

### Accessing I/O devices

An application is often only as useful as the I/O devices with which it communicates. Without being able to communicate with these peripheral devices, the functionality and expandability of the computer is greatly hindered. Therefore, it is important to understand how an operating system handles these devices and how to control this communication.

First, it is necessary to learn the process of how a device is connected to a computer. Due to the enormous range of peripheral devices, there is no standard way in which an operating system can converse with the device. For example, the data transfer rate to a satellite system may be much different than that used to synchronize to a handheld organizer. The memory needed onboard the device may also have very different requirements. This necessitates a way in which the CPU can adjust according to the device. In order to accomplish this task, there exist device drivers. Each device will have a driver that informs it how it should communicate with the computer. This driver file will already have the requirements and limitations of the peripheral chip (the I/O device) written into it, so that the OS only needs to know how to communicate with the driver.

It may be helpful to look at the system in parts. The user will give a command to an application, which is then managed and “supervised” by the OS. The operating system will then follow that command to the necessary device driver. The OS, in a sense, will hand over this command to the driver, and allow it control (most operating systems do not just give control to a device without certain permissions or security procedures). The device can then take this information and translate it into what the peripheral chip needs in order to accomplish the task. Upon this “translation” it is presumed that the device will then be able to complete the task.

Keeping this flow of events in mind, when the time comes to write a device driver, the process of knowing what is needed is greatly simplified. The programmer’s

task is to mediate between these different layers. Therefore, he should first look at how to manipulate the OS into allowing him control of the device. As mentioned previously, most operating systems will not allow access to a peripheral chip without some constraint. For example, in a system like Linux, there are C commands such as “ioperm” that request permission from the OS to use certain addresses. Without this permission, the OS will not allow the addresses to be randomly accessed.

Once a programmer learns how to gain this permission from the specific OS, he can then focus on how to initialize or communicate with the peripheral chip. Each chip will be different, and will have various initialization steps. The best way to find this information is to locate the documentation for that specific chip (assuming there is documentation for it). The documentation should provide the information on how to initialize the chip and ready it for communication. For example, when initializing the 16550 chip, it is necessary to set a divisor latch bit. The documentation has a useful table showing how to accomplish this. (See the appendix for the reference to this document).

After writing out all the necessary steps for communicating with the chip, the programmer should then be able to gain access to that device. These are the basic concepts behind accessing an I/O device. Although it may seem tedious at times to filter through these various layers, it proves very useful in managing a complex system, and for preventing damage to a system from an unaware user.

## **Appendix**

- 1- Michael R. Sweet - *Serial Programming Guide for POSIX Operating System*- Copyright 1994-1999, All Rights Reserved. 5th Edition, 2nd Revision
- 2- Alessandro Rubini, Jonathan Corbet - *Linux Device Drivers, 2nd Edition* - 2nd Edition June 2001
- 3- National Semiconductor – *PC 16550D Universal Asynchronous Receiver/Transmitter with FIFOs* – June 1995