel itself and allow us to talk to hardware directly, while APIs are higher-level abstraction that allow us to utilize drivers within a human-understandable programming environment. APIs are often used in applications that are outside the scope of this article,

Linux Kernel Modules

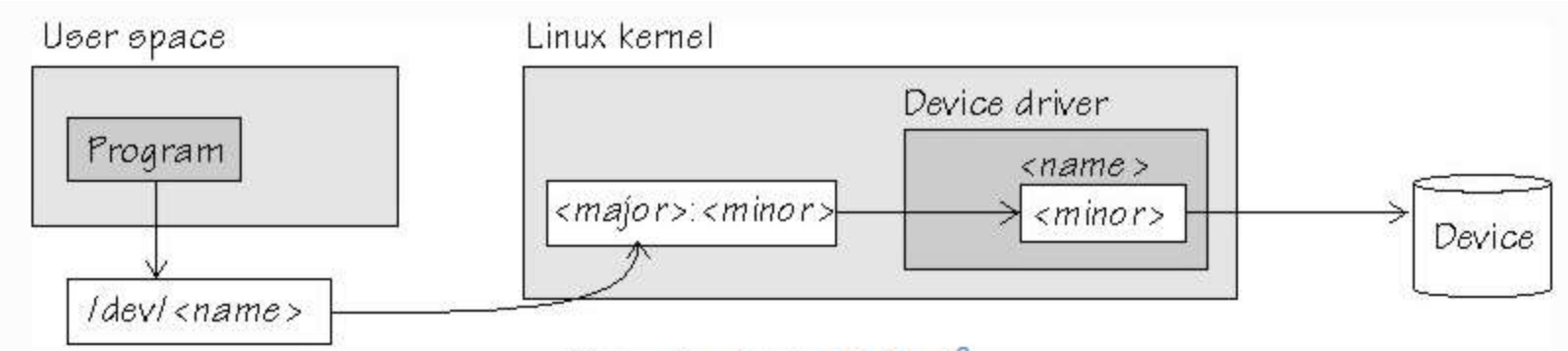
At a module's initalization, the Linux device driver lifetime is managed by the kernel module's module_init and module_exit functions, which are called when the module is loaded or unloaded. They are responsible for registering the module to handle device requests using the internal kernel interfaces. The module has to create a device file (or a network interface), specify a numerical identifier of the device it wishes to manage, and register a number of callbacks to be called when the user interacts with the device file.

On Linux, user applications access the devices via file system entries, usually located in the /dev directory. The module creates all necessary entries during module initialization by calling kernel functions like register_chrdev. An application issues an open system call to obtain a file descriptor, which is then used to interact with the device. This call (and further system calls with the returned descriptor like read, write, or close) are then dispatched to callback functions installed by the module into structures like file_operations or block_device_operations.

The device driver module is responsible for allocating and maintaining any data structures necessary for its operation. A file structure passed into the file system callbacks has a private_data field, which can be used to store a pointer to driver-specific data. The block device and network interface APIs also provide similar fields.

While applications use file system nodes to locate devices, Linux uses a concept of *major* and *minor* numbers to identify devices and their drivers internally. A major number is used to identify device drivers, while a minor number is used by the driver to identify devices managed by it. The driver has to register itself in order to manage one or more fixed major numbers or ask the system to allocate some unused number for it.

Currently, Linux uses 32-bit values for major-minor pairs, with 12 bits allocated for the major number allowing up to 4096 distinct drivers. The major-minor pairs are distinct for character and block devices, so a character device and a block device can use the same pair without conflicts. Network interfaces are identified by symbolic names like eth0, which are again distinct from major-minor numbers of both character and block devices.



Major-minor Devices in Linux [3]

Transferring Data Within the Kernel

Both Linux and Windows support three ways of transferring data between user-level applications and kernel-level drivers:

- **Buffered Input-Output** which uses buffers managed by the kernel. For write operations, the kernel copies data from the user-space buffer into a kernel-allocated buffer and passes it to the device driver. Reads are the same, with kernel copying data from a kernel buffer into the buffer provided by the application.
- **Direct Input-Output** which does not involve copying. Instead, the kernel pins a user-allocated buffer in a physical memory so that it remains there without being swapped out while data is in progress.
- **Memory Mapping** can also be arranged by the kernel so that the kernel and user-space applications can access the same pages of memory using distinct addresses.

Linux provides a number of functions like clear_user, copy_to_user, strncpy_from_user, and some others to perform buffered data transfers between the kernel and user memory. These functions validate pointers to data buffers and handle all details of the data transfer by safely copying the data buffer between memory regions.

However, drivers for block devices operate on entire data blocks of known size, which can be simply moved between the kernel and user address spaces without copying them. This case is automatically handled by the Linux kernel for all block device drivers. The block request queue takes care of transferring data blocks without excess copying, and the Linux system call interface takes care of converting file system requests into block requests.

Finally, the device driver can allocate some memory pages from kernel address space (which is non-swappable) and then use the remap_pfn_range function to map the pages directly into the address space of the user process. The application can then obtain the virtual address of this buffer and use it to communicate with the device driver.

References

1
Driver introduction from Microsoft.
https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/what-is-a-driver-
2
More about operating systems from this computer architecture .
https://minnie.tuhs.org/CompArch/Lectures/week07.html
3
More about device nodes in this IBM .
https://www.ibm.com/docs/ja/linux-on-systems?topic=linuxonibm/com.ibm.linux.z.ludd/ludd_c_udev.html


AMD