

Device Drivers: Their Function in an Operating System

Robert Milton Underwood, Jr.

© 2000

Device Drivers: Their Function in an Operating System

A device driver is a program routine that links a peripheral device to an operating system of a computer. It is essentially a software program that allows a user to employ a device, such as a printer, monitor, or mouse (Levenson & Hertz, 1994). It is written by programmers who comprehend the detailed knowledge of the device's command language and characteristics and contains the specific machine language necessary to perform the functions requested by the application (The Computer Language Company, 2000). When a new hardware device is added to the computer, such as a CD-ROM drive, a printer, or a sound card, its driver must be installed in order to run it. The operating system "calls" the driver, and the driver "drives" the device.

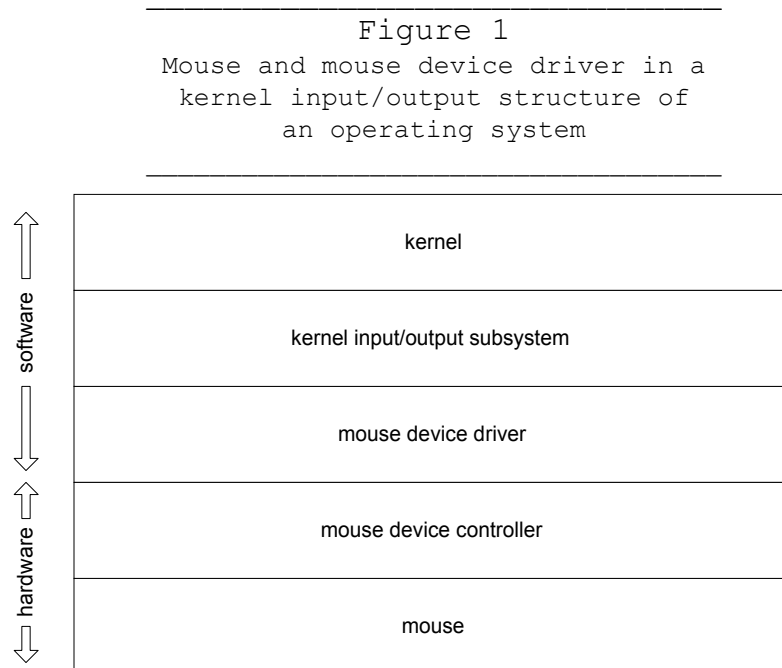
In Windows, for example, everything that is seen on the screen is the result of the display driver (video driver). The display driver effectuates the visual appearance of the screen according to the precise commands that Windows issues to it (The Computer Language Company, 2000).

The driver is the link between the operating system and the peripheral device. If the peripheral device is changed, or if a bug is found in the driver, the driver must also be changed. A new version of the driver is then written and released by the manufacturer of the device. Updated drivers are usually made available on the Web sites and/or bulletin boards of vendors.

The basic input/output (I/O) hardware features, such as ports, buses, and device controllers, accommodate a wide variety of I/O devices. To encapsulate the details and unique features of different devices, the kernel of an operating system is set up to use device driver modules (Silberschatz & Galvin, 1999). The device drivers present a uniform device-access interface to the I/O subsystem.

Each of the different types of I/O devices is accessed through a standardized set of functions--an interface (Silberschatz & Galvin, 1999). The tangible differences are encapsulated in kernel modules (i.e., device drivers) that internally are customized for each device, but that export and utilize one of the standard interfaces.

A device driver sets the direct memory access (DMA) control registers to use appropriate source and destination addresses, and transfer length (Silberschatz & Galvin, 1999). The DMA controller is then instructed to begin the I/O operation. Refer to Figure 1 to see how a device driver (a mouse driver in this example) relates to the structure of the operating system.

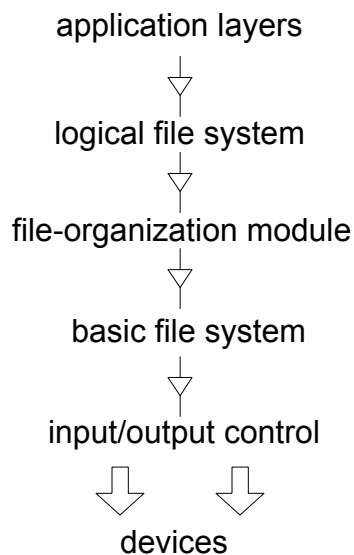


Device drivers are saved as files, and are called upon when a particular peripheral or hardware device is needed. On the Macintosh, for instance, they are stored in the Extensions

folder (Langer, 2000). Like extensions, their features are preset and cannot be modified. Once they are installed, the devices they control become available for use.

