Лекции Рязанова Н.Ю.

Управление устройствами в Linux

https://makelinux.github.io/kernel/map/

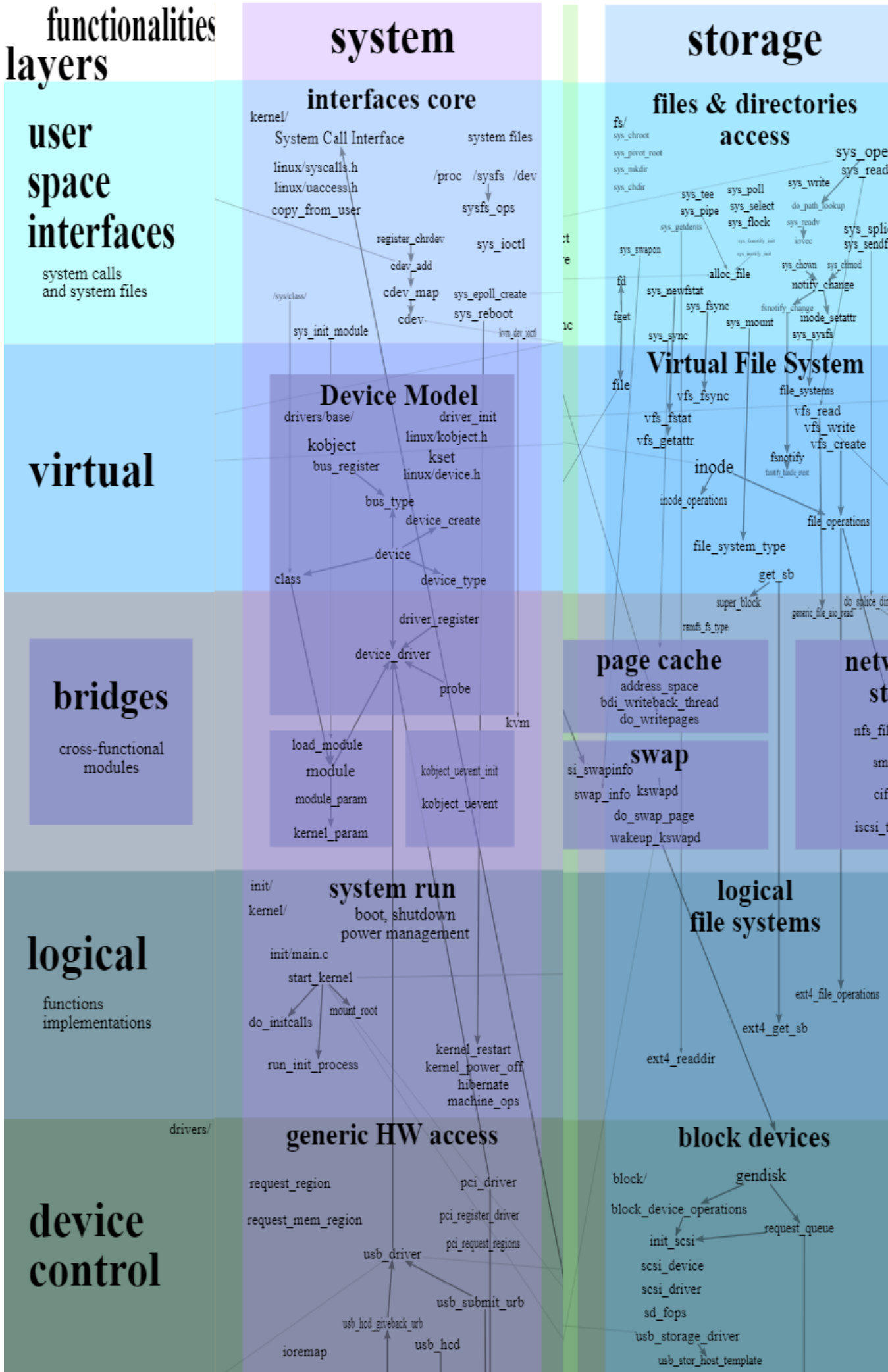
functionalities
layers
system
storage
user space interfaces
system calls and system files
interfaces core
kernel/
System Call Interface
system files
linux/syscalls.h
linux/uaccess.h
copy_from_user
/proc /sysfs /dev
sysfs_ops
register_chrdev
sys_ioctl
cdev_add
cdev_map
sys_epoll_create
sys_reboot
cdev
/sys/class/
sys_init_module
kvm_dev_ioctl
files & directories access
fs/
sys_chroot
sys_pivot_root
sys_mkdir
sys_chdir
sys_tee
sys_poll
sys_write
sys_pipe
sys_select
do_path_lookup
sys_flock
sys_readv
sys_getdents
iovec
sys_swapon
alloc_file
sys_chown
sys_chmod
fd
notify_change
sys_newfstat
sys_fsync
fsnotify_change
fget
sys_mount
inode_setattr
sys_sync
sys_sysfs
virtual
Device Model
drivers/base/
driver_init
kobject
linux/kobject.h
bus_register
kset
linux/device.h
bus_type
device_create
device
class
device_type
driver_register
device_driver
probe
kvm
Virtual File System
file
vfs_fsync
file_systems
vfs_read
vfs_fstat
vfs_write
vfs_getattr
vfs_create
fsnotify
inode
inode_operations
file_operations
file_system_type
get_sb
super_block
ramfs_fs_type
bridges
cross-functional modules
page cache
address_space
bdi_writeback_thread
do_writepages
swap
si_swapinfo
swap_info
kswapd
do_swap_page
wakeup_kswapd
load_module
module
module_param
kernel_param
kobject_uevent_init
kobject_uevent
logical
functions implementations
init/
kernel/
system run
boot, shutdown
power management
init/main.c
start_kernel
do_initcalls
mount_root
run_init_process
kernel_restart
kernel_power_off
hibernate
machine_ops
logical file systems
ext4_file_operations
ext4_get_sb
ext4_readdir
drivers/
generic HW access
request_region
pci_driver
request_mem_region
pci_register_driver
pci_request_regions
usb_driver
usb_submit_urb
usb_hcd_giveback_urb
usb_hcd
ioremap
device control
block devices
block/
gendisk
block_device_operations
request_queue
init_scsi
scsi_device
scsi_driver
sd_fops
usb_storage_driver
usb_stor_host_template

Рис.1

На рис.1 показана часть интерактивной карты ядра Linux, в которой показаны уровни управления внешними устройствами. Даже при беглом взгляде видно, что это многоуровневая система, которая начинается с пользовательского интерфейса, содержащего системные вызовы и системные файлы, такие как виртуальная файловая система /proc, /sysfs, /dev. Затем идет виртуальный уровень: «Модель устройств» и «Виртуальная файловая система». Из карты ядра видно, что блочные устройства связаны с файловой системой. В позиции «Модель устройств» можно выделить три позиции: шину (bus), устройство (device) и драйвер устройства (device_driver). К позиции device_driver направлена стрелка от probe. Между виртуальным уровнем и логическим уровнем находятся, так называемые, мосты – кросс-функциональные модули. Логический уровень расшифровывается как реализация функций. На уровне «Управление устройствами» в позиции Система определяется общий доступ к оборудованию и в позиции Запоминающее устройство (storage) – блочные устройства. Блочные устройства рассматриваются по отношению к файловой системе, так как именно на блочных устройствах хранятся обычные файлы и выделяется область свопинга. Интерфейс аппаратных средств - драйверы, регистры и прерывания. В позиции «Доступ к устройству и драйвер шины» включены позиции, связанные с EHCI — Enhanced Host Controller Interface (Расширенный интерфейс хост-контроллера), который является улучшенной версией UHCI - Universal Host Controller Interface, который работает как PCI-устройство, но, в отличии от EHCI использует порты, а не MMIO (Memory-Mapped-IO). В позиции Драйверы контроллера диска указан интерфейс SCSI (англ. Small Computer System Interface, часто произносится как «скази»), который представляет собой набор стандартов для физического подключения и передачи данных между компьютерами и периферийными запоминающими устройствами. С течением времени этот интерфейс претерпел много изменений: старый 8-битный интерфейс SCSI в наши дни называется narrow SCSI, 16-разрядный интерфейс - wide SCSI, затем после удвоения частоты шины до 10MHz интерфейс стал называться fast SCSI и, наконец, после следующего удвоения частоты шины до 20MHz было введено название ultra SCSI (https://ru.wikipedia.org/wiki/SCSI). В настоящее время, SCSI предлагает очень быструю скорость передачи данных в 320 Мегабайт в секунду (Мб/сек), используя современный интерфейс Ultra320 SCSI. Кроме того, SCSI предлагает большой выбор возможностей, среди которых Command-Tag Queuing (метод оптимизации I/O команд для увеличения производительности). Жесткие диски SCSI отличаются надежностью; на коротком расстоянии можно создать последовательную цепь из 15 устройств, подключенную к каналу SCSI. Эти особенности делают SCSI замечательным выбором для производительных десктопов и рабочих станций, вплоть до серверов предприятий, по настоящее время.

Жесткие диски SAS используют набор команд SCSI и обладают схожей надежностью и производительностью, как и SCSI диски, однако используют последовательную версию интерфейса SCSI, со скоростью 300 Мб/сек. И хотя это немного медленнее, чем SCSI с 320 Мб/сек, интерфейс SAS способен поддерживать до 128 устройств на больших расстояниях, чем Ultra320, и может расширяться до 16000 устройств на канал. Жесткие диски SAS предлагают такую же надежность и скорости вращения (10000-15000), как и диски SCSI.

