Writing Device Drivers: Tutorial

Product Version: Digital UNIX Version 4.0 or higher
March 1996

This guide contains information that systems engineers need to write device drivers for hardware that runs the Digital UNIX operating system. Included is information on driver concepts, device driver interfaces, kernel interfaces that device drivers use, kernel data structures, configuration of device drivers, and header files related to device drivers.

Estratto dal primo capitolo del testo consultabile in forma completa all'indirizzo
http://www.cs.arizona.edu/computer.help/policy/DIGITAL_unix/
AA-PUBVD-TE_html/TITLE.html

1 Introduction to Device Drivers

This chapter presents an overview of device drivers by discussing:

- The purpose of a device driver
- The types of device drivers
- Single binary module
- When a device driver is called
- Device driver configuration
- The place of a device driver in the Digital UNIX operating system

The chapter concludes with an example of how a device driver in Digital UNIX reads a single character.

1.1 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.

1.2 Types of Device Drivers

A device driver is a software module that resides within the Digital UNIX 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
- Pseudodevice drivers

The following sections briefly discuss each type.

1.2.1 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.

1.2.2 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.

1.2.3 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 and reception of network frames over the network interface. This book does not discuss network device drivers.

1.2.4 Pseudodevice Driver

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

1.3 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. Chapter 14 explains how to statically and dynamically configure drivers into the kernel.

1.4 When a Device Driver Is Called

Figure 1-1 shows that the kernel calls a device driver during:

- 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.

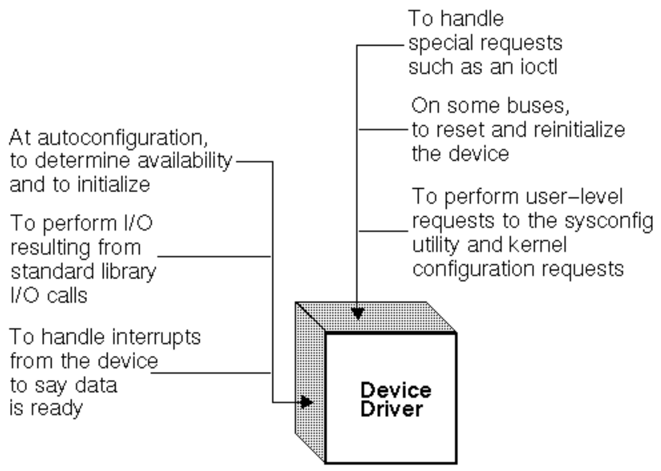
- Reinitialization

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


To handle special requests such as an ioctl
On some buses, to reset and reinitialize the device
To perform user–level requests to the sysconfig utility and kernel configuration requests
At autoconfiguration, to determine availability and to initialize
To perform I/O resulting from standard library I/O calls
To handle interrupts from the device to say data is ready
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.

1.5 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. Chapter 14 describes how to create a single binary module and then how to statically and dynamically configure the single binary module (the device driver) into the kernel.

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.

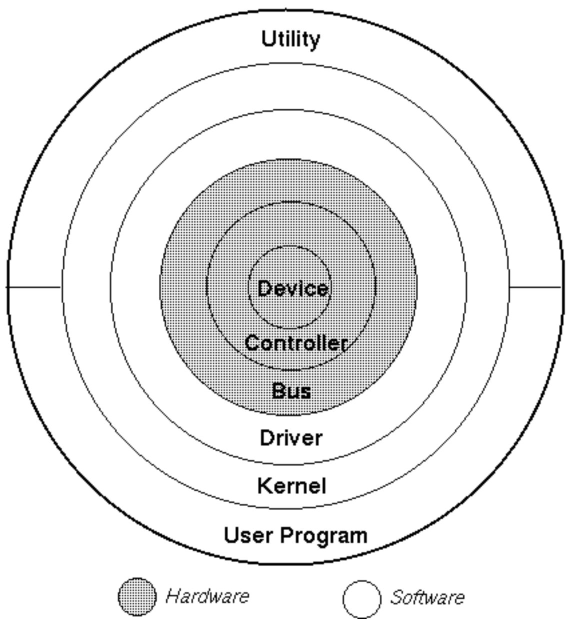
1.6 Place of a Device Driver in Digital UNIX

Figure 1-2 shows the place of a device driver in Digital UNIX relative to the device:

- User program or utility

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

Figure 1-2: Place of a Device Driver in Digital UNIX



- 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.

- 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) may be integral to the controller.
The following sections describe these parts, with an emphasis on how a device driver relates to them.

1.6.1 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.

1.6.2 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

1.6.3 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.

As stated previously, 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 `read` and `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.

1.6.4 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 publicly documented, 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. This book covers topics that all driver writers need to know regardless of the bus.

1.6.5 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.

