[**What is a Device Driver?**](http://www.tldp.org/LDP/khg/HyperNews/get/devices/whatis.html)

A device driver is a program that controls a particular type of device that is attached to your computer. There are device drivers for printers, displays, CD-ROM readers, diskette drives, and so on.A driver provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details of the hardware being used.

A driver typically 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 device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

**Purpose of a Device Driver**

The purpose of a device driver is to handle requests made by the kernel with regard to a particular type of device. There is a well-defined and consistent interface for the kernel to make these requests. By isolating device-specific code in device drivers and by having a consistent interface to the kernel, adding a new device is easier.

**Types of Device Drivers**

A device driver is a software module that resides within the kernel and is the software interface to a hardware device or devices. A hardware device is a peripheral, such as a disk controller, tape controller, or network controller device. In general, there is one device driver for each type of hardware device. Device drivers can be classified as:

* Block device drivers
* Character device drivers (including terminal drivers)
* Network device drivers

lOMoARcPSD|4537428

**• Pseudodevice drivers**

The following sections briefly discuss each type.

**Block Device Driver**

A block device driver is a driver that performs I/O by using file system block-sized buffers from a buffer cache supplied by the kernel. The kernel also provides for the device driver support interfaces that copy data between the buffer cache and the address space of a process.

Block device drivers are particularly well-suited for disk drives, the most common block devices. For block devices, all I/O occurs through the buffer cache.

**Character Device Driver**

A character device driver does not handle I/O through the buffer cache, so it is not tied to a single approach for handling I/O. You can use a character device driver for a device such as a line printer that handles one character at a time. However, character drivers are not limited to performing I/O one character at a time (despite the name ``character'' driver). For example, tape drivers frequently perform

I/O in 10K chunks. You can also use a character device driver when it is necessary to copy data directly to or from a user process.

Because of their flexibility in handling I/O, many drivers are character drivers. Line printers, interactive terminals, and graphics displays are examples of devices that require character device drivers.

A terminal device driver is actually a character device driver that handles I/O character processing for a variety of terminal devices. Like any character device, a terminal device can accept or supply a stream of data based on a request from a user process. It cannot be mounted as a file system and, therefore, does not use data caching.

**Network Device Driver**

A network device driver attaches a network subsystem to a network interface, prepares the network interface for operation, and governs the transmission and4537428reception of network frames over the network interface. This book does not discuss network device drivers.

**Pseudo device Driver**

Not all device drivers control physical hardware. Such device drivers are called ``pseudo device'' drivers. Like block and character device drivers, pseudo device drivers make use of the device driver interfaces. Unlike block and character device drivers, pseudo device drivers do not operate on a bus. One example of a pseudo device driver is the pseudo terminal or pty terminal driver, which simulates a terminal device. The pty terminal driver is a character device driver typically used for remote logins.

**Single Binary Module**

Digital UNIX provides the tools and techniques for you to produce a single binary module. A single binary module is the executable image of a device driver that can be statically or dynamically configured into the kernel. A single binary module has a file extension of .mod. The .mod file for the current version of Digital UNIX is not the same as the .mod file used in previous versions of the operating system. To produce a single binary module, there is code you need to implement in the driver's configure interface. Chapter 6 describes how to write a configure interface so that your device driver can be statically or dynamically configured into the kernel. In addition, there are steps you follow when using the system management tools for statically and dynamically configuring the driver (the single binary module) into the kernel.

oARcPSD|4537428

**When a Device Driver Is Called**

**• Autoconfiguration**

The kernel calls a device driver (specifically, the driver's probe interface) at autoconfiguration time to determine what devices are available and to initialize them.

**• I/O operations**

The kernel calls a device driver to perform I/O operations on the device. These operations include opening the device to perform reads and writes and closing the device.

**• Interrupt handling**

The kernel calls a device driver to handle interrupts from devices capable of generating them.

