A
PROJECT REPORT ON

USB DEVICE DRIVER

DIPLOMA IN EMBEDDED SYSTEM DESIGN (PG-DESD)

BY

KOMAL PHADTARE

ROSHAN KORPE

VEDANT JADHAV

NIKHIL KADU

AT

SUNBEAM INSTITUTE OF INFORMATION TECHNOLOGY, PUNE

SUNBEAM INSTITUTE OF INFORMATION TECHNOLOGY, PUNE.

CERTIFICATE

This is to certify that the project

USB DEVICE DRIVER

Has been submitted by

KOMAL PHADTARE

ROSHAN KORPE

VEDANT JADHAV

NIKHIL KADU

In partial fulfillment of the requirement for the Course of **PG Diploma in Embedded System Design (PG-DESD MARCH 2022)** as prescribed by the **CDAC** ACTS, PUNE.

Place: Pune

Date: -12 SEP 2022

Prof. Sohail Inamdar
Project Guide

ACKNOWLEDGEMENT

It is indeed a matter of great pleasure and proud privilege to be able to present this project on "**USB DEVICE DRIVER**". The completion of this project work is great experience in student's life and its execution is inevitable in hands of guide. We are highly indebted to the Project Guide **Mr. Sohail Inamdar** for his valuable guidance and appreciation for giving form and substance to this project. It is due to his enduring efforts, patience and enthusiasm which has given sense of direction and purposefulness to this project and alternately made it a success.

Amongst the panorama of other people who supported us in this project, we are highly indebted to our technical staff of Sunbeam. It is their support, encouragement and guidance which motivated us to go ahead and make the project a successful venture.

Lastly, we would like to express our gratitude to all other helping hands which have been directly or indirectly associated with our project.

Date: 12/09/2022

ABSTRACT

Device drivers take on a special role in the Linux kernel. They are distinct “black boxes” that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver. This programming interface is such that drivers can be built separately from the rest of the kernel and “plugged in” at runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available.

Device drivers are software interfaces that allow a computer’s central processing unit (CPU) to communicate with peripherals. It make a particular piece of hardware respond to a well-defined internal programming interface. The main strength of device is that they enable programmers to write software that will run on a computer regardless of the type of devices that are connected to that computer. The purpose of this work to develop a linux kernel device driver is to communicate with the STM32F407G-DISC1 device. To achieve the serial communication of STM32(Device) through the USB instead of using another communication protocols.

TABLE OF INDEX

CHAPTER 1. Introduction To Device Driver

One of the many advantages of free operating systems, as typified by Linux, is that their internals are open for all to view. The operating system, once a dark and mysterious area whose code was restricted to a small number of programmers, can now be readily examined, understood, and modified by anybody with the requisite skills. Linux has helped to democratize operating systems. The Linux kernel remains a large and complex body of code, however, and would-be kernel hackers need an entry point where they can approach the code without being overwhelmed by complexity. Often, device drivers provide that gateway.

Device drivers take on a special role in the Linux kernel. They are distinct “black boxes” that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver. This programming interface is such that drivers can be built separately from the rest of the kernel and “plugged in” at runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available.

There are a number of reasons to be interested in the writing of Linux device drivers. The rate at which new hardware becomes available (and obsolete!) alone guarantees that driver writers will be busy for the foreseeable future. Individuals may need to know about drivers in order to gain access to a particular device that is of interest to them. Hardware vendors, by making a Linux driver available for their products, can add the large and growing Linux user base to their potential markets. And the open source nature of the Linux system means that if the driver writer wishes, the source to a driver can be quickly disseminated to millions of users. This book teaches you how to write your own drivers and how to hack around in related parts of the kernel. We have taken a device-independent approach; the programming techniques and interfaces are presented, whenever possible, without being tied to any specific device. Each driver is different; as a driver writer, you need to understand your specific device well. But most of the principles and basic techniques are the same for all drivers.

This book cannot teach you about your device, but it gives you a handle on the background you need to make your device work. As you learn to write drivers, you find out a lot about the Linux kernel in general; this may help you understand how your machine works

and why things aren't always as fast as you expect or don't do quite what you want. We introduce new ideas gradually, starting off with very simple drivers and building on them; every new concept is accompanied by sample code that doesn't need special hardware to be tested. This chapter doesn't actually get into writing code. However, we introduce some background concepts about the Linux kernel that you'll be glad you know later, when we do launch into programming.

1.1 The Role of the Device Driver

As a programmer, you are able to make your own choices about your driver, and choose an acceptable trade-off between the programming time required and the flexibility of the result. Though it may appear strange to say that a driver is "flexible," we like this word because it emphasizes that the role of a device driver is providing mechanism, not policy.

The distinction between mechanism and policy is one of the best ideas behind the Unix design. Most programming problems can indeed be split into two parts: "what capabilities are to be provided" (the mechanism) and "how those capabilities can be used" (the policy). If the two issues are addressed by different parts of the program, or even by different programs altogether, the software package is much easier to develop and to adapt to particular needs.

For example, Unix management of the graphic display is split between the X server, which knows the hardware and offers a unified interface to user programs, and the window and session managers, which implement a particular policy without knowing anything about the hardware. People can use the same window manager on different hardware, and different users can run different configurations on the same workstation. Even completely different desktop environments, such as KDE and GNOME, can coexist on the same system. Another example is the layered structure of TCP/IP networking: the operating system offers the socket abstraction, which implements no policy regarding the data to be transferred, while different servers are in charge of the services (and their associated policies). Moreover, a server like ftpd provides the file transfer mechanism, while users can use whatever client they prefer; both command-line and graphic clients exist, and anyone can write a new user interface to transfer files. Where drivers are concerned, the same separation of mechanism and policy applies. The floppy driver is policy free — its role is only to show the diskette as a continuous array of data blocks. Higher levels of the system provide policies, such as who may access the floppy drive, whether the drive is accessed directly or via a filesystem, and whether users may mount filesystems on the drive. Since different environments usually need to use hardware in different ways, it's important to be as policy free as possible.

When writing drivers, a programmer should pay particular attention to this fundamental concept: write kernel code to access the hardware, but don't force particular policies on the user, since different users have different needs. The driver should deal with making the hardware available, leaving all the issues about how to use the hardware to the applications. A driver, then, is flexible if it offers access to the hardware capabilities without adding constraints. Sometimes, however, some policy decisions must be made. For example, a digital I/O driver may only offer byte-wide access to the hardware in order to avoid the extra code needed to handle individual bits.

You can also look at your driver from a different perspective: it is a software layer that lies between the applications and the actual device. This privileged role of the driver allows the driver programmer to choose exactly how the device should appear: different drivers can offer different capabilities, even for the same device. The actual driver design should be a balance between many different considerations. For instance, a single device may be used concurrently by different programs, and the driver programmer has complete freedom to determine how to handle concurrency. You could implement memory mapping on the device independently of its hardware capabilities, or you could provide a user library to help application programmers implement new policies on top of the available primitives, and so forth. One major consideration is the trade-off between the desire to present the user with as many options as possible and the time you have to write the driver, as well as the need to keep things simple so that errors don't creep in.

Policy-free drivers have a number of typical characteristics. These include support for both synchronous and asynchronous operation, the ability to be opened multiple times, the ability to exploit the full capabilities of the hardware, and the lack of software layers to "simplify things" or provide policy-related operations. Drivers of this sort not only work better for their end users, but also turn out to be easier to write and maintain as well. Being policyfree is actually a common target for software designers.

Many device drivers, indeed, are released together with user programs to help with configuration and access to the target device. Those programs can range from simple utilities to complete graphical applications. Examples include the tunelp program, which adjusts how the parallel port printer driver operates, and the graphical cardctl utility that is part of the PCMCIA driver package. Often a client library is provided as well, which provides capabilities that do not need to be implemented as part of the driver itself.

The scope of this book is the kernel, so we try not to deal with policy issues or with application programs or support libraries. Sometimes we talk about different policies and how to support them, but we won't go into much detail about programs using the device or the policies they enforce. You should understand, however, that user programs are an integral part of a software package and that even policy-free packages are distributed with configuration files that apply a default behaviour to the underlying mechanisms.

1.2 Splitting the Kernel

In a Unix system, several concurrent processes attend to different tasks. Each process asks for system resources, be it computing power, memory, network connectivity, or some other resource. The kernel is the big chunk of executable code in charge of handling all such requests. Although the distinction between the different kernel tasks isn't always clearly marked, the kernel's role can be split (as shown in Figure 1-1) into the following parts:

Process management :

The kernel is in charge of creating and destroying processes and handling their connection to the outside world (input and output). Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality and is also handled by the kernel. In addition, the scheduler, which controls how processes share the CPU, is part of process management. More generally, the kernel's process management activity implements the abstraction of several processes on top of a single CPU or a few of them.

Memory management:

The computer's memory is a major resource, and the policy used to deal with it is a critical one for system performance. The kernel builds up a virtual addressing space for any and all processes on top of the limited available resources. The different parts of the kernel interact with the memory-management subsystem through a set of function calls, ranging from the simple malloc/free pair to much more complex functionalities.

Filesystems:

Unix is heavily based on the filesystem concept; almost everything in Unix can be treated as a file. The kernel builds a structured filesystem on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system. In addition, Linux supports multiple filesystem types, that is, different ways of organizing data on the physical medium. For example, disks may be formatted with the Linux-standard ext3 filesystem, the commonly used FAT filesystem or several others.

Device control:

Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a device driver. The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape drive. This aspect of the kernel's functions is our primary interest in this book.

Networking:

Networking must be managed by the operating system, because most network operations are not specific to a process: incoming packets are asynchronous events. The packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity. Additionally, all the routing and address resolution issues are implemented within the kernel.