Диски SATA являются немного другими. Там, где SCSI и SAS диски уделяют внимание производительности и надежности, диски SATA жертвуют ими в пользу существенного увеличения емкости и снижения стоимости. К примеру, диск SATA в настоящий момент достиг емкости в 1 терабайт (ТБ). SATA используется там, где нужна максимальная емкость, например, для резервного копирования данных или архивирования. Сейчас SATA предлагает соединения точка-точка со скоростью до 300 Мб/сек, и легко опережает традиционный параллельный интерфейс ATA, со скоростью 150 Мб/сек.

Проблема с традиционным SCSI заключается в том, что просто его время заканчивается. Параллельный интерфейс SCSI, обладающий скоростью в 320 Мб/сек, не сможет работать значительно быстрее на существующих в настоящий момент длинах SCSI кабелей. Для сравнения, диски SATA достигнут скорости в 600 Мб/сек в ближайшем будущем, SAS достигнут 1200 Мб/сек. Диски SATA могут, кроме того, работать с интерфейсом SAS, таким образом эти диски могут использоваться одновременно в нескольких системах хранения. Потенциал SATA к увеличению производительности и расширяемости передачи данных намного превышает возможности, имеющиеся у SCSI. Но SCSI не уйдет со сцены в ближайшее время.

На самом нажнем уровне находится физические устройства, которые здесь обозначены как electronics.

Специальные файлы устройств

Специальные файлы устройств обеспечивают единообразный доступ к внешним устройствам. Эти файлы обеспечивают связь между файловой системой и драйверами устройств. В отличие от обычных файлов, специальные файлы устройств в действительности являются только указателями на соответствующие драйверы устройств в ядре. Такая интерпретация специальных файлов обеспечивает доступ к внешним устройствам как к обычным файлам. Так же как обычный файл, файл устройства может быть открыт, закрыт, в него можно писать или из него можно читать. По сравнению с обычными файлами файлы устройств имеют три дополнительных атрибута, которые характеризуют устройство, соответствующее данному файлу:

1. **Класс устройства**. В ОС Linux различают устройства блок-ориентированные и байт-ориентированные. Блок-ориентированные (или блочные) устройства, например, жесткий диск, передают данные блоками. Байт-ориентированные (или символьные) устройства, например, принтер и модем, передают данные посимвольно, как непрерывный поток байтов. Взаимодействие с блочными устройствами может осуществляться лишь через буферную память, а для символьных устройств буфер не требуется. Кроме этих двух классов устройств имеются еще два — небуферизованные байт-ориентированные устройства и именованные каналы (FIFO).
2. **Старший номер устройства**, обозначающий тип устройства, например, жесткий диск или звуковая плата. Текущий список старших номеров устройств можно найти в файле /usr/include/linux/major.h. Вот небольшая выдержка из этого списка

Таблица 9.1. *Старшие номера некоторых устройств*

Старший номер	**Тип устройства**
1	Оперативная память
2	Дисковод гибких дисков
3	Первый контроллер для жестких IDE-дисков
4	Терминалы
5	Терминалы
6	Принтер (параллельный разъем)
8	Жесткие SCSI-диски

14	Звуковые карты
22	Второй контроллер для жестких IDE-дисков

Файлы устройств одного типа имеют одинаковые имена и различаются по номеру, прибавляемому к имени. Например, все файлы сетевых плат Ethernet имеют имена, начинающиеся на **eth:** eth0, eth1 и т. д.

3. **Младший номер устройства** применяется для нумерации устройств одного типа, т. е. устройств с одинаковыми старшими номерами.

Если перейти каталог /dev и выполнить команду ls –l, то можно увидеть два числа, которые разделены запятой. Эти числа представляют собой старший и младший номер каждого из устройств. Ниже приведен типичный пример вывода команды **ls -l**. В этом списке, старшие номера устройств представлены числами 1, 4, 7, и 10, тогда как младшие – числами 1, 3, 5, 64, 65, и 129. Другими словами, символьные и блочные устройства представляются парой чисел: <major>:<minor>.

```
crw-rw-rw- 1 root   root    1, 3   Feb 23 1999  null
crw------- 1 root   root   10, 1   Feb 23 1999  psaux
crw------- 1 rubini tty     4, 1   Aug 16 22:22 tty1
crw-rw-rw- 1 root   dialout 4, 64  Jun 30 11:19 ttyS0
crw-rw-rw- 1 root   dialout 4, 65  Aug 16 00:00 ttyS1
crw------- 1 root   sys     7, 1   Feb 23 1999  vcs1
crw------- 1 root   sys     7, 129 Feb 23 1999  vcsa1
crw-rw-rw- 1 root   root    1, 5   Feb 23 1999  zero
```

Старший номер устройства определяет драйвер, связанный с устройством, т.е. это номер драйвера. Например, устройства /dev/null и /dev/zero используют драйвер с номером 1. Некоторые major номера зарезервированы для определенных устройств. Другие major номера динамически присваиваются драйверам устройств, когда Linux загружается. Например, major 94 всегда обозначает DAST (Direct Access Storage device). Номер 4 – для последовательных интерфейсов - виртуальные консоли и терминалы, 13 – для мышей, 14 – для аудиоустройств. Также, и vcs1, и vcsa1 управляются драйвером с номером 7.

В системе могут существовать псевдоустройства, это – узлы (файлы) устройств, которые не сопоставлены с физическим устройством:

- **/dev/random**, see **`random(4)`**
- **/dev/shm**
- **/dev/null**, **/dev/zero**, see **`null(4)`**
- **/dev/full**, see **`full(4)`**
- **/dev/ttyX**, where X is a number

Если использовать команду ls непосредственно для /dev/zero:

```
# ls -l /dev/zero
crw-rw-rw- 1 root root 1, 5  Nov 5 09:34 /dev/zero
```

Ядро использует старший номер устройства для диспетчеризации запроса на нужный драйвер.

Младший номер устройства используется самим драйвером, чтобы различать отдельные физические или логические устройства. Никакие другие части ядра не используют младший номер устройства.

Старшие номера, которые известны ядру устройств можно увидеть, выполнив команду:

[user]$ cat /proc/devices

Если необходимо подключить к системе какое-то новое устройство, вначале следует проверить, что в каталоге /dev имеется специальный файл (или ссылка на специальный файл) для этого устройства. Специальные файлы устройств создаются с помощью команды mknod (но, естественно, использовать команду mknod без необходимости и полного понимания последствий не стоит). Эта команда имеет следующий формат:

mknod [опции] <имя_устройства> <тип_устройства> <старший_номер> <младший_номер>

где параметр – «тип_устройства» может принимать одно из четырех значений:

- b — блок-ориентированное устройство;
- c — байт-ориентированное (символьное) устройство;
- u — не буферизованное байт-ориентированное устройство;
- p — именованный канал.

Для блок-ориентированных и байт-ориентированных устройств (b, c, u) нужны и старший и младший номера, для именованных каналов номера не используются. В следующем примере создается специальный файл для терминала, подключенного к порту COM3, который в Linux обозначается как /dev/ttyS2:

[root]# mknod -m 660 /dev/ttyS2 c 4 66

Как было сказано, устройства-терминалы представляют собой байт-ориентированные устройства, которые имеют старший номер 4 и младшие номера, которые начинаются с 64.

Внутреннее представление номеров устройств

Для хранения номеров устройств, как старшего так и младшего, в ядре используется тип **dev_t**, определённый в ***<linux/types.h>***.

```
typedef __kernel_dev_t dev_t;
```

Стандарт POSIX.1 определяет существование этого типа, но не оговаривает формат полей и их содержание. Начиная с версии ядра 2.6.0, **dev_t** является 32-х разрядным, 12 бит отведены для старшего номера и 20 - для младшего.

Для получения старшей или младшей части **dev_t** можно использовать макросы из ***<linux/kdev_t.h>***:

```
MAJOR(dev_t dev);
MINOR(dev_t dev); ine MINOR(dev) ((unsigned int) ((dev) & MINORMASK))
```

```
#ifndef _LINUX_KDEV_T_H
#define _ LINUX_KDEV_T_H

#include <uapi/linux/kdev_t.h>

#define MINORBITS 20
#define MINORMASK ((IU << MINORBITS) – 1)
```

#define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev) ((unsigned int) ((dev) & MINORMASK))

Если имеются старший и младший номера, то, наоборот, их необходимо преобразовать в **dev_t**, используя:

MKDEV(int major, int minor);

#define MKDEV(ma, mi) (((ma) << MINORBITS) | (mi))

В ядре Linux определены аналогичные функции:

```
#include <sys/sysmacros.h>

dev_t makedev(unsigned int maj, unsigned int min);

unsigned int major(dev_t dev);
unsigned int minor(dev_t dev);
```

Функция makedev() объединяет major и minor идентификаторы устройств для получения идентификатора устройства, возвращаемого как результат функции. Этот идентификатор устройства можно присвоить, например, mknod (2). Функции major() и minor() выполняют обратную задачу. Эти макросы могут быть полезны, например, для декомпозиции идентификаторов устройств в структуре, возвращаемой stat (2).

Следуя парадигме UNIX в UNIX все – файл, внешние устройства представляются в системе как специальные файлы и имеют inode. inode файла содержит метаданные о файле. Приложения обращаются к символьным и блочным устройствам через inode. Когда создается inode устройства, он сопоставляется с номерами major и minor.