**• Special requests**

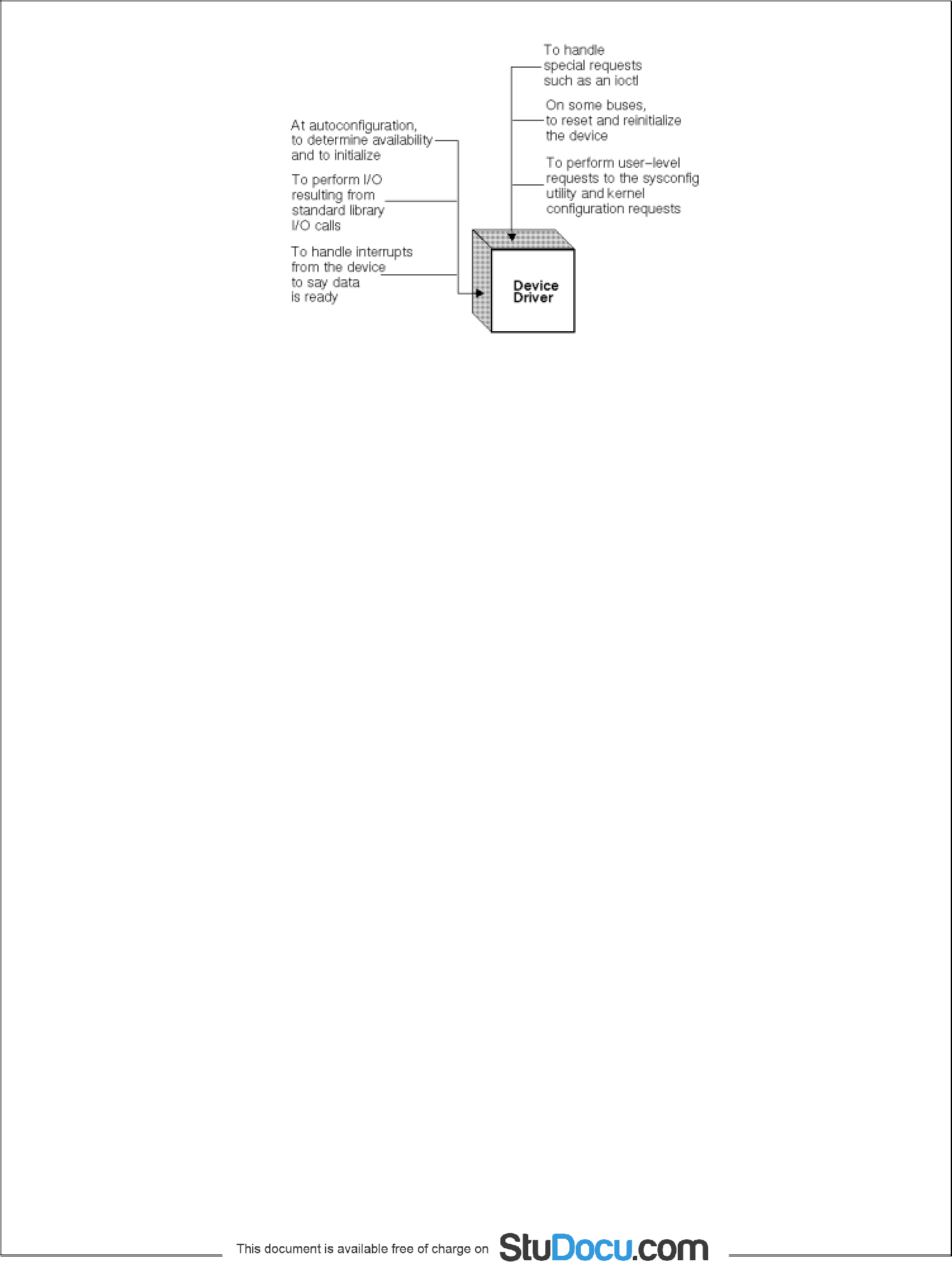
The kernel calls a device driver to handle special requests through ioctl calls.

* **Reiniti**alization

The kernel calls a device driver to reinitialize the driver, the device, or both when the bus (the path from the CPU to the device) is reset.

* **User**-level requests to the sysconfig utility

The kernel calls a device driver (specifically, the driver's configure interface) to handle requests that result from use of the sysconfig utility. The sysconfig utility allows a system manager to dynamically configure, unconfigure, query, and reconfigure a device. These requests cause the kernel to call the device driver's configure interface. In addition, the driver's configure interface performs one-time initializations when called by the boot software or by the sysconfig utility.