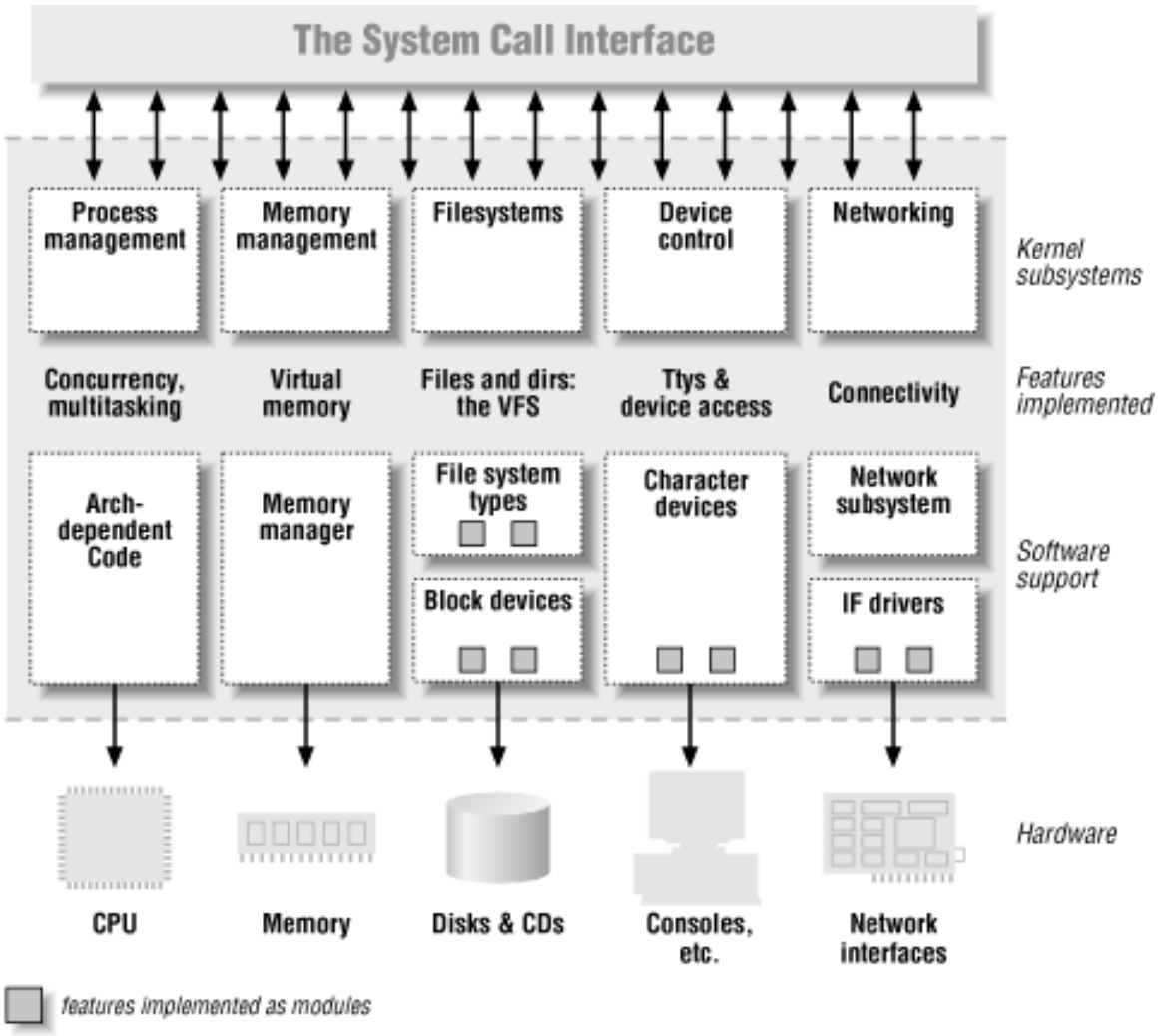


Fig1-1. A split view of the kernel.

1.3 Classes of Devices and Modules

The Linux way of looking at devices distinguishes between three fundamental device types. Each module usually implements one of these types, and thus is classifiable as a char module, a block module, or a network module. This division of modules into different types, or classes, is not a rigid one; the programmer can choose to build huge modules implementing different drivers in a single chunk of code. Good programmers, nonetheless, usually create a different module for each new functionality they implement, because decomposition is a key element of scalability and extendability.

The three classes are:

Character devices

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. The text console (/dev/console) and the serial ports (/dev/ttyS0 and friends) are examples of char devices, as they are well represented by the stream abstraction. Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially. There exist, nonetheless, char devices that look like data areas, and you can move back and forth in them; for instance, this usually applies to frame grabbers, where the applications can access the whole acquired image using mmap or lseek.

Block devices

Like char devices, block devices are accessed by filesystem nodes in the /dev directory. A block device is a device (e.g., a disk) that can host a filesystem. In most Unix systems, a block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length. Linux, instead, allows the application to read and write a block device like a char device — it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface. Like a char device, each block device is accessed through a filesystem node, and the difference between them is transparent to the user. Block drivers have a completely different interface to the kernel than char drivers.

Network interfaces

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a pure software device, like the loopback interface. A network interface is in charge of

sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Many network connections (especially those using TCP) are stream-oriented, but network devices are, usually, designed around the transmission and receipt of packets. A network driver knows nothing about individual connections; it only handles packets.

CHAPTER 2. CHAR DRIVERS

The goal of this chapter is to write a complete char device driver. We develop a character driver because this class is suitable for most simple hardware devices. Char drivers are also easier to understand than block drivers or network drivers (which we get to in later chapters). Our ultimate aim is to write a modularized char driver, but we won't talk about modularization issues in this chapter.

Throughout the chapter, we present code fragments extracted from a real device driver: scull (Simple Character Utility for Loading Localities). scull is a char driver that acts on a memory area as though it were a device. In this chapter, because of that peculiarity of scull, we use the word device interchangeably with "the memory area used by scull."

The advantage of scull is that it isn't hardware dependent. scull just acts on some memory, allocated from the kernel. Anyone can compile and run scull, and scull is portable across the computer architectures on which Linux runs. On the other hand, the device doesn't do anything "useful" other than demonstrate the interface between the kernel and char drivers and allow the user to run some tests.

2.1 Major and Minor Numbers

Char devices are accessed through names in the filesystem. Those names are called special files or device files or simply nodes of the filesystem tree; they are conventionally located in the /dev directory. Special files for char drivers are identified by a "c" in the first column of the output of ls -l. Block devices appear in /dev as well, but they are identified by a "b." The focus of this chapter is on char devices, but much of the following information applies to block devices as well. If you issue the ls -l command, you'll see two numbers (separated by a comma) in the device file entries before the date of the last modification, where the file length normally appears. These numbers are the major and minor device number for the particular device. The following listing shows a few devices as they appear on a typical system. Their major numbers are 1, 4, 7, and 10, while the minors are 1, 3, 5, 64, 65, and 129.

crw-rw-rw- 1 root root 1, 3 Apr 11 2002 null

crw------- 1 root root 10, 1 Apr 11 2002 psaux

crw------- 1 root root 4, 1 Oct 28 03:04 tty1

crw-rw-rw- 1 root tty 4, 64 Apr 11 2002 ttys0

crw-rw---- 1 root uucp 4, 65 Apr 11 2002 ttyS1

crw--w---- 1 vcsa tty 7, 1 Apr 11 2002 vcs1

crw--w---- 1 vcsa tty 7, 129 Apr 11 2002 vcsa1

crw-rw-rw- 1 root root 1, 5 Apr 11 2002 zero

Traditionally, the major number identifies the driver associated with the device. For example, /dev/null and /dev/zero are both managed by driver 1, whereas virtual consoles and serial terminals are managed by driver 4; similarly, both vcs1 and vcsa1 devices are managed by driver 7. Modern Linux kernels allow multiple drivers to share major numbers, but most devices that you will see are still organized on the one-major-one-driver principle. The minor number is used by the kernel to determine exactly which device is being referred to. Depending on how your driver is written (as we will see below), you can either get a direct pointer to your device from the kernel, or you can use the minor number yourself as an index into a local array of devices. Either way, the kernel itself knows almost nothing about minor numbers beyond the fact that they refer to devices implemented by your driver.

2.2The Internal Representation of Device Numbers

Within the kernel, the dev_t type (defined in) is used to hold device numbers — both the major and minor parts. As of Version 2.6.0 of the kernel, dev_t is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number. Your code should, of course, never make any assumptions about the internal organization of device numbers; it should, instead, make use of a set of macros found in . To obtain the major or minor parts of a dev_t, use:

MAJOR(dev_t dev); MINOR(dev_t dev);

If, instead, you have the major and minor numbers and need to turn them into a dev_t, use: MKDEV(int major, int minor);

Note that the 2.6 kernel can accommodate a vast number of devices, while previous kernel versions were limited to 255 major and 255 minor numbers. One assumes that the wider range will be sufficient for quite some time, but the computing field is littered with erroneous assumptions of that nature. So you should expect that the format of dev_t could change again in the future; if you write your drivers carefully, however, these changes will not be a problem.

2.3Allocating and Freeing Device Numbers

One of the first things your driver will need to do when setting up a char device is to obtain one or more device numbers to work with. The necessary function for this task is register_chrdev_region, which is declared in : int register_chrdev_region(dev_t first, unsigned int count, char *name);

Here, first is the beginning device number of the range you would like to allocate. The minor number portion of first is often 0, but there is no requirement to that effect. count is the total number of contiguous device numbers you are requesting. Note that, if count is large, the range you request could spill over to the next major number; but everything will still work properly as long as the number range you request is available. Finally, name is the name of the device that should be associated with this number range; it will appear in /proc/devices and sysfs. As with most kernel functions, the return value from

register_chrdev_region will be 0 if the allocation was successfully performed. In case of error, a negative error code will be returned, and you will not have access to the requested region.

register_chrdev_region works well if you know ahead of time exactly which device numbers you want. Often, however, you will not know which major numbers your device will use; there is a constant effort within the Linux kernel development community to move over to the use of dynamicly-allocated device numbers. The kernel will happily allocate a major number for you on the fly, but you must request this allocation by using a different function: int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);

With this function, dev is an output-only parameter that will, on successful completion, hold the first number in your allocated range. firstminor should be the requested first minor number to use; it is usually 0. The count and name parameters work like those given to request_chrdev_region. Regardless of how you allocate your device numbers, you should free them when they are no longer in use. Device numbers are freed with: void unregister_chrdev_region(dev_t first, unsigned int count);

The usual place to call unregister_chrdev_region would be in your module's cleanup function. The above functions allocate device numbers for your driver's use, but they do not tell the kernel anything about what you will actually do with those numbers. Before a user-space program can access one of those device numbers, your driver needs to connect them to its internal functions that implement the device's operations. We will describe how this connection is accomplished shortly, but there are a couple of necessary digressions to take care of first.

2.4 File Operations

we have reserved some device numbers for our use, but we have not yet connected any of our driver's operations to those numbers. The file_operations structure is how a char driver sets up this connection. The structure, defined in , is a collection of function pointers. Each open file (represented internally by a file structure, which we will examine shortly) is associated with its own set of functions (by including a field called f_op that points to a file_operations structure). The operations are mostly in charge of implementing the system calls and are therefore, named open, read, and so on. We can consider the file to be an "object" and the functions operating on it to be its "methods," using objectoriented programming terminology to denote actions declared by an object to act on itself. This is the first sign of object-oriented programming we see in the Linux kernel, and we'll see more in later chapters.

Conventionally, a file_operations structure or a pointer to one is called fops (or some variation thereof). Each field in the structure must point to the function in the driver that implements a specific operation, or be left NULL for unsupported operations. The exact behavior of the kernel when a NULL pointer is specified is different for each function, as the list later in this section shows. The following list introduces all the operations that an application can invoke on a device. We've tried to keep the list brief so it can be used as a reference, merely summarizing each operation and the default kernel behavior when a NULL pointer is used. As you read through the list of file_operations methods, you will note that a number of parameters include the string _ _user. This annotation is a form of documentation, noting that a pointer is a user-space address that cannot be directly dereferenced. For normal compilation, _ _user has no effect, but it can be used by external checking software to find misuse of user-space addresses.