1.6.6 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

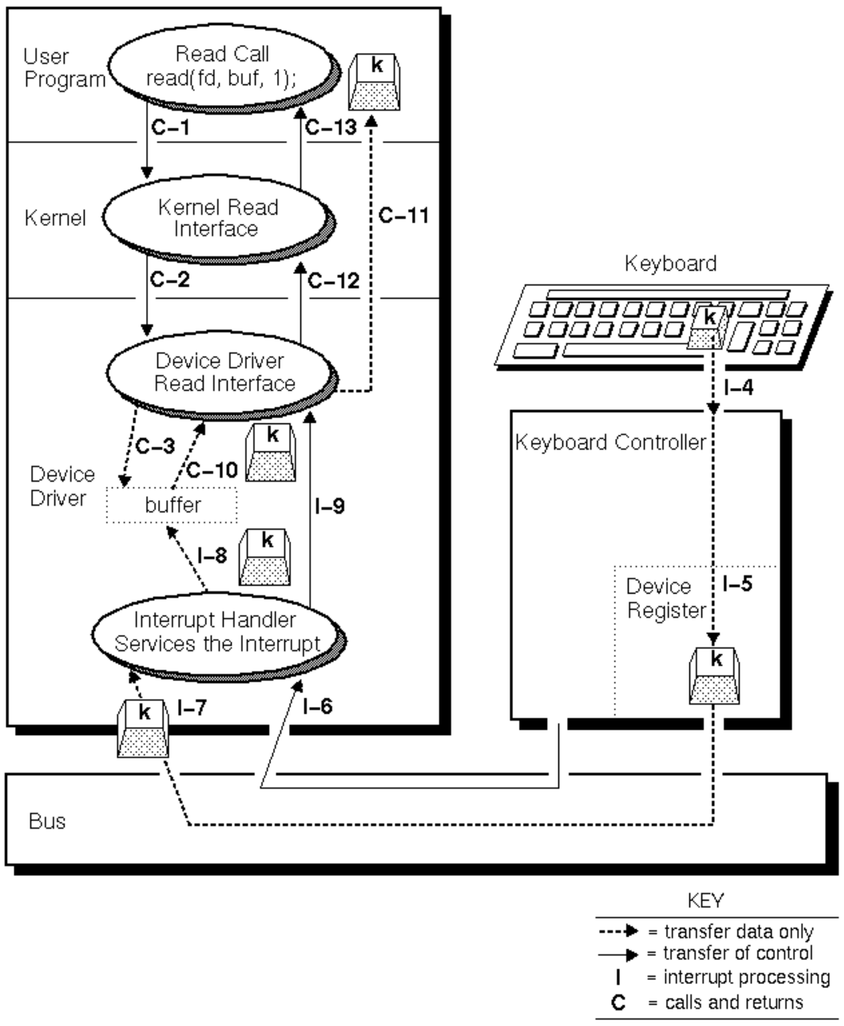
1.7 Example of Reading a Character

This section provides an example of how Digital UNIX processes a read request of a single character in raw mode from a terminal. (Raw mode returns single characters.) Although the example takes a simplified view of character processing, it does show how control can pass from a user program to the kernel to the device driver. It also shows that interrupt processing occurs asynchronously from other device driver activity.
Figure 1-3 summarizes the flow of control between a user program, the kernel, the device driver, and the hardware. The figure shows the following sequence of events:

- A read request is made to the device driver (C-1 to C-3).
- The character is captured by the hardware (I-4 and I-5).
- The interrupt is generated (I-6).
- The interrupt handler services the interrupt (I-7 to I-9).
- The character is returned (C-10 to C-13).

Figure 1-3 provides a snapshot of the processing that occurs in the reading of a single character. The following sections elaborate on this sequence.

Figure 1-3: Simple Character Driver Interrupt Example



1.7.1 A Read Request Is Made to the Device Driver

A user program issues a `read` system call (C-1). The figure shows that the `read` system call passes three arguments: a file descriptor (the `fd` argument), the character pointer to where the information is stored (the `buf` argument), and an integer (the value 1) that tells the driver's `read` interface how many bytes to read. The calling sequence is blocked inside the device driver's `read` interface because the buffer where the data is stored is empty, indicating that there are currently no characters available to satisfy the read. The kernel's `read` interface makes a request of the device driver's `read` interface to perform a read of the character based on the arguments passed by the `read` system call (C-2). Essentially, the driver `read` interface is waiting for a character to be typed at the terminal's keyboard. The currently blocked process that caused the kernel to call the driver's `read` interface is not running in the CPU (C-3).

1.7.2 The Character Is Captured by the Hardware

Later, a user types the letter k on the terminal keyboard (I-4). The letter is stored in the device's data register (I-5).

1.7.3 The Interrupt Is Generated

When the user types a key, the console keyboard controller alters some signals on the bus. This action notifies the CPU that something has changed inside the console keyboard controller. This condition causes the CPU to immediately start running the console keyboard controller's interrupt handler (I-6). The state of the interrupted process (either some other process or the idle loop) is saved so that the process can be returned to its original state as though it had never been interrupted in the first place.

1.7.4 The Interrupt Handler Services the Interrupt

The console device driver's interrupt handler first checks the state of the driver and notices that a pending read operation exists for the original process. The console device driver manipulates the controller hardware by way of the bus hardware in order to obtain the value of the character that was typed. This character value was stored somewhere inside the console controller's hardware (I-7). In this case, the value 107 (the ASCII representation for the k character) is stored. The interrupt handler stores this character value into a buffer that is in a location known to the rest of the console driver interfaces (I-8). It then awakens the original, currently sleeping, process so that it is ready to run again (I-9). The interrupt handler returns, in effect restoring the interrupted process (not the original process yet) so that it may continue where it left off.

1.7.5 The Character Is Returned

Later, the kernel's process scheduler notices that the original process is ready to run, and so allows it to run. After the original process resumes running (after the location where it was first blocked), it knows which buffer to look at to obtain the typed character (C-10). It removes the character from this buffer and puts it into the user's address space (C-11). The device driver's `read` interface returns control to the kernel's `read` interface (C-12). The kernel `read` interface returns control to the user program that previously initiated the read request (C-13).

1.7.6 Summary of the Example

Although this example presents a somewhat simplified view of character processing, it does illustrate how control passes from a user program to the kernel to the device driver. It also shows clearly that interrupt processing occurs asynchronously from other device driver activity.