**Figure 1-1: When the Kernel Calls a Device Driver**

Some of these requests, such as input or output, result directly or indirectly from corresponding system calls in a user program. Other requests, such as the calls at autoconfiguration time, do not result from system calls but from activities that occur at boot time.

**Device Driver Configuration**

Device driver configuration consists of the tasks necessary to incorporate device drivers into the kernel to make them available to system management and other utilities. After you write your device driver you need to create a single binary module (a file with a .mod extension) from the driver source file (a file with a .c extension). After you create the single binary module, you need to configure it into the kernel so that you can test it on a running system. There are two methods of device driver configuration: static configuration and dynamic configuration. Static configuration consists of the tasks and tools necessary to link a device driver (single binary module) directly into the kernel at kernel build time. Dynamic configuration consists of the tasks and tools necessary to link a device

driver (single binary module) directly into the kernel at any point in time. Do not confuse device driver configuration (static configuration and dynamic configuration), which encompasses the tools and steps for configuring the driver into the kernel, with autoconfiguration and configuration. Autoconfiguration is a process that determines what hardware actually exists during the current instance of the running kernel at static configuration time. The autoconfiguration software (specifically, the bus's confl1 interface) calls the driver's probe, attach, and slave interfaces. Thus, the driver's probe, attach, and slave interfaces cooperate with the bus's confl1 interface to determine if devices exist and are functional on a given system.

Configuration is a process associated with handling user-level requests to the sysconfig utility to dynamically configure, unconfigure, query, and reconfigure devices. The cfgmgr framework calls the driver's configure interface as a result of these sysconfig utility requests. The cfgmgr framework also calls the driver's configure interface as a result of static configuration requests. Thus, the driver's configure interface cooperates with the cfgmgr framework to statically configure and to dynamically configure, unconfigure, query, and reconfigure devices. The driver's configure interface also cooperates with the cfgmgr framework to perform one-time initialization tasks such as allocating memory, initializing data structures and variables, and adding driver entry points to the dsent table. A driver's configure interface should be implemented to handle static and dynamic configuration.

**Applications**

Because of the diversity of modern hardware and operating systems, drivers operate in many different environments.Drivers may [interface](http://en.wikipedia.org/wiki/Interface_(computer_science)) with:

* [printers](http://en.wikipedia.org/wiki/Computer_printer)
* [video adapters](http://en.wikipedia.org/wiki/Video_card)
* [Network cards](http://en.wikipedia.org/wiki/Network_interface_controller)
* [Sound cards](http://en.wikipedia.org/wiki/Sound_card)
* Local [buses](http://en.wikipedia.org/wiki/Computer_bus) of various sorts**—**in particular, for [bus mastering](http://en.wikipedia.org/wiki/Bus_mastering) on modern systems
* Low-[bandwidth](http://en.wikipedia.org/wiki/Bandwidth_(computing)) [I/O](http://en.wikipedia.org/wiki/Input/output) buses of various sorts (for [pointing devices](http://en.wikipedia.org/wiki/Pointing_device) such as [mice,](http://en.wikipedia.org/wiki/Computer_mouse) [keyboards,](http://en.wikipedia.org/wiki/Computer_keyboard) [USB,](http://en.wikipedia.org/wiki/Universal_Serial_Bus) etc.)
* [Computer storage](http://en.wikipedia.org/wiki/Computer_storage) devices such as [hard disk,](http://en.wikipedia.org/wiki/Hard_disk_drive) [CD-ROM,](http://en.wikipedia.org/wiki/CD-ROM) and [floppy disk](http://en.wikipedia.org/wiki/Floppy_disk) buses [(ATA,](http://en.wikipedia.org/wiki/Advanced_Technology_Attachment) [SATA,](http://en.wikipedia.org/wiki/Serial_ATA) [SCSI)](http://en.wikipedia.org/wiki/SCSI)
* Implementing support for different [file systems](http://en.wikipedia.org/wiki/File_system)
* [Image scanners](http://en.wikipedia.org/wiki/Image_scanner)

lOMoARcPSD|

* [Digital cameras](http://en.wikipedia.org/wiki/Digital_camera)

Common levels of abstraction for device drivers include:

* For hardware:
  + Interfacing directly
  + Writing to or reading from a [device control register](http://en.wikipedia.org/wiki/Device_control_register)
  + Using some higher-level interface (e.g. [Video BIOS)](http://en.wikipedia.org/wiki/Video_BIOS)
  + Using another lower-level device driver (e.g. file system drivers using disk drivers)
  + Simulating work with hardware, while doing something entirely different
* For software:
  + Allowing the operating system direct access to hardware resources
  + Implementing only [primitives](http://en.wikipedia.org/wiki/Primitive_(computer_science))
  + Implementing an interface for non-driver software (e.g., [TWAIN)](http://en.wikipedia.org/wiki/TWAIN)
  + Implementing a language, sometimes quite high-level (e.g., [PostScript)](http://en.wikipedia.org/wiki/PostScript)

So choosing and installing the correct device drivers for given hardware is often a key component of computer system configuration.

**Place of a Device Driver in System**

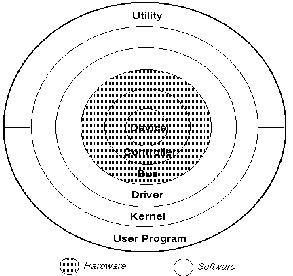
* User program or utility

A user program, or utility, makes calls on the kernel but never directly calls a device driver.

* Kernel

The kernel runs in supervisor mode and does not communicate with a device except through calls to a device driver.

* Device driver

A device driver communicates with a device by reading and writing through a bus to peripheral device registers.

* Bus

The bus is the data path between the main processor and the device controller.

**Figure 1-2: Place of a Device Driver**

* Controller
* A controller is a physical interface for controlling one or more devices. A controller connects to a bus.
* Peripheral device
* A peripheral device is a device that can be connected to a controller, for example, a disk or tape drive. Other devices (for example, the network) maybe integral to the controller.

The following sections describe these parts, with an emphasis on how a device driver relates to them.

**User Program or Utility**

User programs, or utilities, make system calls on the kernel that result in the kernel making requests of a device driver. For example, a user program can make a read system call, which calls the driver's read interface.

**Kernel**

The kernel makes requests to a device driver to perform operations on a particular device. Some of these requests result directly from user program requests. For example:

* Block I/O (open, strategy, close)
* Character I/O (open, write, close)

Autoconfiguration requests, such as probe and attach, do not result directly from a user program, but result from activities performed by the kernel. At boot time, for example, the kernel (specifically, the bus code) calls the driver's probe interface. Configuration requests, such as configure, unconfigure, and query, result from a system manager's use of the sysconfig utility.

A device driver may call on kernel support interfaces to support such tasks as:

* Sleeping and waking (process rescheduling)
* Scheduling events
* Managing the buffer cache
* Moving or initializing data

**Device Driver**

A device driver, run as part of the kernel software, manages each of the device controllers on the system. Often, one device driver manages an entire set of identical device controller interfaces.

With Digital UNIX, you can statically configure more device drivers into the kernel than there are physical devices in the hardware system. At boot time, the autoconfiguration software determines which of the physical devices are accessible and functional and can produce a correct run-time configuration for that instance of the running kernel. Similarly, when a driver is dynamically configured, the kernel performs the configuration sequence for each instance of the physical device.

The kernel makes requests of a driver by calling the driver's standard entry points (such as the probe, attach, open, read, write, close entry points). In the case of I/O requests such as readand write, it is typical that the device causes an interrupt upon completion of each I/O operation. Thus, a write system call from a user program may result in several calls on the interrupt entry point in addition to the original call on the write entry point. This is the case when the write request is segmented into several partial transfers at the driver level.

Device drivers, in turn, make calls upon kernel support interfaces to perform the tasks mentioned earlier.The device register offset definitions giving the layout of the control registers for a device are part of the source for a device driver. Device drivers, unlike the rest of the kernel, can access and modify these registers. Digital UNIX provides generic CSR I/O access kernel interfaces that allow device drivers to read from and write to these registers.

**Bus**

When a device driver reads or writes to the hardware registers of a controller, the data travels across a bus.A bus is a physical communication path and an access protocol between a processor and its peripherals. A bus standard, with a predefined set of logic signals, timings, and connectors, provides a means by which many types of device interfaces (controllers) can be built and easily combined within a computer system. The term *OPENBUS* refers to those buses whose architectures and interfaces are publiclydocumented, allowing a vendor to easily plug in hardware and software components. The TURBOchannel bus, the EISA bus, the PCI bus, and the VMEbus, for example, can be classified as having OPENbus architectures.