The rest of the chapter, after describing some other important data structures, explains the role of the most important operations and offers hints, caveats, and real code examples. We defer discussion of the more complex operations to later chapters, because we aren't ready to dig into topics such as memory management, blocking operations, and asynchronous notification quite yet.

struct module *owner

The first file_operations field is not an operation at all; it is a pointer to the module that "owns" the structure. This field is used to prevent the module from being unloaded while its operations are in use. Almost all the time, it is simply initialized to THIS_MODULE, a macro defined in .

ssize_t (*read) (struct file *, char _ _user *, size_t, loff_t *);

Used to retrieve data from the device. A null pointer in this position causes the read system call to fail with - EINVAL (“Invalid argument”). A nonnegative return value represents the number of bytes successfully read (the return value is a “signed size” type, usually the native integer type for the target platform).

ssize_t (*write) (struct file *, const char _ _user *, size_t, loff_t *);

Sends data to the device. If NULL, -EINVAL is returned to the program calling the write system call. The return value, if nonnegative, represents the number of bytes successfully written.

int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);

The ioctl system call offers a way to issue device-specific commands (such as formatting a track of a floppy disk, which is neither reading nor writing). Additionally, a few ioctl commands are recognized by the kernel without referring to the fops table. If the device doesn’t provide an ioctl method, the system call returns an error for any request that isn’t predefined (-ENOTTY, “No such ioctl for device”).

int (*open) (struct inode *, struct file *);

Though this is always the first operation performed on the device file, the driver is not required to declare a corresponding method. If this entry is NULL, opening the device always succeeds, but your driver isn’t notified.

int (*flush) (struct file *);

The flush operation is invoked when a process closes its copy of a file descriptor for a device; it should execute (and wait for) any outstanding operations on the device. This must not be confused with the fsync operation requested by user programs. Currently, flush is used in very few drivers; the SCSI tape driver uses it, for example, to ensure that all data written makes it to the tape before the device is closed. If flush is NULL, the kernel simply ignores the user application request.

int (*release) (struct inode *, struct file *);

This operation is invoked when the file structure is being released. Like open, release can be NULL.

2.5 Char Device Registration

The kernel uses structures of type struct cdev to represent char devices internally. Before the kernel invokes your device's operations, you must allocate and register one or more of these structures. To do so, your code should include , where the structure and its associated helper functions are defined. There are two ways of allocating and initializing one of these structures. If you wish to obtain a standalone cdev structure at runtime, you may do so with code such as: struct cdev *my_cdev = cdev_alloc( ); my_cdev->ops = &my_fops;

Chances are, however, that you will want to embed the cdev structure within a device-specific structure of your own; that is what scull does. In that case, you should initialize the structure that you have already allocated with: void cdev_init(struct cdev *cdev, struct file_operations *fops);

Either way, there is one other struct cdev field that you need to initialize. Like the file_operations structure, struct cdev has an owner field that should be set to THIS_MODULE. Once the cdev structure is set up, the final step is to tell the kernel about it with a call to: int cdev_add(struct cdev *dev, dev_t num, unsigned int count);

Here, dev is the cdev structure, num is the first device number to which this device responds, and count is the number of device numbers that should be associated with the device. Often count is one, but there are situations where it makes sense to have more than one device number correspond to a specific device. Consider, for example, the SCSI tape driver, which allows user space to select operating modes (such as density) by assigning multiple minor numbers to each physical device.

There are a couple of important things to keep in mind when using cdev_add. The first is that this call can fail. If it returns a negative error code, your device has not been added to the system. It almost always succeeds, however, and that brings up the other point: as soon as cdev_add returns, your device is "live" and its operations can be called by the kernel. You should not call cdev_add until your driver is completely ready to handle operations on the device. To remove a char device from the system, call: void cdev_del(struct cdev *dev); Clearly, you should not access the cdev structure after passing it to cdev_del.

2.6 Char Driver methods

2.6.1 The Open Method

The open method is provided for a driver to do any initialization in preparation for later operations. In most drivers, open should perform the following tasks:

- Check for device-specific errors (such as device-notready or similar hardware problems)
- Initialize the device if it is being opened for the first time
- Update the f_op pointer, if necessary
- Allocate and fill any data structure to be put in filp- >private_data

The first order of business, however, is usually to identify which device is being opened. Remember that the prototype for the open method is:

int (*open)(struct inode *inode, struct file *filp);

The inode argument has the information we need in the form of its i_cdev field, which contains the cdev structure we set up before. The only problem is that we do not normally want the cdev structure itself, we want the scull_dev structure that contains that cdev structure. The C language lets programmers play all sorts of tricks to make that kind of conversion; programming such tricks is error prone, however, and leads to code that is difficult for others to read and understand. Fortunately, in this case, the kernel hackers have done the tricky stuff for us, in the form of the container_of macro, defined in *<linux/kernel.h>*: container_of(pointer, container_type, container_field);

This macro takes a pointer to a field of type container_field, within a structure of type container_type, and returns a pointer to the containing structure.

2.6.2 The Release Method

The role of the release method is the reverse of open. Sometimes you'll find that the method implementation is called device _close instead of device _release. Either way, the device method should perform the following tasks:

- Deallocate anything that open allocated in filp- >private_data
- Shut down the device on last close

The basic form of scull has no hardware to shut down, so the code required is minimal:

int scull_release(struct inode *inode, struct file *filp)

{ return 0; }

You may be wondering what happens when a device file is closed more times than it is opened. After all, the dup and fork system calls create copies of open files without calling open; each of those copies is then closed at program termination. For example, most programs don't open their stdin file (or device), but all of them end up closing it. How does a driver know when an open device file has really been closed? The answer is simple: not every close system call causes the release method to be invoked. Only the calls that actually release the device data structure invoke the method — hence its name. The kernel keeps a counter of how many times a file structure is being used. Neither fork nor dup creates a new file structure (only open does that); they just increment the counter in the existing structure. The close system call executes the release method only when the counter for the file structure drops to 0, which happens when the structure is destroyed. This relationship between the release method and the close system call guarantees that your driver sees only one release call for each open.

Note that the flush method is called every time an application calls close. However, very few drivers implement flush, because usually there's nothing to perform at close time unless release is involved. As you may imagine, the previous discussion applies even when the application terminates without explicitly closing its open files: the kernel automatically closes any file at process exit time by internally using the close system call.

2.6.3 Read and Write Methods

The read and write methods both perform a similar task, that is, copying data from and to application code. Therefore, their prototypes are pretty similar, and it's worth introducing them at the same time:

ssize_t read(struct file *filp, char _ _user *buff, size_t count, loff_t *offp);

ssize_t write(struct file *filp, const char _ _user *buff, size_t count, loff_t *offp);

For both methods, filp is the file pointer and count is the size of the requested data transfer. The buff argument points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed. Finally, offp is a pointer to a "long offset type" object that indicates the file position the user is accessing. The return value is a "signed size type"; its use is discussed later.

Let us repeat that the buff argument to the read and write methods is a user-space pointer. Therefore, it cannot be directly dereferenced by kernel code. There are a few reasons for this restriction:

- Depending on which architecture your driver is running on, and how the kernel was configured, the user-space pointer may not be valid while running in kernel mode at all. There may be no mapping for that address, or it could point to some other, random data.
- Even if the pointer does mean the same thing in kernel space, user-space memory is paged, and the memory in question might not be resident in RAM when the system call is made. Attempting to reference the user-space memory directly could generate a page fault, which is something that kernel code is not allowed to do. The result would be an "oops," which would result in the death of the process that made the system call.
- The pointer in question has been supplied by a user program, which could be buggy or malicious. If your driver ever blindly dereferences a user-supplied pointer, it provides an open doorway allowing a user-space program to access or overwrite memory anywhere in the system. If you do not wish to be responsible for compromising the security of your users' systems, you cannot ever dereference a user-space pointer directly.

Obviously, your driver must be able to access the userspace buffer in order to get its job done. This access must always be performed by special, kernel-supplied functions, however, in order to be safe. We introduce some of those functions (which are defined in) here, and the rest in the Section 6.1.4; they use some special, architecture-dependent magic to ensure that data transfers between kernel and user space happen in a safe and correct way.

The code for read and write in scull needs to copy a whole segment of data to or from the user address space. This capability is offered by the following kernel functions, which copy an arbitrary array of bytes and sit at the heart of most read and write implementations:

```
unsigned long copy_to_user(void _ _user *to, const void *from, unsigned long
count);

unsigned long copy_from_user(void *to, const void _ _user *from, unsigned long
count);
```

Although these functions behave like normal memcpy functions, a little extra care must be used when accessing user space from kernel code. The user pages being addressed might not be currently present in memory, and the virtual memory subsystem can put the process to sleep while the page is being transferred into place. This happens, for example, when the page must be retrieved from swap space. The net result for the driver writer is that any function that accesses user space must be reentrant, must be able to execute concurrently with other driver functions, and, in particular, must be in a position where it can legally sleep.

The role of the two functions is not limited to copying data to and from user-space: they also check whether the user space pointer is valid. If the pointer is invalid, no copy is

performed; if an invalid address is encountered during the copy, on the other hand, only part of the data is copied. In both cases, the return value is the amount of memory still to be copied. The scull code looks for this error return, and returns -EFAULT to the user if it's not 0.

The topic of user-space access and invalid user space pointers is somewhat advanced. However, it's worth noting that if you don't need to check the user-space pointer you can invoke _ _copy_to_user and _ _copy_from_user instead. This is useful, for example, if you know you already checked the argument. Be careful, however; if, in fact, you do not check a user-space pointer that you pass to these functions, then you can create kernel crashes and/or security holes.

As far as the actual device methods are concerned, the task of the read method is to copy data from the device to user space (using copy_to_user), while the write method must copy data from user space to the device (using copy_from_user). Each read or write system call requests transfer of a specific number of bytes, but the driver is free to transfer less data — the exact rules are slightly different for reading and writing and are described later in this chapter.

Whatever the amount of data the methods transfer, they should generally update the file position at *offp to represent the current file position after successful completion of the system call. The kernel then propagates the file position change back into the file structure when appropriate. The pread and pwrite system calls have different semantics, however; they operate from a given file offset and do not change the file position as seen by any other system calls. These calls pass in a pointer to the usersupplied position, and discard the changes that your driver makes. Figure 3-2 represents how a typical read implementation uses its arguments.

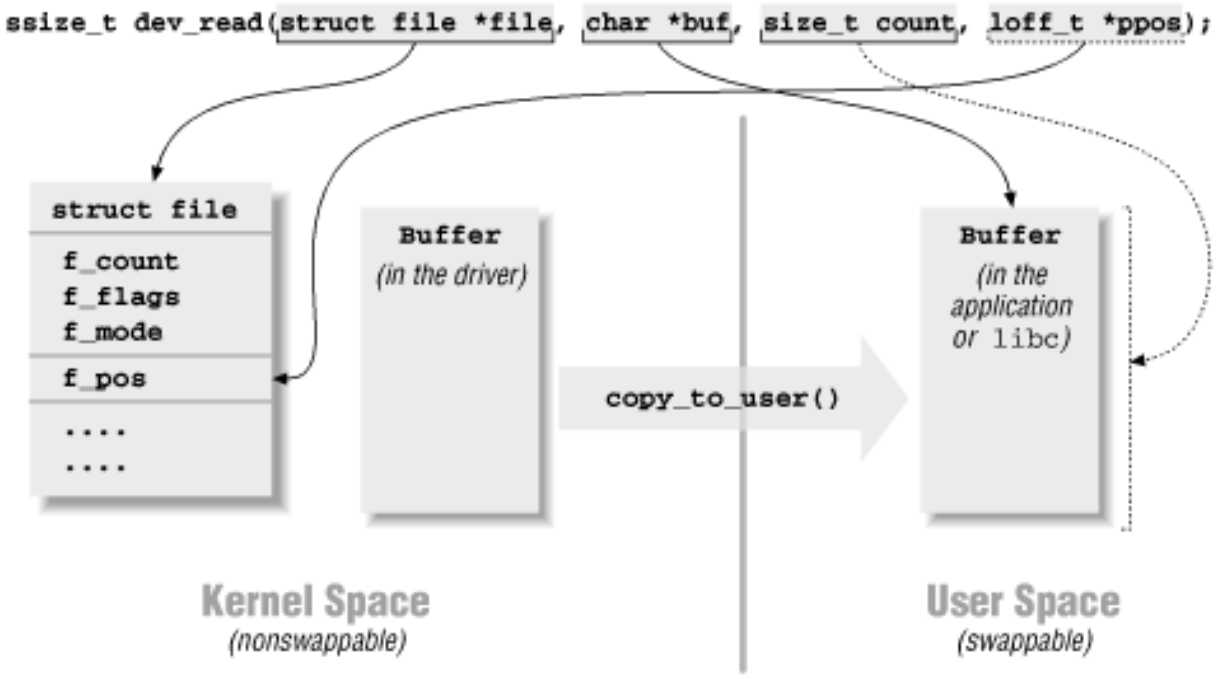


Fig.2.1 The arguments to Read

Both the read and write methods return a negative value if an error occurs. A return value greater than or equal to 0, instead, tells the calling program how many bytes have been successfully transferred. If some data is transferred correctly and then an error happens, the return value must be the count of bytes successfully transferred, and the error does not get reported until the next time the function is called. Implementing this convention requires, of course, that your driver remember that the error has occurred so that it can return the error status in the future.