В struct inode имеются соответствующие поля:

```
/*
 * Keep mostly read-only and often accessed (especially for
 * the RCU path lookup and 'stat' data) fields at the beginning
 * of the 'struct inode'
 */
struct inode {
        umode_t                 i_mode;
        …
        const struct inode_operations *i_op;   /* операции, определенные на inode*/
        struct super_block   *i_sb;
        struct address_space        *i_mapping;
        …
        /* Stat data, not accessed from path walking */
        unsigned long           i_ino;       /*номер inode*/
        …
        dev_t                   i_rdev;      /*фактический номер устройства, содержащий
major, minor*/
        …
        union {
                const struct file_operations     *i_fop;     /* former ->i_op->default_file_ops */
                void (*free_inode)(struct inode *);
        };
        …
        struct list_head        i_devices;
```

```
        union {
                struct pipe_inode_info          *i_pipe;
                struct block_device *i_bdev;
                struct cdev                     *i_cdev;
                char                            *i_link;
                unsigned                i_dir_seq;
        };
        …
} __randomize_layout;
```

Приложение может извлечь метаданные из inode, используя системный вызов stat(2) (или связанные вызовы), который возвращает структуру stat.

```
struct stat {
    dev_t     st_dev;       /* ID of device containing file */
    ino_t     st_ino;       /* Inode number */
    mode_t    st_mode;      /* File type and mode */
    nlink_t   st_nlink;     /* Number of hard links */
    uid_t     st_uid;       /* User ID of owner */
    gid_t     st_gid;       /* Group ID of owner */
    dev_t     st_rdev;      /* Device ID (if special file) */
    off_t     st_size;      /* Total size, in bytes */
    blksize_t st_blksize;   /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;    /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */

#define st_atime st_atim.tv_sec      /* Backward compatibility */
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
};
```

Поле st_dev - описывает устройство, на котором «живет» inode. Это устройство идентифицируется комбинацией его major идентификатором, котрый идентифицирует общий класс устройства, и minor идентификатором, который идентифицирует конкретный экземпляр в общем классе.

Поле st_mode – определяет тип файла и режим.

POSIX ссылается на биты stat.st_mode, соответствующие маске S_IFMT, в качестве типа файла, 12 битов, соответствующих маске 07777, в качестве битов режима файла и наименьших 9 бит (0777) в качестве битов разрешения файла.

Следующие значения маски определены для типа файла:

S_IFMT 0170000 bit mask for the file type bit field
S_IFSOCK 0140000 socket
S_IFLNK 0120000 symbolic link
S_IFREG 0100000 regular file
S_IFBLK 0060000 block device
S_IFDIR 0040000 directory
S_IFCHR 0020000 character device
S_IFIFO 0010000 FIFO

Thus, to test for a regular file (for example), one could write:

```
stat(pathname, &sb);
if ((sb.st_mode & S_IFMT) == S_IFREG) {
```

```
        /* Handle regular file */
    }
```

Поскольку тесты описанной выше формы являются общими, POSIX определяет дополнительные макросы, позволяющие писать тест типа файла в st_mode более кратко:

S_ISREG(m) is it a regular file?
S_ISDIR(m) directory?
S_ISCHR(m) character device?
S_ISBLK(m) block device?
S_ISFIFO(m) FIFO (named pipe)?
S_ISLNK(m) symbolic link? (Not in POSIX.1-1996.)
S_ISSOCK(m) socket? (Not in POSIX.1-1996.)

Таким образом, предыдущий фрагмент кода можно переписать так:

```
stat(pathname, &sb);
if (S_ISREG(sb.st_mode)) {
    /* Handle regular file */
}
```

Поле st_rdev – устройство, представленное данным inode. Если этот файл, имеющий inode, представляет устройство, тогда inode содержит major и minor идентификаторы этого устройства.
Макросы major(3) и minor(3) могут быть полезны для разложения идентификатора устройства в этом поле.

devfs

http://rus-linux.net/MyLDP/file-sys/holm/l-fs4_ru.html

В то время как сама идея отображения устройств как специальных файлов хороша, следует заметить, что обычные Linux системы управляют ими далеко не оптимальным и громоздким способом. В наши дни в Linux поддерживается много самого разного hardware. При "классическом" подходе это означает, что в каталоге /dev "обитают" сотни специальных файлов для "презентации" соответствующих hardware. Большинство таких специальных файлов "don't even map" на реально существующее на вашей машине устройство (но в каталоге /dev такой специальный файл все равно должен присутствовать на случай добавления нового hardware/drivers). Такой подход вносит много путаницы.

Только одного такого аргумента достаточно, чтобы осознать необходимость "перестройки" каталога /dev, конечно, при соблюдении принципа "обратной совместимости". Чтобы хорошо понять, как именно devfs решает большинство проблем связанных с "классическим" каталогом /dev, посмотрим на devfs с позиции разработки нового драйвера для устройства.

Традиционный (без devfs) kernel-based device driver "прописывает" устройство в остальной части системы через системные вызовы register_blkdev() или register_chrdev() (зависит от того, регистрируется блочное или символьное устройство).

Major номер (unsigned 8-bit integer) передается как параметр либо register_blkdev(), либо register_chrdev(). После регистрации устройства ядро знает, что этот конкретный major номер соответствует конкретному драйверу для устройства, которое выполнило вызов register_???dev().

Вопрос, а какой major номер разработчик драйвера должен использовать для передачи с запросом register_???dev()? Проблемы нет, если developer драйвера не планирует его использования "внешним миром". В этом случае сгодится любой major номер, лишь бы он

не конфликтовал с другими major номерами, используемыми в конкретном частном случае. Как альтернатива, разработчик может "динамически" ассигновать major номер перед register_???dev(). Однако, "по большому счету", такое решение приемлемо только в случае, если драйвер не предназначен для широкого использования.

Если developer хочет предложить свой драйвер для широкого использования (а большинство Linux developers имеет тенденцию делать именно так), то использование major номера "от балды" или даже вариант с его динамическим ассигнованием "не пройдет". В таких случаях разработчик драйвера должен войти в контакт с Linux kernel developers и получить для своего специфического устройства "официальный" major номер. После этого, для всех Linux пользователей это устройство (и только оно) будет связано с таким major номером.

Важно иметь "официальный" major номер, так как для взаимодействия с этим специфическим устройством, администратор должен в каталоге /dev создать специальный файл. Когда device node (специальный файл) создается, он должен получить тот же major, как зарегистрирован во внутренних структурах ядра. После этого, когда пользовательский процесс выполняет операцию над файлом-устройством, ядро знает, с каким драйвером нужно связаться. Иначе, mapping от специального файла на kernel driver сделано по major номером, а не по именам устройств. Такое "непонимание" device name - особенность non-devfs систем.

Как только device driver получает официальный major, он может использоваться публично, а device node можно включать в дерево /dev разных дистрибутивов через официальный сценарий /dev/MAKEDEV. Такой сценарий помогает суперпользователю автоматизировать процесс создания device nodes с правильными major и minor номерами, правами и владельцами.

К сожалению, при таком подходе имеется много проблем. У разработчика драйвера появляется "головная боль" от необходимости согласования действий с kernel developers для получения "официального" major. То же относится к самим kernel developers. Возникает потребность отслеживания процедуры ассигнования всех major номеров. Во многом это похоже на проблемы системного администратора при использовании статического ассигнования IP адресов в локальной сети, когда сеть начинает "разрастаться". Точно так же, как системный администратор может "разрубить узел", воспользовавшись DHCP, можно было бы использовать подобный подход для регистрации devices.

Проблема не только в этом. Linux подошел к границе, когда все "официальные" major и minor номера будут исчерпаны. Конечно, можно просто увеличить разрядность номеров, но при этом станет еще сложнее отслеживать уникальность major для драйверов. Имеется и более радикальный способ решения проблемы. Это переход на devfs.

После правильного конфигурирования devfs, что подразумевает добавление поддержки devfs в ядро и выполнение множества достаточно хитрых изменений в сценариях запуска, суперпользователь перезагружает систему. Стартует ядро и device drivers начинают регистрировать свои устройства для остальной части системы. Если это non-devfs система, как и ранее, выполняются системные вызовы register_blkdev() и register_chrdev() (вместе с сопровождающими вызовы major номерами). Однако, если enabled devfs, то device drivers для регистрации своих устройств используют новый, улучшенный kernel call, называемый devfs_register(). Хотя можно указать major и minor номера для обратной совместимости, жесткого требования, делать именно так, не существует. Вместо этого вызов devfs_register() передает path на устройство как параметр, и именно так оно впоследствии появиться под /dev.

Например, драйвер устройства foo регистрирует свое устройство в devfs. При этом драйвер передает параметр *foo0* с вызовом devfs_register(), сообщая ядру, что в корне devfs

namespace должен быть создан новый файл-устройство *foo0*. В ответ на вызов devfs_register() добавляется foo0 device node к корню devfs namespace и запись о том, что этот новый foo0 node должен отобразится на foo device driver в ядре.

Пример получения и освобождения номеров устройств для блочных устройств

Рассмотрим получение и освобождение номера блочного устройства на следующем примере:

```
static int __init sblkdev_init(void)
 {
     int ret = SUCCESS;
     _sblkdev_major = register_blkdev(_sblkdev_major, _sblkdev_name);
   if (_sblkdev_major <= 0)
   {
       printk(KERN_WARNING "sblkdev: unable to get major number\n");
       return -EBUSY;
   }
    ret = sblkdev_add_device();
    if (ret) unregister_blkdev(_sblkdev_major, _sblkdev_name);
     return ret;
}
 static void __exit sblkdev_exit(void)
 {
     sblkdev_remove_device();
     if (_sblkdev_major > 0) unregister_blkdev(_sblkdev_major, _sblkdev_name);
}
module_init(sblkdev_init);
module_exit(sblkdev_exit);
```

При загрузке модуля выполняется функция sblkdev_init(), при выгрузке – функция sblkdev_exit().
Функция register_blkdev() регистрирует блочное устройство. Ему выделяется **major** номер.

int register_blkdev(unsigned int *major*, **const char *** *name*);

major - запрашиваемый основной номер устройства [1..255]. Если *major = 0, то* выделяется какой-либо неиспользованный номер major.

name - имя нового блочного устройства в виде строки с нулевым символом в конце. В качестве name в этот вызов передаётся родовое имя класса устройств, например, для дисков **xda**, **xdb**, ... , создаваемых в примере ниже, это будет "**xd**".