To provide an efficient and convenient access to the hard disk, the operating system requires the file system to allow the data to be stored, located, and retrieved easily. The file system is composed of several different levels. The lowest level (see Figure 2) is the input/output control, and consists of device drivers and interrupt handlers to transfer information between the memory and the hard disk. A device driver is basically a translator (Silberschatz & Galvin, 1999). Its input consists of high-level commands, and its output consists of low-level, hardware-specific instructions, which are utilized by the hardware controller, which interfaces the I/O device to the rest of the operating system. The device driver usually writes specific bit patterns to designated locations in the I/O controller's memory to let the controller know on which device location to act and what subsequent actions to provide.

Figure 2
Layered file system of a computer



The effort required to support different brands of peripheral devices was one of the major reasons DOS gave way to the Windows operating system. For example, with DOS systems, in order to provide reliable control over the printing of a document, each application provides its own set of printer drivers for the most popular printers. But with PCs running Windows, and the Macintosh, the printer driver is installed by the operating system, not by each application. After initial installation, all applications gain access to the printer through the operating system and its one driver for the specific printer that will be used with the system.

Part of the reason for Windows' success as a replacement for DOS as the dominant PC operating system format lies in two key architectural strategies: standardization and encapsulation. With standardization programming interfaces, applications can take advantage of the advanced features offered by Windows. Although DOS had a certain level of standardization with its INT 21h interface (Norton, 1992), it was not completely effective for many DOS programs. With DOS, for example, there were no standards for drawing figures on the screen in graphics modes, or from reading from and writing to the COM port. By standardizing the interfaces at higher levels of the operating system, Windows provided a consistent and complete program interface.

There are different perspectives from which to consider the technical aspects of device drivers and the role they have in an operating system. One perspective is that of a System Administrator. From a System Administrator's viewpoint using Windows, a device driver is a set of instructions that coordinates the operating system with hardware such as printers, storage units, network equipment, fax machines, scanners and digital cameras. The WDM is a model for device drivers that can run on both Windows98 and Windows NT; the WDM drivers are the same files for both operating systems (Ziff-Davis, 1998). In theory, the WDM should greatly

reduce the system administration burden of maintaining multiple device driver versions, and is also an important concern for hardware vendors who are encouraged to create devices that are consistently recognized in both versions of Windows.

The WDM has also been helpful with regards to Windows2000, as it has allowed a common set of device drivers for both Windows2000 and its predecessor. With the release of Windows2000, the objective of standardization among Windows device drivers has been largely achieved.

In the case of streaming media software, the WDM has shifted the processing from the user mode to WDM kernel streaming. The objective was to improve overall speed and performance. An application must be specifically written for WDM to take advantage of this architecture. The same applies to the new WDM Still Image Architecture for specific support of digital cameras and scanners.

Programmers have a slightly different perspective from which to consider device drivers and their role in an operating system. A Windows2000 programmer may consider a device driver as code that communicates between Windows2000 and hardware such as a modem, network card, or printer. Without it, a device is not recognized by Windows2000. Some drivers are shipped with Windows2000. Other drivers must be obtained from the hardware manufacturer (Williams & Walla, 2000).

A Windows programmer will also recognize that Windows is device-independent. Programming in Windows is tougher than programming in DOS in several ways, but there are also many aspects that are easier, device independence being one of them. Windows device drivers are written in such a way that you can use the exact same drawing code for any type of video driver. (Edson, 1993).

Writing a driver for a PC peripheral under Windows is relatively easy for programmers because these operating systems allow direct access to the PC's I/O ports through programming commands. One caveat for programmers is that if they make a programming error and write data to the wrong address, the system can crash. Windows NT is less likely to crash, but a premium is paid for added stability in the form of the requirement of strict conformity to rules for controlling mapped I/O ports, and there is a loss of direct access to those ports. To control I/O ports under Windows NT, applications must communicate through a kernel device driver. Well-debugged device drivers help make Windows NT less likely to crash (Gaudin, 1999). With Windows NT, a shareware ActiveX control and its associated dynamic link library (DLL) allow the user to control plug-in cards and other peripherals (Bates, 2000). Some I/O cards come with a Windows NT driver, so not everyone will need the shareware control.

Programmers also try to continually improve the functionality of device drivers with regards to their relationship among numerous devices. Being able to have a number of devices communicate with each other requires a common language and protocol (Cravotta, 2000). For device connectivity to be successful, devices must be able to reliably and transparently exchange services. The connectivity challenge lies in deciding how to exchange services once a connection is established. Connectivity technology such as Jini, supported by Sun Microsystems, may be able to solve this problem by addressing the $N \times N$ device-driver problem. The standard connectivity model requires a total of $N \times N$ drivers to support the varying combination of devices and service providers, and several vendors have tried to accomplish this objective. Jini is a Java-code protocol for the creation of an exchange of services. The basic concept behind Jini technology is that a device active locates services, downloads them for use, and then gets rid of them when they are no longer needed. Consider an example wherein Jini would negotiate

between a personal digital assistant and a printer to allow a document to be printed. Jini does not actually download the services themselves. Instead, it downloads an interface through which a device can interact with the services (Cravotta, 2000).