Although kernel functions return a negative number to signal an error, and the value of the number indicates the kind of error that occurred (as introduced in Chapter 2), programs that run in user space always see -1 as the error return value. They need to access the errno variable to find out what happened. The user-space behavior is dictated by the POSIX standard, but that standard does not make requirements on how the kernel operates internally.

CHAPTER 3. USB PROTOCOL

What is the Universal Serial Bus (USB)?

The large diversity of ports (such as parallel, serial, midi, joystick) with their specific requirements and the lack of plug-and-play feature were almost the main reasons that pushed the most known companies in the technology domain to seek for a substitution. These companies developed the USB standard and formed the USB Implementers Forum.
The USB provided the most successful serial interface having the following characteristics:

- simplicity and flexibility (plug-and-play)
- bi-directionality
- increasing speeds
- low cost

Since developed, the USB has been continuously ameliorated always keeping compatibility with the new technologies evolution and requirements.
The STM32 USB hardware and software are compliant with USB1.1 and USB2.0 specifications and all the following sections speak about these standard compliant devices and hosts.

3.1 Speed

The USB2.0 supports three speeds:

- ***Low speed*** (**LS**): supports the transfer rate of ***1.5 Mb/s***. This speed is mainly dedicated to interactive devices (such as mouse, keyboard)
- ***Full speed*** (**FS**): supports the transfer rate of ***12 Mb/s***. This speed is mainly dedicated to phone and audio devices (such as microphone, speaker)
- ***High speed*** (**HS**): supports the transfer rate of ***480 Mb/s***. This speed is mainly dedicated to video and storage devices (such as printer, camera)

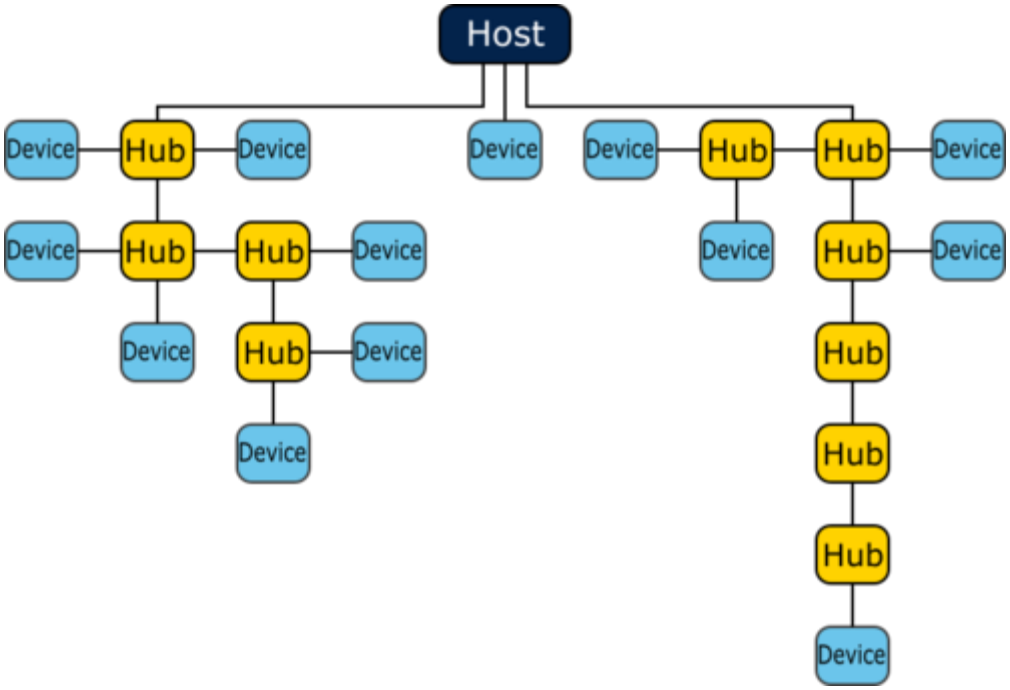
At protocol level, the USB grants a very high compatibility, so the users cannot see big differences between dealing with different speeds.

3.2 USB Interconnect Components

The USB interconnect has three main components:

- **Host or Root Hub**: it is unique for every USB system. It is responsible of initiating all transactions.
- **Function or Device**: the final point in the interconnect ensuring the user's required roles (such as keyboard, mouse, microphone)
- **Hub**: a bridge ensuring the communication between the host and many devices. It has one upstream port to be connected directly (point to point connection through USB cable) or indirectly (connection through another hub) to the host and many downstream ports to be connected directly or indirectly to the USB functions.

The previous components can be connected to each other through USB cables with a maximum length of 5 meters.
The maximum number of hubs connected in series allowed by the USB specification is 5.
Thus, a function or device can be connected to the host through one or more hub(s).
The following figure provides an example of USB system components connection.



3.3 USB Interconnect Topology

The USB physical interconnect is characterized by a tired star topology. Each star has a hub at the center with one upstream connection directly or indirectly with the host and one or more downstream connection(s) with function or other hub.
A maximum of 127 devices (functions or hubs) can be connected to one host (root hub) with a maximum of 5 hubs connected in series.

3.4 Power

Generally, the host (root hub) provides power for functions direct connection. Some hubs may supply power for directly connected downstream functions. For the functions, there are two types:

- **Bus powered functions**: these devices rely totally on the bus power coming from the upstream hub.
- **Self powered functions**: these devices are capable of providing their own power independently from the bus.

3.5 System configuration

The USB system has an intelligent mechanism to detect the device attachment and detachment at any time.

- Device attachment: the host can detect at any time the detachment of a device by continuously querying a bit for all the connected hub ports. Once a device is attached, the host enables the port and gives the device a unique address then establish a communication with this newly connected device to conclude if it is a function or a hub.
- Device detachment: once a device is detached, the corresponding port is disabled. If a hub is detached, all the downstream devices' ports that were attached to the removed hub are disabled and the detached hub's upstream port is disabled.
- Bus enumeration: it is a set of hardware and software events allowing the host to continuously manage the process of events of the newly connected or disconnected devices. It includes also the set of events ensuring the removal of a device.

3.6 Class

Depending on the required USB device functionality and application, the device and the host features vary. The USB IF defines a variety of classes ensuring the classification according to the required functionality. Every class defines an association of:

- Data including characteristics that must be stored within the device and given to the host when requested.
- Software drivers stored within the host and loaded after negotiation with the device and after discovery of its characteristics.

An overview of all the defined USB classes and codes is provided by the USB IF at this link.
For more details about the STM32 USB device classes, please refer to The USB Device Classes section.
For more details about the STM32 USB Host classes, please refer to The USB Host Classes section.

3.7 USB Data Transfers

The USB communication is based on four main transfer types:

- **Control transfer**: mainly used for the configuration data of the newly attached device.
- **Bulk transfer**: used for large amounts of data transmission or reception.
- **Interrupt transfer**: used for limited data transmission with minimal latency.
- **Isochronous transfer**: used for data transfer with real-time requirements.

More detailed explanations of the USB characteristics and data flow are explained with code examples in the following pages.

3.8 Data D

irection

As it is outlined in the previous section, the USB supports only 4 transfer types. For all the transfers, it is always the host who initiates the communication with the device. In fact, the USB can transfer data in two directions where the reference is always the host:

- data **out** is the data that is transferred from the host to the device
- data **in** is the data that is transferred from the device into the host

3.9 Transactions

Each transfer independently from its type is based on one or more transaction(s) to ensure the information exchange between a host and a device. Each transaction itself consists of up to three phases or packets:

- **Token**: it is always required to start every transaction. It is USB packet sent from the host to the device providing some information about the transaction (such as type, destination, data phase direction)
- **Data**: this phase is optional. Its existence, direction and size are indicated within the token packet. During this phase, the source of data sends the specified data if it exists.

- **Handshake**: this phase is optional. Its existence and direction can be inferred from the token packet. In fact, depending on the direction of data specified in the token packet, the data destination sends this packet to acknowledge the data reception status.

3.10 Requests

As mentioned previously, the host always initiates the communication with the device by sending request. All USB requests have common fields. Each field size, value and signification is known by the device and the host. It is always the host who is responsible of filling a request field and sending it and, on the other side, the device receives the request and decodes it to be prepared for the next communication phase.

3.11 Descriptor

It is a set of data blocks stocked within the device's memory and organized in a defined manner known by the device and the host. Some descriptors are required for all the devices and the host cannot continue the communication with any device missing a required descriptor. Some descriptors are optional and can differ from a device to another depending on its functionalities. Every descriptor includes a defined set of information about the device and will be sent to the host as answer to a defined request.

3.12 Endpoint

It is a source or destination data buffer that must be implemented on the device side. Each amount of data that is received from or sent to the host will be placed into an endpoint. Each endpoint is uniquely characterized by a number and direction which means that for a given number, there is a unique pair of endpoints of the same type and number but each one deals with the data of only one direction:

- **IN Endpoint** for data that will be transferred into the host.
- **OUT Endpoint** for data that will be transferred from the host.

A device has always more than one endpoint pair identified by their numbers:

- Endpoint zero: it is a pair of OUT and IN endpoints dedicated for control data transfers which means it is a control endpoint. It is required to establish the first communication transfers between the host and the device while the other endpoints are not yet configured.
- Other endpoints: will be configured after negotiation between the host and the device. Each endpoint is independent from the other and can handle a different transfer type.

3.13 Address

A unique value assigned by the host and it varies from 1 to 127. Address 0 is always given to the newly attached device before being addressed by the host.

3.14 Enumeration

It is the procedure ensuring the control of the device status changes and the real time management of any device attachment and detachment. During this step, there is a combination of hardware and software negotiation allowing the host to decode the device nature. At device software level, this procedure ensures the correct reception and decoding of the host request and then device state modification accordingly.