Device driver writers must understand the bus that the device is connected to.

lMoARcPSD|4537428

**Device Controller**

A device controller is the hardware interface between the computer and a peripheral device. Sometimes a controller handles several devices. In other cases, a controller is integral to the device.

**Peripheral Device**

A peripheral device is hardware, such as a disk controller, that connects to a computer system. It can be controlled by commands from the computer and can send data to the computer and receive data from it.Examples of peripheral devices include:

* A data acquisition device, like a digitizer
* A line printer
* A disk or tape drive

**Kernel mode vs. user mode**

Device drivers, particularly on modern [Microsoft Windows](http://en.wikipedia.org/wiki/Microsoft_Windows) platforms, can run in [kernel-mode](http://en.wikipedia.org/wiki/CPU_modes) [(Ring 0 on x86 CPUs)](http://en.wikipedia.org/wiki/Ring_(computer_security)) or in [user-mode](http://en.wikipedia.org/wiki/User_space) (Ring 3 on x86 CPUs). The primary benefit of running a driver in user mode is improved stability, since a poorly written user mode device driver cannot crash the system by overwriting kernel memory On the other hand, user/kernel-mode transitions usually impose a considerable performance overhead, thereby prohibiting user-mode drivers for low latency and high throughput requirements.

Kernel space can be accessed by user module only through the use of system calls. End user programs like the UNIX shell or other GUI-based applications are part of the user space. These applications interact with hardware through kernel supported functions.

**Device drivers in Windows, MAC OS & linux.**

[Microsoft](http://en.wikipedia.org/wiki/Microsoft) has attempted to reduce system instability due to poorly written device drivers by creating a new framework for driver development, called [Windows Driver](http://en.wikipedia.org/wiki/Windows_Driver_Foundation) [Foundation](http://en.wikipedia.org/wiki/Windows_Driver_Foundation) (WDF). This includes [User-Mode Driver Framework](http://en.wikipedia.org/wiki/User-Mode_Driver_Framework) (UMDF) that28 encourages development of certain types of drivers**—**primarily those that implement [message-based protocol](http://en.wikipedia.org/wiki/Message-based_protocol) for communicating with their devices**—**as user-mode drivers. If such drivers malfunction, they do not cause system instability. The [Kernel-Mode Driver Framework](http://en.wikipedia.org/wiki/Kernel-Mode_Driver_Framework) (KMDF) model continues to allow development of kernel-mode device drivers, but attempts to provide standard implementations of functions that are known to cause problems, including cancellation of I/O operations, power management, and plug and play device support.

[Apple](http://en.wikipedia.org/wiki/Apple_Inc.) has an open-source framework for developing drivers on [Mac OS X](http://en.wikipedia.org/wiki/Mac_OS_X) called the [I/O Kit.](http://en.wikipedia.org/wiki/I/O_Kit)

In [Linux](http://en.wikipedia.org/wiki/Linux_kernel) environments, programmers can build device drivers as parts of the [kernel,](http://en.wikipedia.org/wiki/Linux_kernel) separately as loadable [modules,](http://en.wikipedia.org/wiki/Loadable_kernel_module) or as user-mode drivers (for certain types of devices where kernel interfaces exist, such as for USB devices). [Makedev](http://en.wikipedia.org/wiki/Makedev) includes a list of the devices in Linux: ttyS (terminal), lp [(parallel port),](http://en.wikipedia.org/wiki/Parallel_port) hd (disk), loop, sound (these include [mixer,sequencer,](http://en.wikipedia.org/wiki/Music_sequencer) [dsp,](http://en.wikipedia.org/wiki/Digital_signal_processor) and audio).

The [Microsoft Windows](http://en.wikipedia.org/wiki/Microsoft_Windows) [.sys](http://en.wikipedia.org/wiki/.sys) files and [Linux](http://en.wikipedia.org/wiki/Linux) .ko modules contain loadable device drivers. The advantage of loadable device drivers is that they can be loaded only when necessary and then unloaded, thus saving kernel memory.