I/O drivers with some operating systems, such as Linux, are at the kernel-level. Kernel-level drivers can be decidedly useful because they offer direct access to hardware such as interrupts, I/O ports, and physical memory (Marsh, 2000). The Linux kernel-driver structure supports functions that let applications *read from* and *write to* devices, and perform other file operations such as *open* and *close*.

Depending on the version of Linux that is used, the operating system offers several schemes that accommodate kernel-space drivers and make their facilities available to application programs that need them. Linux can accommodate monolithic-driver and modular-driver code. A monolithic driver gets permanently attached to the kernel during kernel compilation (Marsh, 2000). If the monolithic-driver code is changed, the kernel must be recompiled, which can be a daunting task for programmers. An alternative to recompiling the kernel is to use modular drivers that application programs load and unload on demand by using a supervisory program (or daemon) called “kerneld.” When a process invokes kerneld, the daemon monitors messages that pass between calling applications and the kernel. The daemon alerts applications when the kernel tries to access drivers that are not available. If the application can direct kerneld to the missing drivers, execution of the process continues. If not, the kernel gives an error code to the application.

Linux developers place drivers in the `/dev` directory. The `/proc` directory is a directory that has information about the state of the running system. According to Marsh (2000), both `/dev` and `/proc` files use the same set of *major* and *minor* numbers to track and identify device drivers.

To implement a driver, a developer first loads driver code identified by a name and a major number into the kernel.

Devising software driver standards has been an important topic for discussion in the past couple of years (Masi, 1999). This is especially true for the Unix system, mainly because of its various versions. Vendors including Intel, Compaq, Hewlett-Packard, and Sun Microsystems have worked on a standard interface that will make it easier to write device drivers for Intel-based Unix servers (Scheier, 1998).

One way that standardization of device drivers can be considered beneficial would be by business travelers at airports. The logical extension of the Internet-bar concept is to have a facility, even a kiosk at an airport, where telecommunications connections can be made, and where documents can be printed out by business travelers. Service providers at airports may be wise to encourage the support of most or all devices, so that revenue streams can be maximized. A traveler may have created a customer profile sheet during a flight, and would want to be certain that her notebook computer could print out the document at the airport's printer.

There is still work to be done to help solve the overall connectivity problems that may exist among various devices. With networks, for instance, there needs to be a system in place for handling duplicate services, such as two different printers (e.g., either two different brands; or one color, and one black-and-white printer). The foundation for connectivity is coming into place. Numerous vendors are working hard to ensure its success, and it is in their best business interest to do so. It is not so much that connectivity issues should be solved by a universal connectivity standard, but rather, there should be a pragmatic way to bridge the various standards. What is continually important is that as devices become more technologically

advanced, it becomes increasingly important that device drivers are written with precision and with consummate reliability.

References

Bates, P. (2000, April 15). Control I/O ports from Windows NT. Test & Measurement World. [on-line].

Computer Language Company, Inc. Driver. (2000). [on-line].

Cravotta, N. (2000, Jan. 6). Device connectivity: a whole new set of secret handshakes. EDN. [on-line].

Edson, D. (1993). Writing Windows applications from start to finish (p. 14). New York: M & T Books.

Gaudin, S. (1999, June 7). Device drivers make Windows NT more stable. Computerworld. 79.

Langer, M. (2000). Mac OS9 (p. 274). Berkeley, CA: Peachpit Press.

Levenson, S. & Hertz, E. (1994). Now that I have OS/2 2.1 on my computer...what do I do next.....? (2nd Ed.) (p. 107). New York: Van Nostrand Reinhold.

Marsh, D. (2000, Oct. 15). The ins and outs of Linux kernel device drivers. Test & Measurement World, [on-line].

Masi, C. (April, 1999). Seeking the Holy Grail of device drivers. IEEE Spectrum. [on-line].

Newton's Telecom Dictionary. Device Driver. (2000) [on-line].

Norton, D. (1992). Writing Windows device drivers (p. 1). Reading, MA: Addison-Wesley Publishing Company.

Scheier, R. (1998, September 21). Push for common Unix drivers. Computerworld. 4.

Silberschatz, A. & Galvin, P. (1999). Operating systems concepts (5th Ed.) (pp. 30, 370 . 398, 408-410). New York: John Wiley & Sons.

Williams, R. & Walla, M. (2000). The Ultimate Windows2000 system administrator's guide. Reading, MA: Addison Wesley Longman.

Ziff-Davis Publishing Co. (1998, June 30). Windows Driver Model. PC Magazine. [online].