Регистрация имени устройства создаёт соответствующую запись в файле **/proc/devices**, но не создаёт самого устройства в **/dev**:

```
1    $ cat /proc/devices | grep xd
2    252 xd
```

Имя должно быть уникальным в системе. Возвращаемое значение зависит от основного входного параметра. Если *major* номер устройства был запрошен в диапазоне [1..255], то функция возвращает ноль при успешном завершении или отрицательный код ошибки. Если был запрошен неиспользуемый основной номер с **параметром major = 0**, тогда возвращаемое значение является назначенным *major* в диапазоне [1..255], иначе будет возвращен отрицательный код ошибки.

Функция unregister_blkdev() — освобождает этот номер.

В данном контексте функция sblkdev_add_device() интересна тем, что в ней определяются поля структуры struct gendisk, в которой имеются **поля major, first_minor, minor**:

```
struct gendisk {
        /* major, first_minor and minors are input parameters only,
         * don't use directly.  Use disk_devt() and disk_max_parts().
         */
        int major;        /* major number of driver */
        int first_minor;
        int minors;   /* maximum number of minors, =1 for
                       * disks that can't be partitioned. */
        char disk_name[DISK_NAME_LEN]; /* name of major driver
*/
        char *(*devnode)(struct gendisk *gd, umode_t *mode);
        unsigned int events;            /* supported events */
        unsigned int async_events;      /* async events, subset of all */
        /* Array of pointers to partitions indexed by partno.
         * Protected with matching bdev lock but stat and other
         * non-critical accesses use RCU.  Always access through
         * helpers.
         */
        struct disk_part_tbl __rcu *part_tbl;
        struct hd_struct part0;
        const struct block_device_operations *fops;
        struct request_queue *queue;
        void *private_data;
        int flags;
        struct rw_semaphore lookup_sem;
        struct kobject *slave_dir;
        struct timer_rand_state *random;
        atomic_t sync_io;               /* RAID */
        struct disk_events *ev;
#ifdef  CONFIG_BLK_DEV_INTEGRITY
        struct kobject integrity_kobj;
#endif  /* CONFIG_BLK_DEV_INTEGRITY */
        int node_id;  /* номер inode */
        struct badblocks *bb;
        struct lockdep_map lockdep_map;
};
```

Функция:

static int sblkdev_add_device(void)

```
{
   int ret = SUCCESS;
        …
         struct gendisk *disk = alloc_disk(1); //only one partition
        …
         disk->flags |= GENHD_FL_NO_PART_SCAN; //only one partition
         disk->flags |= GENHD_FL_REMOVABLE;
         disk->major = _sblkdev_major;
         disk->first_minor = 0;
        …
}
```

Получение и освобождение номеров устройств для символьных устройств

Аналогично блочным устройствам для символьных устройств существует функция register_chrdevice():

#include <linux/fs.h>

int register_chrdev(unsigned int *major*, **const char***name*, **struct file_operations***ops*);
int unregister_chrdev(unsigned int *major*, **const char ***name*);

Функция register_chrdev() связывает major номер символьного устройства с набором точек входа драйвера. Структура file_operations содержит указатели на функции, которые драйвер использует для реализации интерфейса ядра с драйвером.

Параметр *major* - это *major* номер, назначаемый драйверу символьного устройства и сопоставляемый с таблицей функций. Параметр name представляет собой краткое имя устройства и отображается в списке /proc/devices. Он также должен точно соответствовать имени, переданному функции unregister_chrdev() при освобождении функций. Модуль драйвера устройства может зарегистрировать столько разных основных номеров, сколько он поддерживает, хотя обычно это не делается. Функция unregister_chrdev() освобождает старший номер и обычно вызывается в функции module_cleanup для удаления драйвера из ядра.

В случае успеха, register_chrdev возвращает 0, если major - это число, отличное от 0. В противном случае (если major = 0) Linux выберет старший номер и вернет выбранное значение.

В случае ошибки возвращается один из следующих кодов:

- -EINVAL
- Указанный номер недействителен (> MAX_CHRDEV)
- -EBUSY
- Основной номер занят

Функция unregister_chrdev() вернет 0 в случае успеха или -EINVAL, если основной номер не зарегистрирован с соответствующим именем.

Существует набор функций, которые решают эту задачу. Эти функции приведены в приложении 1. Например, функция register_chrdev_region() регистрирует диапазон номеров устройств.

Одним из первых шагов, который необходимо сделать разрабатываемому драйверу при установке символьного устройства, является получение одного или нескольких номеров устройств для работы с ними. Одной из функций для выполнения этой задачи является ***register_chrdev_region***, которая объявлена в ***<linux/fs.h>***:

int register_chrdev_region(dev_t first, unsigned int count, char *name);

Здесь **first** это - начало диапазона номеров устройств, который вы хотели бы выделить. Младшее число **first** часто 0, но не существует никаких требований на этот счёт. **count** - запрашиваемое общее число смежных номеров устройств. Заметим, что если число **count** большое, запрашиваемый диапазон может перекинуться на следующей старший номер, но всё будет работать правильно, если запрашиваемый диапазон чисел доступен. Наконец, **name** - имя устройства, которое должно быть связано с этим диапазоном чисел; оно будет отображаться в ***/proc/devices*** и sysfs.

Функция ***register_chrdev_region*** хорошо работает, если заранее известно, какие именно номера устройств будут использоваться. Однако, часто не известно, какие старшие номера устройств будут использоваться. Поэтому сообщество разработчиков ядра Linux прилагает постоянные усилия, чтобы перейти к использованию динамически выделяемых номеров устройств. Ядро будет выделить старший номер "на лету", но для этого необходимо запрашивать другую функцию:

int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *nam e);

В этой функции **dev** является только выходным значением, которое при успешном завершении содержит первый номер выделенного диапазона. **firstminor** должен иметь значение первого младшего номера для использования; как правило, 0.
Параметры **count** и **name** аналогичны ***register_chrdev_region***.

Например:

```
de <linux/kdev_t.h>
de <linux/fs.h>
de <linux/cdev.h>
de <linux/device.h>
…
dev_t dev = 0;
static int __init etx_driver_init(void)
{
        /*Allocating Major number*/
        if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
                printk(KERN_INFO "Cannot allocate major number\n");
                return -1;
        }
        printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

        /*Creating cdev structure*/
```

```
        cdev_init(&etx_cdev,&fops);
        etx_cdev.owner = THIS_MODULE;
        etx_cdev.ops = &fops;

        /*Adding character device to the system*/
        if((cdev_add(&etx_cdev,dev,1)) < 0){
             unregister_chrdev_region(dev,1);
             return -1;
           }
       …
   }
```

Независимо от того, как были назначены номера устройств, нужно их освободить, если они больше не используются. Номера устройств освобождаются функцией:

void unregister_chrdev_region(dev_t first, unsigned int count);

Обычное место для вызова ***unregister_chrdev_region*** будет в функции module_cleanup или exit загружаемого модуля ядра. Например:

```
 void __exit etx_driver_exit(void)
 {
        …
        unregister_chrdev_region(dev, 1);
           printk(KERN_INFO "Device Driver Remove...Done!!!\n");
 }
```

Приведённые функции выделяют номера устройств для использования драйвером, но ничего не говорят ядру, как в действительности эти номера будут использоваться. Перед тем, как какая-либо программа из пространства пользователя сможет получить доступ к одному из этих номеров устройств, драйверу необходимо подключить их к своим внутренним функциям, которые осуществляют операции, связанные с устройством.

Внешние устройства и драйверы устройств

Устройства

На самом низком уровне каждое устройство в Linux представлено экземпляром **struct device**. Эта структура содержит информацию, которая необходима ядру для представления устройства. Но ядро содержит набор подсистем таких как pci, pci express, usb. Большинство подсистем отслеживают дополнительную информацию об устройствах, которые к ним относятся. В результате устройства редко представляются структурой struct device. Эта структура входит в состав таких структур, как struct pci_device или struct usb_device, которые рассмотрены ниже.

```
struct device {
        struct kobject kobj;
        …
        const char          *init_name; /* initial name of the device< - > исходное название устройства */
        const struct device_type *type;
        struct bus_type     *bus;           /* type of bus device is on */
        struct device_driver *driver;  /* which driver has allocated this
```

```
                                                    device< - > драйвер, относящийся к
устройству */
        void            *platform_data;     /* Platform specific data, device
                                               core doesn't touch it */
        void            *driver_data;       /* Driver data, set and get with
         …
#ifdef CONFIG_GENERIC_MSI_IRQ_DOMAIN
        struct irq_domain   *msi_domain;
#endif
#ifdef CONFIG_PINCTRL
        struct dev_pin_info *pins;
#endif
#ifdef CONFIG_GENERIC_MSI_IRQ
        struct list_head    msi_list;
#endif
        const struct dma_map_ops *dma_ops;
        u64             *dma_mask;          /* dma mask (if dma'able device) */
        ...
#ifdef CONFIG_NUMA
        int             numa_node;          /* NUMA node this device is close to */
#endif
        dev_t                       devt;   /* dev_t, creates the sysfs "dev" */
        u32                         id;     /* device instance */
             ...
};
```