3.15 Device status

From being completely detached until being completely recognized by the USB Host and ensuring its function, the USB device goes through consecutive stages:

- **Attached**: it is the stage when the device is physically connected to the USB host but not yet powered. This stage is mainly ensured by hardware.
- **Powered**: it is the second stage and is corresponding to a device that was attached to USB Host and just powered. This stage is mainly ensured by hardware.
- **Default**: this stage is reached when the attached device is powered then reset by the host. This stage is assigned to the device by its software each time the device is newly attached then powered then reset or an old attached device received a reset. At this stage, the USB Device operates with convenient speed (selected by hardware during reset) and has the default address which is the address number 0.
- **Addressed**: after going through all the previous stages, the USB device reaches this state by receiving its unique address (different from 0) from the host. This stage is reached after correct process of the host request by the device software.
- **Configured**: the device reaches this stage after receiving the convenient request from the host with a non-zero configuration number. This stage is reached after correct process of the host request by the device software.
- **Suspended**: the device must enter this stage if there is no data on the traffic for a known period that depends on the speed. In fact, the host forces the device to enter this state electrically depending on its speed. When detecting this electrical indication, the USB device software must change its state into suspended.

CHAPTER 4. USB DRIVER

The universal serial bus (USB) is a connection between a host computer and a number of peripheral devices. It was originally created to replace a wide range of slow and different buses — the parallel, serial, and keyboard connections — with a single bus type that all devices could connect to. USB has grown beyond these slow connections and now supports almost every type of device that can be connected to a PC. The latest revision of the USB specification added high-speed connections with a theoretical speed limit of 480 MBps.

A USB subsystem is not laid out as a bus; it is rather a tree built out of several point-to-point links. The links are four-wire cables (ground, power, and two signal wires) that connect a device and a hub, just like twistedpair Ethernet. The USB host controller is in charge of asking every USB device if it has any data to send. Because of this topology, a USB device can never start sending data without first being asked to by the host controller. This configuration allows for a very easy plug-and-play type of system, whereby devices can be automatically configured by the host computer.

The bus is very simple at the technological level, as it's a single-master implementation in which the host computer polls the various peripheral devices. Despite this intrinsic limitation, the bus has some interesting features, such as the ability for a device to request a fixed bandwidth for its data transfers in order to reliably support video and audio I/O. Another important feature of USB is that it acts merely as a communication channel between the device and the host, without requiring specific meaning or structure to the data it delivers.

The USB protocol specifications define a set of standards that any device of a specific type can follow. If a device follows that standard, then a special driver for that device is not necessary. These different types are called classes and consist of things like storage devices, keyboards, mice, joysticks, network devices, and modems. Other types of devices that do not fit into these classes require a special vendor-specific driver to be written for that specific device. Video devices and USB-to-serial devices are a good example where there is no defined standard, and a driver is needed for every different device from different manufacturers.

These features, together with the inherent hotplug capability of the design, make USB a handy, low-cost mechanism to connect (and disconnect) several devices to the computer without the need to shut the system down, open the cover, and swear over screws and wires. The Linux kernel supports two main types of USB drivers: drivers on a host system and

drivers on a device. The USB drivers for a host system control the USB devices that are plugged into it, from the host's point of view (a common USB host is a desktop computer.) The USB drivers in a device, control how that single device looks to the host computer as a USB device. As the term "USB device drivers" is very confusing, the USB developers have created the term "USB gadget drivers" to describe the drivers that control a USB device that connects to a computer (remember that Linux also runs in those tiny embedded devices, too.) This chapter details how the USB system that runs on a desktop computer works. USB gadget drivers are outside the realm of this book at this point in time.

USB drivers live between the different kernel subsytems (block, net, char, etc.) and the USB hardware controllers. The USB core provides an interface for USB drivers to use to access and control the USB hardware, without having to worry about the different types of USB hardware controllers that are present on the system.

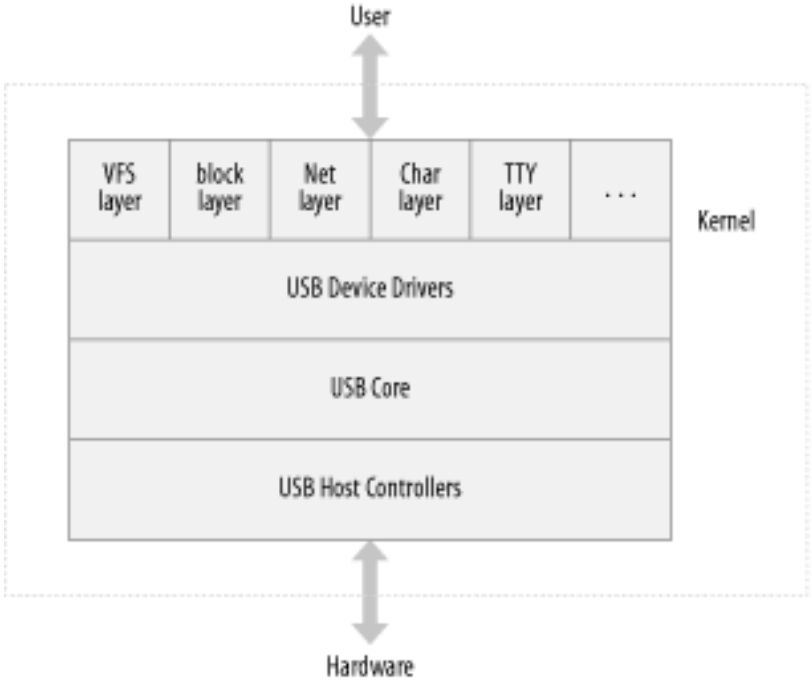


Fig.4.1 USB driver overview

4.1 USB Device Basics

A USB device is a very complex thing, as described in the official USB documentation (available at http://www.usb.org). Fortunately, the Linux kernel provides a subsystem called the USB core to handle most of the complexity. This chapter describes the interaction between a driver and the USB core. Figure 4.1 shows how USB devices consist of configurations, interfaces, and endpoints and how USB drivers bind to USB interfaces, not the entire USB device.

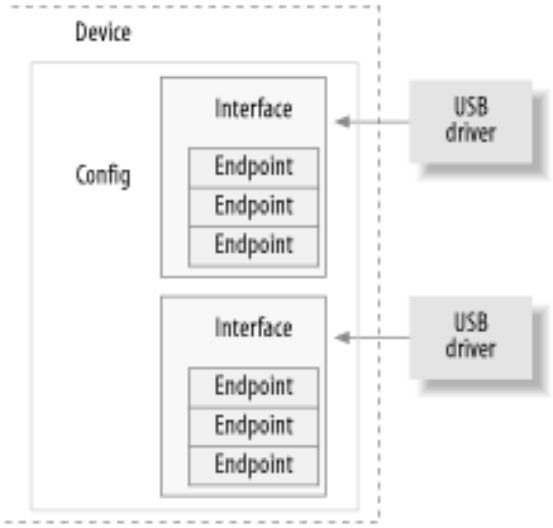


Fig 4.1 USB device overview

4.1.1 Endpoints

The most basic form of USB communication is through something called an endpoint. A USB endpoint can carry data in only one direction, either from the host computer to the device (called an OUT endpoint) or from the device to the host computer (called an IN endpoint). Endpoints can be thought of as unidirectional pipes.

A USB endpoint can be one of four different types that describe how the data is transmitted:

CONTROL

Control endpoints are used to allow access to different parts of the USB device. They are commonly used for configuring the device, retrieving information about the device, sending commands to the device, or retrieving status reports about the device. These endpoints are usually small in size. Every USB device has a control endpoint called

“endpoint 0” that is used by the USB core to configure the device at insertion time. These transfers are guaranteed by the USB protocol to always have enough reserved bandwidth to make it through to the device.

INTERRUPT

Interrupt endpoints transfer small amounts of data at a fixed rate every time the USB host asks the device for data. These endpoints are the primary transport method for USB keyboards and mice. They are also commonly used to send data to USB devices to control the device, but are not generally used to transfer large amounts of data. These transfers are guaranteed by the USB protocol to always have enough reserved bandwidth to make it through.

BULK

Bulk endpoints transfer large amounts of data. These endpoints are usually much larger (they can hold more characters at once) than interrupt endpoints. They are common for devices that need to transfer any data that must get through with no data loss. These transfers are not guaranteed by the USB protocol to always make it through in a specific amount of time. If there is not enough room on the bus to send the whole BULK packet, it is split up across multiple transfers to or from the device. These endpoints are common on printers, storage, and network devices.

ISOCHRONOUS

Isochronous endpoints also transfer large amounts of data, but the data is not always guaranteed to make it through. These endpoints are used in devices that can handle loss of data, and rely more on keeping a constant stream of data flowing. Real-time data collections, such as audio and video devices, almost always use these endpoints.

Control and bulk endpoints are used for asynchronous data transfers, whenever the driver decides to use them. Interrupt and isochronous endpoints are periodic. This means that these endpoints are set up to transfer data at fixed times continuously, which causes their bandwidth to be reserved by the USB core. USB endpoints are described in the kernel with the structure struct usb_host_endpoint. This structure contains the real endpoint information in another structure called struct usb_endpoint_descriptor. The latter structure contains all of the USB-specific data in the exact format that the device itself specified. The fields of this structure that drivers care about are:

bEndpointAddress

This is the USB address of this specific endpoint. Also included in this 8-bit value is the direction of the endpoint. The bitmasks USB_DIR_OUT and USB_DIR_IN can be placed against this field to determine if the data for this endpoint is directed to the device or to the host.

bmAttributes

This is the type of endpoint. The bitmask USB_ENDPOINT_XFERTYPE_MASK should be placed against this value in order to determine if the endpoint is of type USB_ENDPOINT_XFER_ISOC, USB_ENDPOINT_XFER_BULK, or of type USB_ENDPOINT_XFER_INT. These macros define a isochronous, bulk, and interrupt endpoint, respectively.

wMaxPacketSize

This is the maximum size in bytes that this endpoint can handle at once. Note that it is possible for a driver to send amounts of data to an endpoint that is bigger than this value, but the data will be divided up into wMaxPacketSize chunks when actually transmitted to the device. For high-speed devices, this field can be used to support a high-bandwidth mode for the endpoint by using a few extra bits in the upper part of the value. See the USB specification for more details about how this is done.

bInterval

If this endpoint is of type interrupt, this value is the interval setting for the endpoint — that is, the time between interrupt requests for the endpoint. The value is represented in milliseconds.

4.1.2 Interfaces

USB endpoints are bundled up into interfaces. USB interfaces handle only one type of a USB logical connection, such as a mouse, a keyboard, or a audio stream. Some USB devices have multiple interfaces, such as a USB speaker that might consist of two interfaces: a USB keyboard for the buttons and a USB audio stream. Because a USB interface represents basic functionality, each USB driver controls an interface; so, for the speaker example, Linux needs two different drivers for one hardware device.

USB interfaces may have alternate settings, which are different choices for parameters of the interface. The initial state of a interface is in the first setting, numbered 0. Alternate settings can be used to control individual endpoints in different ways, such as to reserve different amounts of USB bandwidth for the device. Each device with an isochronous endpoint uses alternate settings for the same interface.

USB interfaces are described in the kernel with the struct usb_interface structure. This structure is what the USB core passes to USB drivers and is what the USB driver then is in charge of controlling. The important fields in this structure are:

struct usb_host_interface *altsetting

An array of interface structures containing all of the alternate settings that may be selected for this interface. Each struct usb_host_interface consists of a set of endpoint configurations as defined by the struct usb_host_endpoint structure described above. Note that these interface structures are in no particular order.

unsigned num_altsetting

The number of alternate settings pointed to by the altsetting pointer.

struct usb_host_interface *cur_altsetting

A pointer into the array altsetting, denoting the currently active setting for this interface.

int minor

If the USB driver bound to this interface uses the USB major number, this variable contains the minor number assigned by the USB core to the interface. This is valid only after a successful call to usb_register_dev (described later in this chapter). There are other fields in the struct usb_interface structure, but USB drivers do not need to be aware of them.