Устройства, подключаемые к шине pci представляются структурой struct **pci_dev,** которая содержит строку **struct device dev.** Данная строка представляет универсальный интерфейс устройства в системе.

```
/* The pci_dev structure describes PCI devices */
struct pci_dev {
        struct list_head bus_list;      /* Node in per-bus list */
        struct pci_bus      *bus;       /* Шина, на которой находится устройство */
        struct pci_bus      *subordinate;   /* Bus this device bridges to */
         …
        unsigned short      vendor; /*поставщик*/
        unsigned short      device;  /*устройство*/
        unsigned short      subsystem_vendor;
        unsigned short      subsystem_device;
        unsigned int        class;      /* 3 bytes: (base,sub,prog-if) */
        ...
        u8                  pin;        /* Interrupt pin this device uses */
        ...
        struct pci_driver *driver;  /* Драйвер, связанный с конкретным устройством
*/
        ...
        struct device_dma_parameters dma_parms;
         …
        struct device       dev;        /* универсальный интерфейс устройства */
         ...
        /*
         * Instead of touching interrupt line and base address registers
         * directly, use the values stored here. They might be different!
```

```
*Вместо непосредственного использования линии прерывания и регистров базового адреса используйте значения,
* хранящиеся здесь. Они могут быть разными!
         */
        unsigned int        irq;
        struct resource resource[DEVICE_COUNT_RESOURCE]; /* I/O and memory regions + expansion ROMs */
        ...
        pci_dev_flags_t dev_flags;
        ...
        unsigned long       priv_flags;         /* Private flags for the PCI driver */
};
```

Шины PCI и PCI Express

PCI (англ. *Peripheral component interconnect* - взаимосвязь периферийных компонентов) — шина ввода-вывода для подключения периферийных устройств к материнской плате компьютера.

Стандарт на шину PCI определяет:

- физические параметры (например, разъёмы и разводку сигнальных линий);
- электрические параметры (например, напряжения);
- логическую модель (например, типы циклов шины, адресацию на шине).

В 2002-х году появилась первая базовая спецификация PCI Express и был осуществлен переход со стандарта PCI на PCI Express (англ. *Peripheral Component Interconnect Express*, или **PCIe**, или **PCI-e,** также известная как **3GIO** for *3rd Generation I/O*). У стандарта PCIe имеется одно важное преимущество: вместо последовательной шины, которой является шина PCI, стала использоваться двухточечная система доступа. В отличие от стандарта PCI, использовавшего для передачи данных общую шину с подключением параллельно нескольких устройств, PCI Express, если рассматривать в общем, является пакетной сетью с топологией типа звезда. В настоящее время устройства интегрируются к материнским платам компьютера, а иногда и к другому компьютеру через разъемы типа PCIe (PCI Express).

На материнской плате компьютера доступно в основном два разных типа слотов расширения:

- для обеспечения обратной совместимости настольные компьютеры по-прежнему оснащены слотами PCI;
- слоты PCI Express, которые доступны в четырех разных размерах.

Порт PCI Express - это логическая структура моста PCI-PCI. Существует два типа портов PCI Express: корневой порт и порт коммутатора. Корневой порт образует канал PCI Express от корневого комплекса PCI Express, а порт коммутатора соединяет каналы PCI Express с внутренними логическими шинами PCI. Порт коммутатора, у которого есть его вторичная шина, представляющая логику внутренней маршрутизации коммутатора, называется восходящим портом коммутатора. Нисходящий порт коммутатора соединяет внутреннюю шину маршрутизации коммутатора с шиной, представляющей нисходящий канал PCI Express от коммутатора PCI Express.

В существующих ядрах Linux **модель драйвера устройства** Linux позволяет обрабатывать физическое устройство только одним драйвером. Порт PCI Express - это мостовое устройство PCI-PCI с несколькими различными службами. Чтобы поддерживать чистое и простое решение, каждая служба может иметь свой собственный драйвер службы программного обеспечения. В этом случае несколько сервисных драйверов будут конкурировать за одно устройство моста PCI-PCI. Например, если драйвер службы горячего подключения собственного корневого порта PCI Express загружается первым, он запрашивает корневой порт моста PCI-PCI. Поэтому ядро не загружает другие служебные драйверы для этого корневого порта. Другими словами, невозможно одновременно загружать и запускать несколько сервисных драйверов на мостовом устройстве PCI-PCI, используя текущую модель драйвера.

Для включения одновременной работы нескольких служебных драйверов необходим *драйвер шины PCI Express Port Bus*, который управляет всеми заполненными портами PCI Express и по мере необходимости распределяет все предоставленные сервисные запросы в соответствующие служебные драйверы. Некоторые ключевые преимущества использования драйвера шины PCI Express Port перечислены ниже:

• Разрешить одновременную работу нескольких сервисных драйверов на устройстве с мостовым портом PCI-PCI.

• Разрешить использование сервисных драйверов в независимом поэтапном подходе.

• Разрешить запуск одного служебного драйвера на нескольких устройствах мостового порта PCI-PCI.

• Управлять ресурсами распределительного моста PCI-PCI и распределять их по требуемым служебным драйверам.

Драйверы устройств PCI реализованы на основе модели драйверов устройств Linux. Все служебные драйверы **являются драйверами устройств PCI**. Как обсуждалось выше, невозможно загрузить какой-либо служебный драйвер после того, как ядро загрузит драйвер шины PCI Express Port. Для соответствия модели драйвера **шины PCI Express Port Port** требуются некоторые минимальные изменения в существующих служебных драйверах, которые не влияют на функциональность существующих служебных драйверов. Драйвер службы должен использовать два API, показанных ниже, для регистрации своей службы с помощью **драйвера шины PCI Express Port**. Важно, чтобы служебный драйвер инициализировал структуру данных pcie_port_service_driver, включенную в заголовочный файл /include/linux/pcieport_if.h, перед вызовом этих API. Невыполнение этого требования приведет к несоответствию идентификатора, что не позволит драйверу шины PCI Express загрузить драйвер службы.

Определены специальные API. Например,

int pcie_port_service_register(struct pcie_port_service_driver *new) заменяет API Linux Driver Model’s pci_register_device. A int pcie_port_service_unregister(struct pcie_port_service_driver *new) заменяет pci_unregister_device.

Шина USB

USB (Universal Serial Bus — универсальная последовательная шина) является промышленным стандартом расширения архитектуры PC, ориентированным на интеграцию с телефонией и устройствами бытовой электроники. Версия 1.0 была опубликована в январе 1996 года. С середины 1996 года выпускаются PC со встроенным контроллером USB, реализуемым чипсетом. Ожидается появление модемов, клавиатур, сканеров, динамиков и других устройств ввода/вывода с поддержкой USB, а также мониторов с USB-адаптерами - они будут играть роль хабов для подключения других устройств.

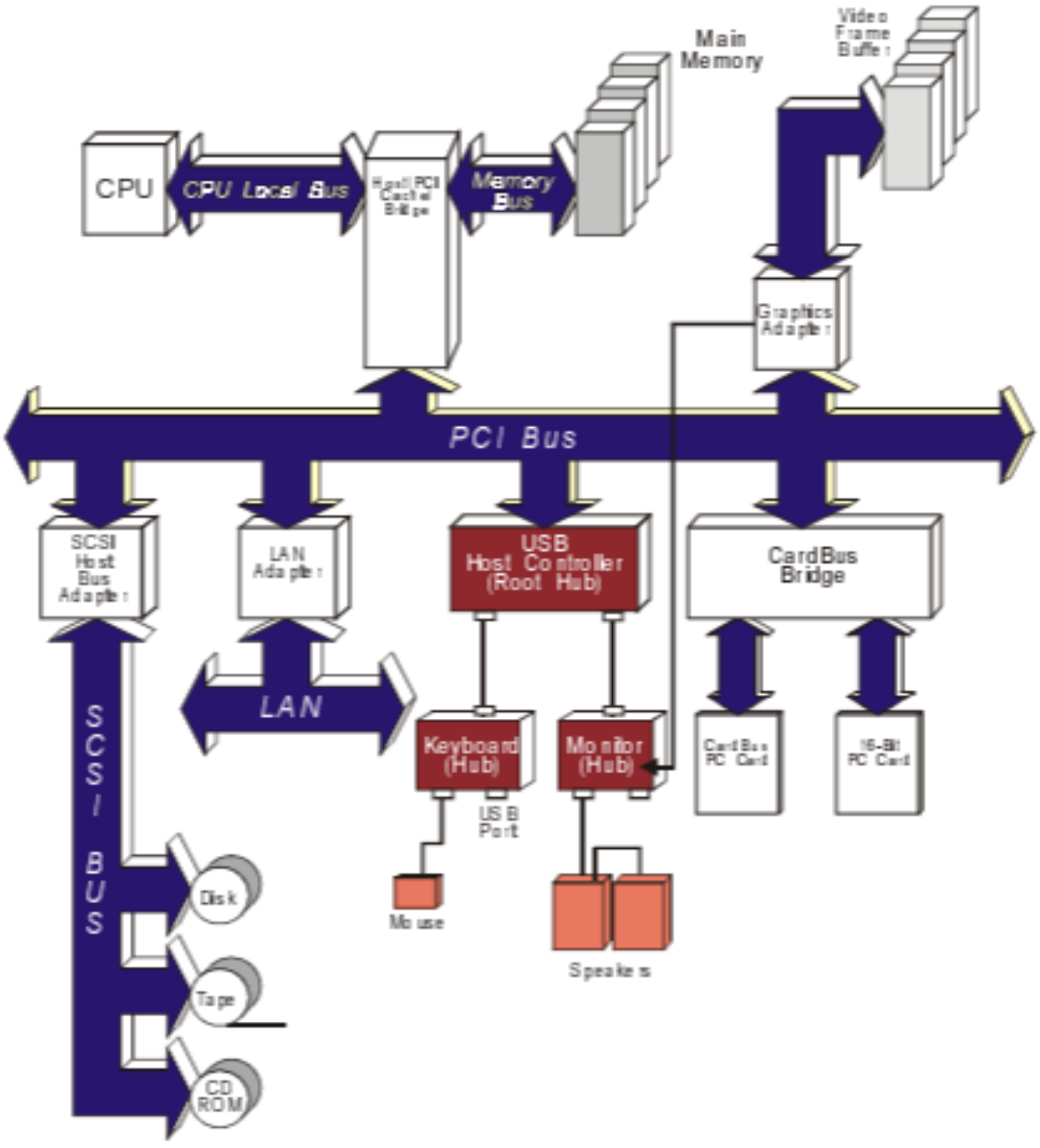

Рис.2 Система USB реализована на базовой платформе PCI

Данная иллюстрация показывает шину USB, реализованную в системе, построенной на шине PCI.

http://www.usb.org.