4.1.3 Configuration

USB interfaces are themselves bundled up into configurations. A USB device can have multiple configurations and might switch between them in order to change the state of the device. For example, some devices that allow firmware to be downloaded to them contain multiple configurations to accomplish this. A single configuration can be enabled only at one point in time. Linux does not handle multiple configuration USB devices very well, but, thankfully, they are rare.

Linux describes USB configurations with the structure struct usb_host_config and entire USB devices with the structure struct usb_device. USB device drivers do not generally ever need to read or write to any values in these structures, so they are not defined in detail here. The curious reader can find descriptions of them in the file include/linux/usb.h in the kernel source tree.

A USB device driver commonly has to convert data from a given struct usb_interface structure into a struct usb_device structure that the USB core needs for a wide range of function calls. To do this, the function interface_to_usbdev is provided. Hopefully, in the

future, all USB calls that currently need a struct usb_device will be converted to take a struct usb_interface parameter and will not require the drivers to do the conversion.

So to summarize, USB devices are quite complex and are made up of lots of different logical units. The relationships among these units can be simply described as follows:

- Devices usually have one or more configurations.
- Configurations often have one or more interfaces.
- Interfaces usually have one or more settings.
- Interfaces have zero or more endpoints.

4.2 Writing a USB Driver

The approach to writing a USB device driver is similar to a pci_driver: the driver registers its driver object with the USB subsystem and later uses vendor and device identifiers to tell if its hardware has been installed.

4.2.1 What Devices Does the Driver Support?

The struct usb_device_id structure provides a list of different types of USB devices that this driver supports. This list is used by the USB core to decide which driver to give a device to, and by the hotplug scripts to decide which driver to automatically load when a specific device is plugged into the system.

The struct usb_device_id structure is defined with the following fields:

_ _u16 match_flags

Determines which of the following fields in the structure the device should be matched against. This is a bit field defined by the different USB_DEVICE_ID_MATCH_* values specified in the include/linux/mod_devicetable.h file. This field is usually never set directly but is initialized by the USB_DEVICE type macros described later.

_ _u16 idVendor

The USB vendor ID for the device. This number is assigned by the USB forum to its members and cannot be made up by anyone else.

_ _u16 idProduct

The USB product ID for the device. All vendors that have a vendor ID assigned to them can manage their product IDs however they choose to.

_ _u16 bcdDevice_lo

_ _u16 bcdDevice_hi

Define the low and high ends of the range of the vendorassigned product version number. The bcdDevice_hi value is inclusive; its value is the number of the highestnumbered device. Both of these values are expressed in binary-coded decimal (BCD) form. These variables, combined with the idVendor and idProduct, are used to define a specific version of a device.

_ _u8 bDeviceClass

_ _u8 bDeviceSubClass

_ _u8 bDeviceProtocol

Define the class, subclass, and protocol of the device, respectively. These numbers are assigned by the USB forum and are defined in the USB specification. These values specify the behavior for the whole device, including all interfaces on this device.

_ _u8 bInterfaceClass

_ _u8 bInterfaceSubClass

_ _u8 bInterfaceProtocol

Much like the device-specific values above, these define the class, subclass, and protocol of the individual interface, respectively. These numbers are assigned by the USB forum and are defined in the USB specification.

kernel_ulong_t driver_info

This value is not used to match against, but it holds information that the driver can use to differentiate the different devices from each other in the probe callback function to the USB driver.

As with PCI devices, there are a number of macros that are used to initialize this structure:

USB_DEVICE(vendor, product)

Creates a struct usb_device_id that can be used to match only the specified vendor and product ID values. This is very commonly used for USB devices that need a specific driver.

USB_DEVICE_VER(vendor, product, lo, hi)

Creates a struct usb_device_id that can be used to match only the specified vendor and product ID values within a version range.

USB_DEVICE_INFO(class, subclass, protocol)

Creates a struct usb_device_id that can be used to match a specific class of USB devices.

USB_INTERFACE_INFO(class, subclass, protocol)

Creates a struct usb_device_id that can be used to match a specific class of USB interfaces. So, for a simple USB device driver that controls only a single USB device from a single vendor, the struct usb_device_id table would be defined as:

/* table of devices that work with this driver */ static struct usb_device_id skel_table [ ] = { { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) }, { } /* Terminating entry */ }; MODULE_DEVICE_TABLE (usb, skel_table);

As with a PCI driver, the MODULE_DEVICE_TABLE macro is necessary to allow user-space tools to figure out what devices this driver can control. But for USB drivers, the string usb must be the first value in the macro.

4.2.2 Registering a USB Driver

The main structure that all USB drivers must create is a struct usb_driver. This structure must be filled out by the USB driver and consists of a number of function callbacks and variables that describe the USB driver to the USB core code:

struct module *owner

Pointer to the module owner of this driver. The USB core uses it to properly reference count this USB driver so that it is not unloaded at inopportune moments. The variable should be set to the THIS_MODULE macro.

const char *name

Pointer to the name of the driver. It must be unique among all USB drivers in the kernel and is normally set to the same name as the module name of the driver. It shows up in sysfs under /sys/bus/usb/drivers/ when the driver is in the kernel.

const struct usb_device_id *id_table

Pointer to the struct usb_device_id table that contains a list of all of the different kinds of USB devices this driver can accept. If this variable is not set, the probe function callback in the USB driver is never called. If you want your driver always to be called for every USB device in the system, create a entry that sets only the driver_info field:

static struct usb_device_id usb_ids[ ] = {

{.driver_info = 42},

{ }

};

int (*probe) (struct usb_interface *intf, const struct usb_device_id *id)

Pointer to the probe function in the USB driver. This function is called by the USB core when it thinks it has a struct usb_interface that this driver can handle. A pointer to the struct usb_device_id that the USB core used to make this decision is also passed to this function. If the USB driver claims the struct usb_interface that is passed to it, it should initialize the device properly and return 0. If the driver does not want to claim the device, or an error occurs, it should return a negative error value.

void (*disconnect) (struct usb_interface *intf)

Pointer to the disconnect function in the USB driver. This function is called by the USB core when the struct usb_interface has been removed from the system or when the driver is being unloaded from the USB core.

So, to create a value struct usb_driver structure, only five fields need to be initialized:
static struct usb_driver skel_driver = {

.owner = THIS_MODULE,

.name = "skeleton",

.id_table = skel_table,

.probe = skel_probe,

.disconnect = skel_disconnect, };

The struct usb_driver does contain a few more callbacks, which are generally not used very often, and are not required in order for a USB driver to work properly:

int (*ioctl) (struct usb_interface *intf, unsigned int code, void *buf)

Pointer to an ioctl function in the USB driver. If it is present, it is called when a user-space program makes a ioctl call on the usbfs filesystem device entry associated with a USB device attached to this USB driver. In pratice, only the USB hub driver uses this ioctl, as there is no other real need for any other USB driver to use it.

int (*suspend) (struct usb_interface *intf, u32 state)

Pointer to a suspend function in the USB driver. It is called when the device is to be suspended by the USB core.

int (*resume) (struct usb_interface *intf)

Pointer to a resume function in the USB driver. It is called when the device is being resumed by the USB core.

To register the struct usb_driver with the USB core, a call to usb_register_driver is made with a pointer to the struct usb_driver. This is traditionally done in the module initialization code for the USB driver:

static int _ _init usb_skel_init(void)

{ int result;

/* register this driver with the USB subsystem */

result = usb_register(&skel_driver); if (result) err("usb_register failed. Error number %d", result);

return result; }

When the USB driver is to be unloaded, the struct usb_driver needs to be unregistered from the kernel. This is done with a call to usb_deregister_driver. When this call happens, any USB interfaces that were currently bound to this driver are disconnected, and the disconnect function is called for them.

static void _ _exit usb_skel_exit(void) {

/* deregister this driver with the USB subsystem */

usb_deregister(&skel_driver); }

4.2.3 Probe and disconnect

In the struct usb_driver structure described in the previous section, the driver specified two functions that the USB core calls at appropriate times. The probe function is called when a device is installed that the USB core thinks this driver should handle; the probe function should perform checks on the information passed to it about the device and decide whether

the driver is really appropriate for that device. The disconnect function is called when the driver should no longer control the device for some reason and can do clean-up.

Both the probe and disconnect function callbacks are called in the context of the USB hub kernel thread, so it is legal to sleep within them. However, it is recommended that the majority of work be done when the device is opened by a user if possible, in order to keep the USB probing time to a minimum. This is because the USB core handles the addition and removal of USB devices within a single thread, so any slow device driver can cause the USB device detection time to slow down and become noticeable by the user.

In the probe function callback, the USB driver should initialize any local structures that it might use to manage the USB device. It should also save any information that it needs about the device to the local structure, as it is usually easier to do so at this time. As an example, USB drivers usually want to detect what the endpoint address and buffer sizes are for the device, as they are needed in order to communicate with the device. Here is some example code that detects both IN and OUT endpoints of BULK type and saves some information about them in a local device structure.

usb_get_intfdata is usually called in the open function of the USB driver and again in the disconnect function. Thanks to these two functions, USB drivers do not need to keep a static array of pointers that store the individual device structures for all current devices in the system. The indirect reference to device information allows an unlimited number of devices to be supported by any USB driver. If the USB driver is not associated with another type of subsystem that handles the user interaction with the device (such as input, tty, video, etc.), the driver can use the USB major number in order to use the traditional char driver interface with user space. To do this, the USB driver must call the usb_register_dev function in the probe function when it wants to register a device with the USB core. Make sure that the device and driver are in a proper state to handle a user wanting to access the device as soon as this function is called.

The usb_register_dev function requires a pointer to a struct usb_interface and a pointer to a struct usb_class_driver. This struct usb_class_driver is used to define a number of different parameters that the USB driver wants the USB core to know when registering for a minor number. This structure consists of the following variables:

char *name

The name that sysfs uses to describe the device. A leading pathname, if present, is used only in devfs and is not covered in this book. If the number of the device needs to be in the name, the characters %d should be in the name string. For example, to create the devfs name usb/foo1 and the sysfs class name foo1, the name string should be set to usb/foo%d.

struct file_operations *fops;

Pointer to the struct file_operations that this driver has defined to use to register as the character device.

mode_t mode; The mode for the devfs file to be created for this driver; unused otherwise. A typical setting for this variable would be the value S_IRUSR combined with the value S_IWUSR, which would provide only read and write access by the owner of the device file.

int minor_base;

This is the start of the assigned minor range for this driver. All devices associated with this driver are created with unique, increasing minor numbers beginning with this value. Only 16 devices are allowed to be associated with this driver at any one time unless the CONFIG_USB_DYNAMIC_MINORS configuration option has been enabled for the kernel. If so, this variable is ignored, and all minor numbers for the device are allocated on a first-come, first-served manner. It is recommended that systems that have enabled this option use a program such as udev to manage the device nodes in the system, as a static /dev tree will not work properly.

When the USB device is disconnected, all resources associated with the device should be cleaned up, if possible. At this time, if usb_register_dev has been called to allocate a minor number for this USB device during the probe function, the function usb_deregister_dev must be called to give the minor number back to the USB core. In the disconnect function, it is also important to retrieve from the interface any data that was previously set with a call to usb_set_intfdata.