Устройства (Device) USB могут являться хабами, функциями или их комбинацией. *Хаб* (Hub) обеспечивает дополнительные точки подключения устройств к шине. *Функции (Function)* USB предоставляют системе дополнительные возможности, например подключение к ISDN, цифровой джойстик, акустические колонки с цифровым интерфейсом и т. п. Устройство USB должно иметь интерфейс USB, обеспечивающий полную поддержку протокола USB, выполнение стандартных операций (конфигурирование и сброс) и предоставление информации, описывающей устройство. Многие устройства, подключаемые к USB, имеют в своем составе и хаб, и функции. Работой всей системы USB управляет *хост-контроллер (Host Controller),* являющийся программно-аппаратной подсистемой хост-компьютера.

Физическое соединение устройств осуществляется по топологии многоярусной *звезды.* Центром каждой звезды является *хаб,* каждый кабельный сегмент соединяет две точки - хаб с другим хабом или с функцией. В системе имеется один (и только один) *хост-контроллер,* расположенный в вершине пирамиды устройств и хабов. Хост-контроллер интегрируется с *корневым хабом (Root Hub),* обеспечивающим одну или несколько точек подключения - *портов.* Контроллер USB, входящий в состав чипсетов, обычно имеет встроенный двухпортовый хаб. Логически устройство, подключенное к любому хабу USB и сконфигурированное (см. ниже), может рассматриваться как непосредственно подключенное к хост-контроллеру.

Функции представляют собой устройства, способные передавать или принимать данные или управляющую информацию по шине. Функции представляют собой отдельные ПУ с кабелем, подключаемым к порту хаба. Физически в одном корпусе может быть несколько функций со встроенным хабом, обеспечивающим их подключение к одному порту. Эти комбинированные устройства для хоста являются хабами с постоянно подключенными устройствами-функциями.

Каждая функция предоставляет конфигурационную информацию, описывающую возможности ПУ и требования к ресурсам. Перед использованием функция должна быть сконфигурирована хостом - ей должна быть выделена полоса в канале и выбраны опции конфигурации [5].

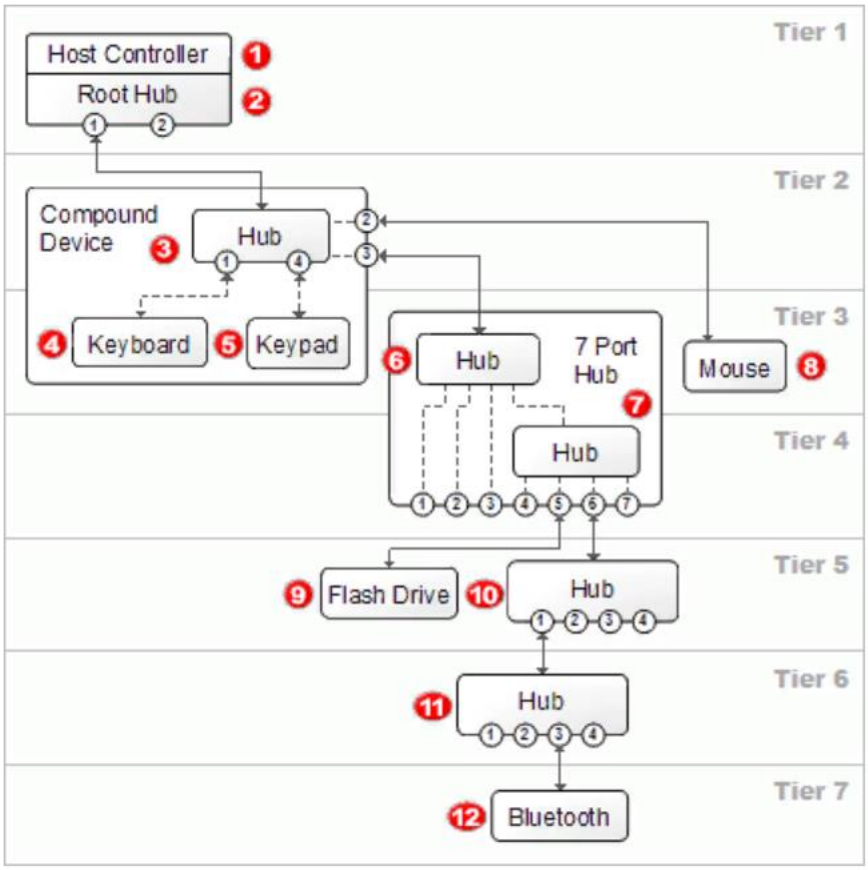

Рис.3 Пример топологии USB с точки зрения хоста

Топология шины - это модель соединения между хостом и периферийными устройствами USB. Архитектура USB имеет четко определенную физическую и логическую топологию шины, которая полностью описана в спецификации USB. На рисунке 3 показаны 7 уровней топологии:

1. USB host with host controller
2. 2-port root hub integrated into the host controller
3. 4-port hub integrated into the keyboard (part of the compound device)
4. USB keyboard (part of the compound device)
5. USB keypad (part of the compound device)
6. 4-port hub (part of the 7-port hub)
7. 4-port hub (part of the 7-port hub)
8. USB mouse
9. USB flash drive
10. 4-port hub
11. 4-port hub
12. USB bluetooth adapter

Можно сказать, что на рис.3 показано USB-дерево или, так называемая, многоярусная звезда (tiered star). Host контроллер это – корневой узел USB дерева. Hub – концентратор или повторитель имеет одно соединение, которое называется upstream port, к верхнему уровню USB дерева и некоторое количество портов для подключения внешних устройств или других хабов. Хабы являются активными электронными устройствами.

Аналогично, struct pci_dev структура struct usb_dev содержит struct device dev.

Эти структуры объявлены в системе для более детального представления специфических устройств.

В ОС Linux USB устройства описываются следующей структурой:

```
/* USB_DT_DEVICE: Device descriptor – вспомогательная структура */
struct usb_device_descriptor {
        __u8  bLength;
        __u8  bDescriptorType;

        __le16 bcdUSB;
        __u8  bDeviceClass;
        __u8  bDeviceSubClass;
        __u8  bDeviceProtocol;
        __u8  bMaxPacketSize0;
        __le16 idVendor;
        __le16 idProduct;
        __le16 bcdDevice;
        __u8  iManufacturer;
        __u8  iProduct;
        __u8  iSerialNumber;
        __u8  bNumConfigurations;
} __attribute__ ((packed));

struct usb_device {
        int                     devnum;
        char                    devpath[16];
        u32                     route;
        enum usb_device_state       state;  /*Состояние устройства: настроено, не
подключено и т. д.*/
        enum usb_device_speed       speed; /*Скорость устройства: высокая / полная /
низкая (или ошибка)*/
        ...
        struct usb_device *parent;
        struct usb_bus *bus;   /*шина, частью которого мы являемся*/
        struct usb_host_endpoint ep0; /*данные конечной точки 0 (канал управления по
умолчанию)*/

        struct device dev;   /* универсальный интерфейс устройства */
        struct usb_device_descriptor descriptor;
        struct usb_host_bos *bos;
        struct usb_host_config *config;

        struct usb_host_config *actconfig;
        struct usb_host_endpoint *ep_in[16]; /*массив конечных точек IN*/
        struct usb_host_endpoint *ep_out[16]; /*массив конечных точек OUT*/
        …
        u8 portnum; /*номер родительского порта (источник 1)*/
        u8 level;   /*количество предков USB-концентраторов*/
        u8 devaddr;
        unsigned can_submit:1;        /*URB могут быть представлены*/
        …
        int string_langid;              /*идентификатор языка для строк*/
        /* static strings from the device */
        char *product;                  /*идентификатор продукта, если есть (статический)*/
```

```
        char *manufacturer;         /*Строка i-производителя, если имеется
(статическая)*/
        char *serial;               /* серийный номер*/
        struct list_head filelist;
        int maxchild;               /*количество портов в хабе*/
        u32 quirks;
        atomic_t urbnum;            /*количество URB, представленных для всего
устройства*/
        unsigned long active_duration;
#ifdef CONFIG_PM
        unsigned long connect_time; /*время, когда устройство было впервые подключено/*
        ...
        unsigned port_is_suspended:1;
#endif
        struct wusb_dev *wusb_dev;
        int slot_id;
        enum usb_device_removable removable;
        ...
}
```

Структуры для символьных и блочных устройств

Традиционно с операционных системах различается два типа устройств: символьные и блочные. Для их описания существуют **дополнительные структуры**, детализирующие особенности типов устройств, но на эти структуры ссылается **inode**:

```
struct inode {
        …
        const struct inode_operations       *i_op;
        struct super_block    *i_sb;
        struct address_space *i_mapping;
        …
        dev_t                 i_rdev;
        …
        union {
                const struct file_operations       *i_fop;      /*
former ->i_op->default_file_ops */
                void (*free_inode)(struct inode *);
        };
        …
        struct list_head      i_devices;
        union {
                struct pipe_inode_info      *i_pipe;
                struct block_device    *i_bdev;
                struct cdev            *i_cdev;
                char                   *i_link;
                unsigned               i_dir_seq;
        };

        …
}  __randomize_layout;
```

То есть, структуры block_device и cdev определены в файловой системе Linux.

Как уже было сказано**, struct cdev** является одним из элементов структуры inode, которая используется ядром для представления файлов. Структура cdev - это внутренняя структура ядра, которая представляет символьные устройства.

```
struct cdev {
        struct kobject kobj;
        struct module *owner;
        const struct file_operations *ops;
        struct list_head list;
        dev_t dev;
        unsigned int count;
} __randomize_layout;
```

При создании *символьного устройства* в данной структуре нужно заполнить только два поля:

1. **file_operation** (Эта структура является одним из полей структуры cdev)
2. **owner** (Обязательное значение - THIS_MODULE)

Существует два способа выделения и инициализации структуры:

1) Runtime Allocation

2) Own allocation.

Если нужно получить автономную структуру cdev во время выполнения, то это можно сделать с помощью следующего кода:

```
struct cdev *my_cdev = cdev_alloc( );
my_cdev->ops = &my_fops;
```

Или же можно встроить структуру cdev в собственную структуру устройства, используя следующую функцию:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

Как только структура cdev настроена с file_operations и owner, последний шаг - сообщить ядру об этом с помощью вызова:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

Где: dev - структура cdev, num - номер первого устройства, на которое отвечает это устройство, count - количество номеров устройств, которые должны быть связаны с устройством. Часто счет равен единице, но есть ситуации, когда имеет смысл иметь более одного номера устройства, соответствующего конкретному устройству. Если эта функция возвращает отрицательный код ошибки, то устройство не было добавлено в систему.

После вызова **cdev_add ()** устройство сразу же активируется. Все функции, которые были определены (через структуру file_operations), могут быть вызваны.

Чтобы удалить устройство char из системы, нужно вызвать функцию: **void cdev_del(struct cdev *dev).**

Драйвер блочного устройства обеспечивает доступ к устройствам, которые передают произвольно доступные данные блоками фиксированного размера, в первую очередь на дисках. Очевидно, что блочные устройства принципиально отличаются от символьных устройств, и эти особенности не могу не учитываться в ядре. В результате драйверы блочных устройств имеют соответствующий интерфейс, который определяется специфическими задачами и проблемами блочных устройств.

Эффективные блочные драйверы имеют решающее значение для производительности системы. Современные системы с виртуальной памятью работают путем перемещения данных во вторичную память, которая обычно представляет собой жесткий диск. Можно сказать, что драйверы блочных устройств - это канал между памятью ядра и вторичным хранилищем; следовательно, они могут рассматриваться как составляющие подсистемы виртуальной памяти.

Большая часть дизайна блочного слоя сосредоточена на производительности.

Аналогично struct cdev структура struct block_device является одним из элементов структуры inode, которая используется ядром для представления файлов. Структура block_device - это внутренняя структура ядра, которая представляет блочные устройства. Это поле содержит указатель на эту структуру, когда inode ссылается на файл устройства типа block_device.

```
struct block_device {
        dev_t                   bd_dev;  /* not a kdev_t - it's a search key */
        int                     bd_openers;
        struct inode *          bd_inode;       /* will die */
        struct super_block *    bd_super;
        struct mutex            bd_mutex;       /* open/close mutex */
        void *                  bd_claiming;
        void *                  bd_holder;
        int                     bd_holders;
        bool                    bd_write_holder;
#ifdef CONFIG_SYSFS
        struct list_head        bd_holder_disks;
#endif
        struct block_device *   bd_contains;
        unsigned                bd_block_size;
        u8                      bd_partno;
        struct hd_struct *      bd_part;
        /* number of times partitions within this device have been opened. */
        unsigned                bd_part_count;
        int                     bd_invalidated;
        struct gendisk *        bd_disk;
        struct request_queue *  bd_queue;
        struct backing_dev_info *bd_bdi;
        struct list_head        bd_list;
        /*
         * Private data.  You must have bd_claim'ed the block_device
         * to use this.  NOTE:  bd_claim allows an owner to claim
         * the same device multiple times, the owner must take special
         * care to not mess up bd_private for that case.
         */
        unsigned long           bd_private;
        /* The counter of freeze processes */
        int                     bd_fsfreeze_count;
        /* Mutex for freeze */
        struct mutex            bd_fsfreeze_mutex;
} __randomize_layout;
```

Драйверы блочных устройств, также как и драйверы символьных устройств, должны использовать набор интерфейсов для регистрации, чтобы сделать блочные устройства доступными для ядра. Концепции похожи, но детали регистрации блочных и символьных устройств различны.

Первым шагом, предпринимаемым большинством блочных драйверов, является регистрация себя в ядре. Функция для этой задачи - register_blkdev (которая объявлена в <linux / fs.h>):

int register_blkdev(unsigned int major, const char *name);

Аргументами данной функции являются: major - основной номер, который будет использовать устройство, name – имя, которое ядро будет отображать в / proc / devices. Если major передается как 0, ядро выделяет новый главный номер и возвращает его вызывающей стороне. Как всегда, отрицательное возвращаемое значение из register_blkdev указывает, что произошла ошибка. Начина с ядра 2.6 вызов этой функции совершенно необязателен. Функции, выполняемые register_blkdev, со временем уменьшались. Единственными задачами, которые выполняет этот вызовом в настоящий момент, являются: 1) выделение динамического старшего номера, если требуется, и 2) создание записи в / proc / devices.

Символьные устройства для регистрации в системе нужных операций используют структуру file_operations. Блочные устройства используют для этого struct block_device_operations. На эту структуру ссылается **struct gendisk**. Структура **block_device_operations** объявлена в include/linux/blkdev.h:

```
struct block_device_operations {
        int (*open) (struct block_device *, fmode_t);
        void (*release) (struct gendisk *, fmode_t);
        int (*rw_page)(struct block_device *, sector_t, struct page *, unsigned int);
        int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        unsigned int (*check_events) (struct gendisk *disk,
                                      unsigned int clearing);
        /* ->media_changed() is DEPRECATED, use ->check_events() instead */
        int (*media_changed) (struct gendisk *);
        void (*unlock_native_capacity) (struct gendisk *);
        int (*revalidate_disk) (struct gendisk *);
        int (*getgeo)(struct block_device *, struct hd_geometry *);
        /* this callback is with swap_lock and sometimes page table lock held */
        void (*swap_slot_free_notify) (struct block_device *, unsigned long);
        int (*report_zones)(struct gendisk *, sector_t sector,
                            struct blk_zone *zones, unsigned int *nr_zones,
                            gfp_t gfp_mask);
        struct module *owner;
        const struct pr_ops *pr_ops;
};
```

Функции open() и release() работают так же, как их эквиваленты для драйверов символов. Эти функции вызываются всякий раз, когда устройство открывается и закрывается. Как видно из приведенной структуры одним из параметров функций release, media_change, revalidate_disk, report_zones является параметр типа struct gendisk.

С другой стороны, структура **block_device operations** определяет поле *fops в уже упоминавшейся структуре gendisk. Данная структура struct gendisk (объявлена в <linux / genhd.h>) представляет в ядре отдельное дисковое устройство. Фактически, ядро также использует структуры gendisk для представления разделов, но авторам драйверов не нужно об этом знать. В struct gendisk есть несколько полей, которые должны быть инициализированы драйвером блочного устройства:

```
struct gendisk {
        /* major, first_minor and minors are input parameters only,
         * don't use directly.  Use disk_devt() and disk_max_parts().
         */
        int major;                      /* major number of driver */
        int first_minor;
        int minors;     /* maximum number of minors, =1 for disks that can't be partitioned. */
        char disk_name[DISK_NAME_LEN];          /* name of major driver */
        char *(*devnode)(struct gendisk *gd, umode_t *mode);
        unsigned int events;                    /* supported events */
        unsigned int async_events;  /* async events, subset of all */
        /* Array of pointers to partitions indexed by partno.
         * Protected with matching bdev lock but stat and other
         * non-critical accesses use RCU.  Always access through
         * helpers.
         */
        struct disk_part_tbl __rcu *part_tbl;
        struct hd_struct part0;
        const struct block_device_operations *fops;
        struct request_queue *queue;
        void *private_data;
        int flags;
        struct rw_semaphore lookup_sem;
        struct kobject *slave_dir;
        struct timer_rand_state *random;
        atomic_t sync_io;               /* RAID */
        struct disk_events *ev;
#ifdef  CONFIG_BLK_DEV_INTEGRITY
        struct kobject integrity_kobj;
#endif  /* CONFIG_BLK_DEV_INTEGRITY */
        int node_id;
        struct badblocks *bb;
        struct lockdep_map lockdep_map;
};
```

В известной работе [2] приводится пример драйвера блочного устройства sbull, который реализует блок-ориентированное устройство на основе памяти. По сути, это виртуальный диск. Ядро уже содержит гораздо лучшую реализацию ramdisk, но драйвер, который назван **sbull**, позволяет продемонстрировать создание блочного драйвера, минимизируя при этом несвязанную сложность.

Нужно отметить, что примеры, относящиеся к созданию символьных и блочных устройств, в литературе называются драйверами и используют специальные функции регистрации, добавления и т.п., но эти реализации не построены на основе **специальных структур**, описывающих драйверы struct XXX_driver.