Note the call to lock_kernel in the previous code snippet. This takes the big kernel lock, so that the disconnect callback does not encounter a race condition with the open call when trying to get a pointer to the correct interface data structure. Because the open is called with the big kernel lock taken, if the disconnect also takes that same lock, only one portion of the driver can access and then set the interface data pointer. Just before the disconnect function is called for a USB device, all urbs that are currently in transmission for the device are canceled by the USB core, so the driver does not have to explicitly call usb_kill_urb for these urbs. If a driver tries to submit a urb to a USB device after it has been disconnected with a call to usb_submit_urb, the submission will fail with an error value of -EPIPE.

CHAPTER 5. Getting Started with STM32 and USB

5.1 What does the STM32 support?

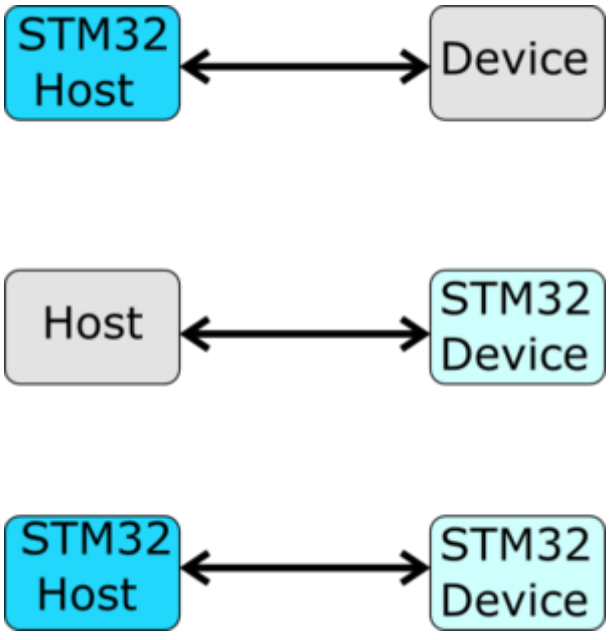
All the STM32 families (not all products) include the USB IP. Depending on the hardware, each STM32 MCU including the USB can support:

- Device in FS speed only.
- OTG (dual role: device and host) in FS speed.
- OTG in HS speed.

Some STM32 MCUs include two USB peripherals and support both OTG in FS and HS speeds.

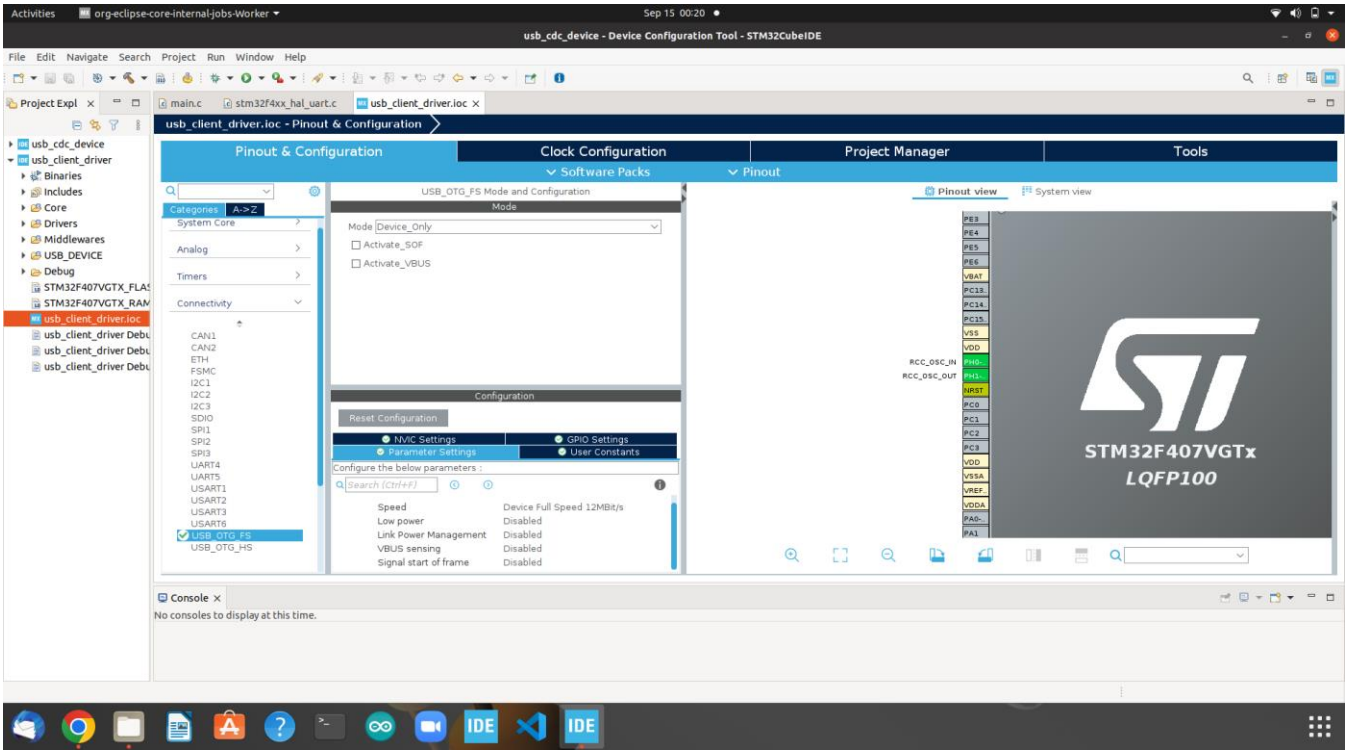
5.2 What role can the STM32 MCU play within a USB system?

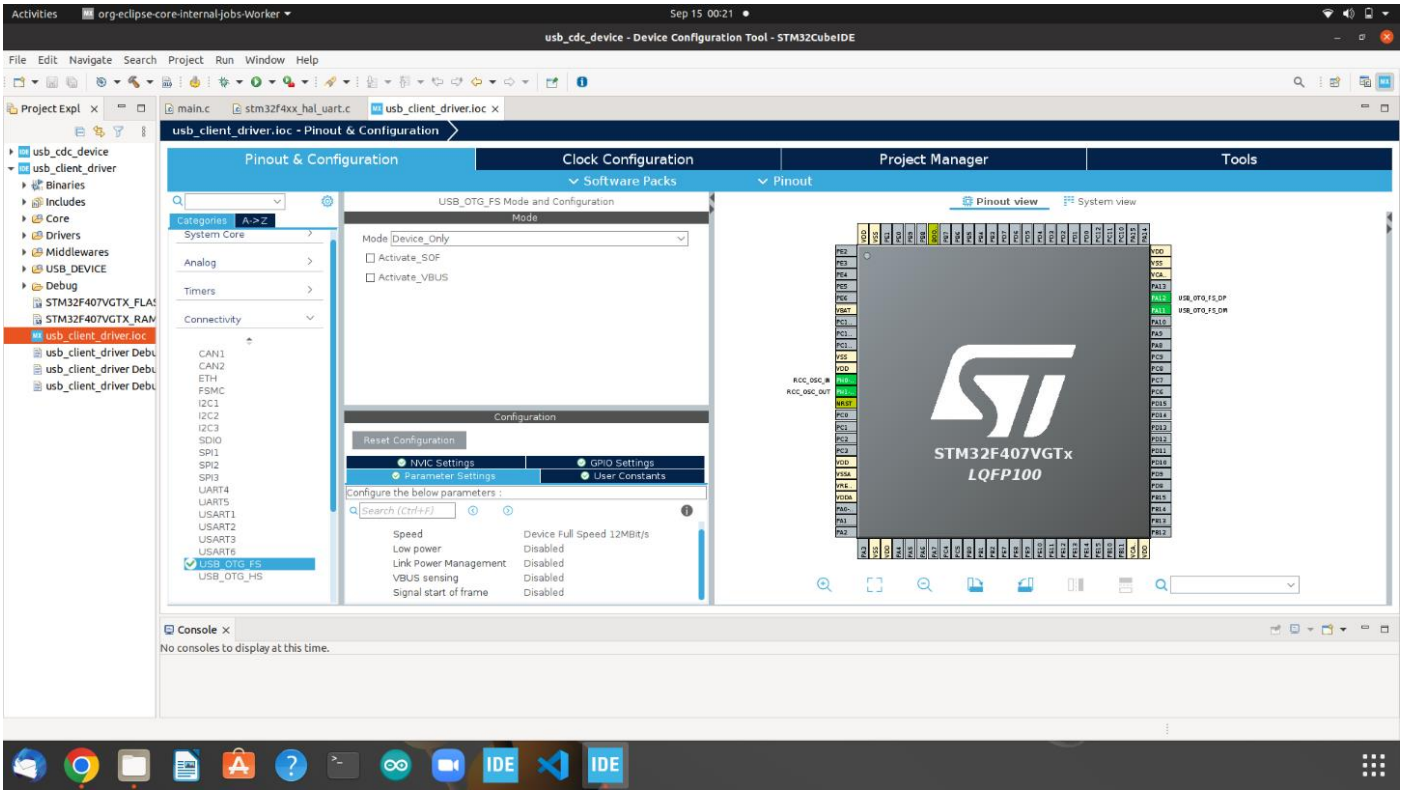
A basic USB system can be established by a host and a device attached with a USB cable. The following figure shows the possible roles that can keep the STM32 MCU's USB:

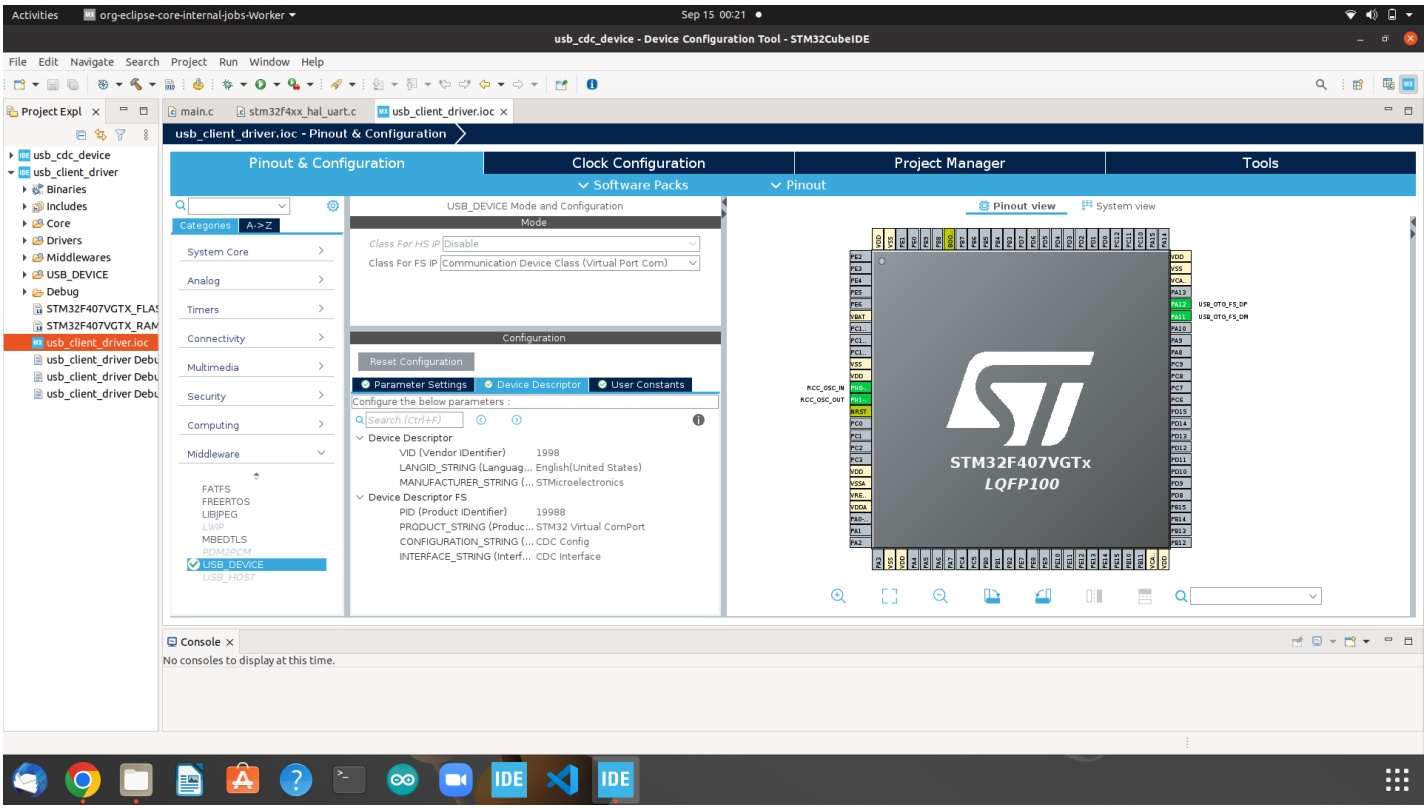


For more complex USB system, the device can be attached to the host through one or more hub(s).

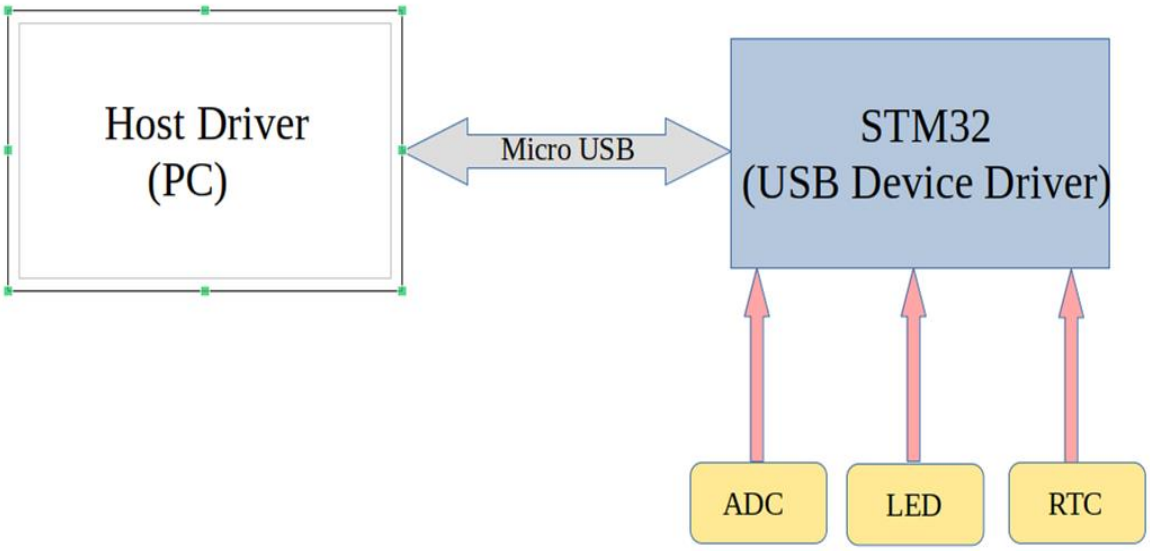
5.3 Creating STM32 has a USB client device using STM32 CUBE IDE



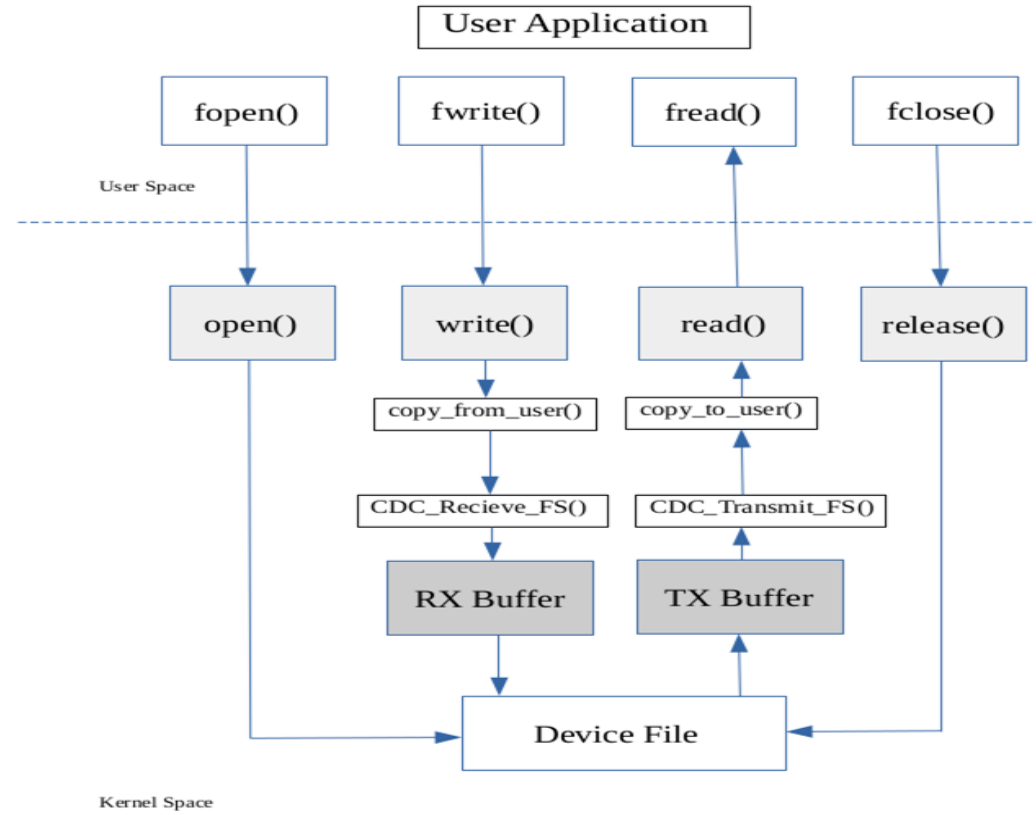

usb_cdc_device - Device Configuration Tool - STM32CubeIDE
usb_client_driver.ioc - Pinout & Configuration
Pinout & Configuration
Clock Configuration
Project Manager
Tools
Software Packs
Pinout
USB_OTG_FS Mode and Configuration
Mode
Mode Device_Only
Activate_SOF
Activate_VBUS
Configuration
Reset Configuration
NVIC Settings
GPIO Settings
Parameter Settings
User Constants
Configure the below parameters :
Speed Device Full Speed 12MBit/s
Low power Disabled
Link Power Management Disabled
VBUS sensing Disabled
Signal start of frame Disabled
Pinout view
System view
STM32F407VGTx
LQFP100
Console
No consoles to display at this time.


usb_cdc_device - Device Configuration Tool - STM32CubeIDE
usb_client_driver.ioc - Pinout & Configuration
Pinout & Configuration
Clock Configuration
Project Manager
Tools
Software Packs
Pinout
USB_DEVICE Mode and Configuration
Mode
Class For HS IP Disable
Class For FS IP Communication Device Class (Virtual Port Com)
Configuration
Reset Configuration
Parameter Settings
Device Descriptor
User Constants
Configure the below parameters :
Device Descriptor
VID (Vendor IDentifier) 1998
LANGID_STRING (Languag... English(United States)
MANUFACTURER_STRING (... STMicroelectronics
Device Descriptor FS
PID (Product IDentifier) 19988
PRODUCT_STRING (Produc... STM32 Virtual ComPort
CONFIGURATION_STRING (... CDC Config
INTERFACE_STRING (Interf... CDC Interface
Pinout view
System view
STM32F407VGTx
LQFP100
Console
No consoles to display at this time.

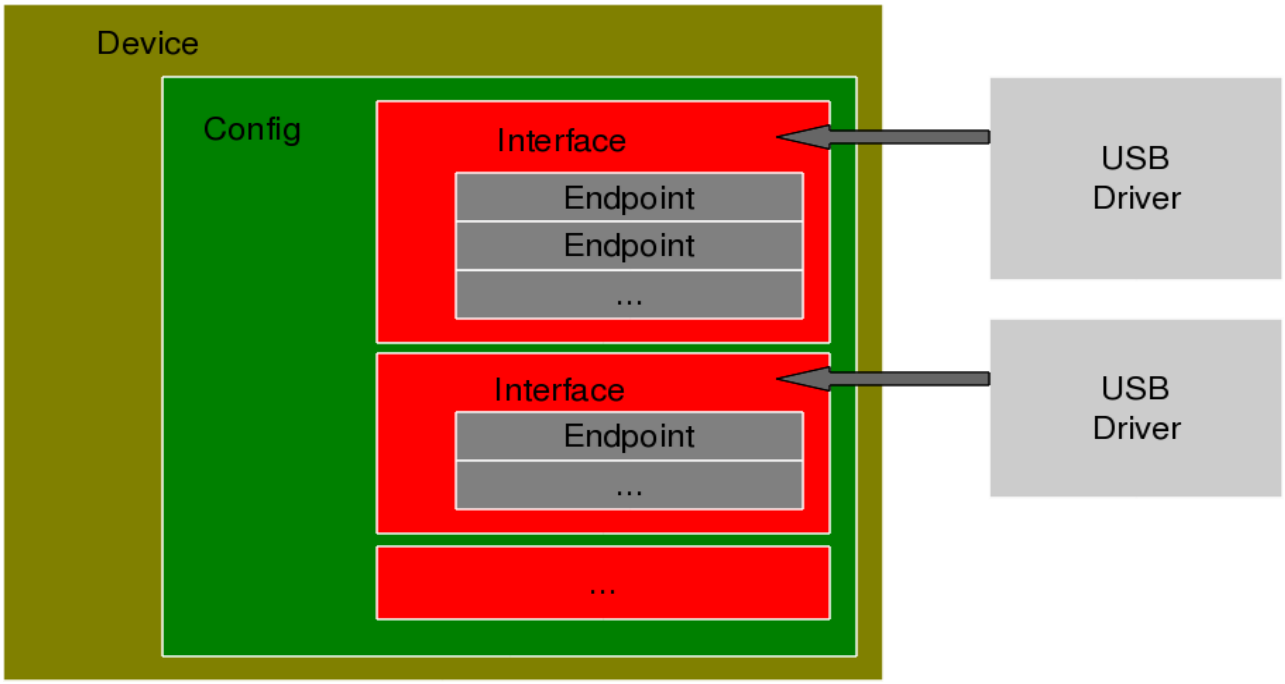
5.4 Block Diagram



5.5 Flow Chart



5.6 USB Device Overview



5.7 Device Library organization