Структуры драйверов

Программный уровень управления внешними устройствами имеет два уровня:

1. Верхний – драйверы устройств.
2. Нижний – функции, для работы с устройствами, предоставляемые ОС.

Рассмотрим три системные структуры: struct device_driver, struct pci_driver, struct usb_driver.

Структура **struct device_driver** представляет универсальную модель драйвера устройства, которая отслеживает все драйверы, известные системе. Основная причина такого отслеживания заключается в том, чтобы позволить ядру драйвера сопоставлять драйверы с новыми устройствами. Драйверы устройств различаются не только в зависимости от типов устройств, но и от того с какой шиной они работают. Но драйверы устройств могут экспортировать информацию и переменные конфигурации, которые не зависят от конкретного устройства и способа взаимодействия с системой.

Аналогично тому, что struct device представляет универсальный интерфейс устройства struct device_driver представляет универсальную структуру драйвера.

```
struct device_driver {
        const char              *name;
        struct bus_type         *bus;
        struct module           *owner;
        const char              *mod_name;      /* used for built-in modules */
        bool suppress_bind_attrs;   /* disables bind/unbind via sysfs */
        enum probe_type probe_type;
        const struct of_device_id     *of_match_table;
        const struct acpi_device_id *acpi_match_table;
        int (*probe) (struct device *dev); /*Вызывается для запроса существования определенного устройства,
                                                может ли этот драйвер работать с ним, и
                                                связать драйвер с конкретным устройством*/
        void (*sync_state)(struct device *dev);
        int (*remove) (struct device *dev); /*Вызывается, когда устройство удаляется из
                                системы, чтобы  отсоединить устройство от этого драйвера.*/
        void (*shutdown) (struct device *dev); /*Вызывается во время выключения, чтобы
                                                отключить устройство*/
        int (*suspend) (struct device *dev, pm_message_t state); /*Вызывается перевести
                                устройство в спящий режим. Обычно в состоянии низкого
                                энергопотребления*/
        int (*resume) (struct device *dev); /* Вызывается, чтобы вывести устройство из спящего режима.*/
        const struct attribute_group **groups;
        const struct attribute_group **dev_groups;
        const struct dev_pm_ops *pm;
        void (*coredump) (struct device *dev);
        struct driver_private *p;
};
```

Для драйверов устройств, подключенных к шине PCI определена структура:

```
struct pci_driver {
        struct list_head        node;
```

```
        const char              *name;
        const struct pci_device_id *id_table;   /* Должен быть ненулевым, чтобы вызывалась
probe */
        int  (*probe)(struct pci_dev *dev, const struct pci_device_id *id);      /* Новое устройство
вставлено */
        void (*remove)(struct pci_dev *dev);    /* Устройство удалено (NULL, если драйвер без
                                                поддержки «горячей» замены)*/
        int  (*suspend)(struct pci_dev *dev, pm_message_t state);   /* Устройство
приостановлено */
        int  (*resume)(struct pci_dev *dev);    /* Устройство проснулось */
        void (*shutdown)(struct pci_dev *dev);
        int  (*sriov_configure)(struct pci_dev *dev, int num_vfs); /* On PF */
        const struct pci_error_handlers *err_handler;
        const struct attribute_group **groups;
        struct device_driverdriver;
        struct pci_dynids    dynids;
};
```

- **name**— имя драйвера, которое должно быть уникальным среди всех PCI-драйверов в ядре и обычно устанавливается равным имени модуля драйвера (после загрузки драйвера это имя появляется в /sys/bus/pci/drivers/).
- **id_table**— массив записей **pci_device_id**.
- **probe**— функция обратного вызова, используемая для инициализации устройства; ***функция probe*** вызывается (во время выполнения функции **pci_register_driver ()** для уже существующих устройств или позже, если новое устройство вставляется) для всех устройств PCI, которые соответствуют таблице идентификаторов и еще не «принадлежат» другим драйверам. Эта функция получает "struct pci_dev \ *" для каждого устройства, чья запись в таблице идентификаторов соответствует устройству. Функция probe возвращает ноль, когда driver выбирает «владение» устройством или код ошибки (отрицательное число) в противном случае. Функция probe всегда вызывается из контекста процесса, поэтому она может спать.
- **remove**— функция обратного вызова при удалении устройства; функция remove () вызывается всякий раз, когда устройство, обрабатываемое этим драйвером, удаляется (либо во время отмены регистрации драйвера, либо когда оно вручную извлекается из слота с возможностью горячей замены). Функция удаления всегда вызывается из контекста процесса, поэтому она может спать.
- **suspend**— функция менеджера энергосохранения вызывается, когда устройство переходит в пассивное состояние (засыпает).
- **resume**— функция менеджера энергосохранения, вызываемая при пробуждении устройства.
- **shutdown** - Hook к списку reboot_notifier_list (kernel / sys.c). Предназначен для остановки любых операций DMA на холостом ходу. Полезно для включения функции пробуждения по локальной сети (NIC) или изменения состояния питания устройства перед перезагрузкой. например drivers/net/e100.c.

Функции probe(), remove(), suspend(), resume(), shutdown() получают указатель на устройство, представленное структурой **pci_dev**.

Для USB драйверов определена структура struct usb_driver. Эта структура используется практически во всех примерах, приводимых в специальной литературе и источниках. Но в ядре Linux объявлена еще одна структура struct usb_device_driver (см. приложение 2).

Драйверы интерфейса USB должны предоставлять методы name, probe () и disconnect () и id_table. Другие поля драйвера являются необязательными. id_table используется для горячего подключения. Он содержит набор дескрипторов, и специализированные данные

могут быть связаны с каждой записью. Эта таблица используется поддержкой горячего подключения как в режиме пользователя, так и в режиме ядра.

```
struct usb_driver {
        const char *name;

        int (*probe) (struct usb_interface *intf,
                      const struct usb_device_id *id);
        void (*disconnect) (struct usb_interface *intf);
        int (*unlocked_ioctl) (struct usb_interface *intf, unsigned int code,
                               void *buf);
        int (*suspend) (struct usb_interface *intf, pm_message_t message);
        int (*resume) (struct usb_interface *intf);
        int (*reset_resume)(struct usb_interface *intf);
        int (*pre_reset)(struct usb_interface *intf);
        int (*post_reset)(struct usb_interface *intf);
        const struct usb_device_id *id_table;
        const struct attribute_group **dev_groups;
        struct usb_dynids dynids;
        struct usbdrv_wrap drvwrap;
        unsigned int no_dynamic_id:1;
        unsigned int supports_autosuspend:1;
        unsigned int disable_hub_initiated_lpm:1;
        unsigned int soft_unbind:1;
};
```

Как видим, в структуре функции probe(), disconnect(), suspend(), resume() в отличие от функций pci_driver получают указатель на struct usb_interface (см. приложение 2).

Анализ приведенных структур показывает, что в состав каждой входят поля с функциями, определенными для работы с драйверами: probe, disconnect, suspend, resume и другие.

Рассмотри как любой драйвер может определить свои функции на примере драйвера pci:

```
Static struct pci_driver my_driver = {

        .name = “my_pci_driver”,

        .probe = my_probe,

        .remove = my_remove,

        .id_table = my_id_table,

}
```

После определения данной структуры драйвер может быть зарегистрирован с помощью функции:

int pci_register_driver(struct pci_driver *dev);

При успешной регистрации функция возвращает 0. Это означает, что устройство готово к работе.

При выгрузке модуля вызывается функция:

void pci_unregister_driver(struct pci_driver *dev);

Кроме имени драйвера и входящих в структуры функций probe и remove, в структуру включена строка id_table.

```
#define PCI_ANY_ID (~0)
/**
 * struct pci_device_id - PCI device ID structure
 * @vendor:                     Vendor ID to match (or PCI_ANY_ID)
 * @device:                     Device ID to match (or PCI_ANY_ID)
 * @subvendor:                  Subsystem vendor ID to match (or PCI_ANY_ID)
 * @subdevice:                  Subsystem device ID to match (or PCI_ANY_ID)
 * @class:                      Device class, subclass, and "interface" to match.
 *                              See Appendix D of the PCI Local Bus Spec or
 *                              include/linux/pci_ids.h for a full list of classes.
 *                              Most drivers do not need to specify class/class_mask
 *                              as vendor/device is normally sufficient.
 * @class_mask:                 Limit which sub-fields of the class field are compared.
 *                              See drivers/scsi/sym53c8xx_2/ for example of usage.
 * @driver_data:  Data private to the driver.
 *                              Most drivers don't need to use driver_data field.
 *                              Best practice is to use driver_data as an index
 *                              into a static list of equivalent device types,
 *                              instead of using it as a pointer.
 */
struct pci_device_id {
        __u32 vendor, device;                   /* Vendor and device ID or PCI_ANY_ID*/
        __u32 subvendor, subdevice;     /* Subsystem ID's or PCI_ANY_ID */
        __u32 class, class_mask;        /* (class,subclass,prog-if) triplet */
        kernel_ulong_t driver_data;     /* Data private to the driver */
};
```

id_table – массив структур или таблица, которая должна заканчиваться пустым элементом. Каждый из ненулевых элементов задается одним из предописанных макросов, например:

```
static pci_device_id my_id_table[] ={

        {PCI_DEVICE(0x0F0F, 0x0F0E)},

        {PCI_DEVICE(0x0F0F, 0x0F0D)},

        {0,}

}
```

Затем используется макрос **MODULE_DEVICE_TABLE,** чтобы экспортировать созданную таблицу в пространство пользователя. Это делается для того, чтобы системы горячего подключения и загрузки модулей (**sysfs, udev** и т.д.) смогли узнать, с какими устройствами работает данный модуль.

Для нашего примера это выглядит следующим образом:

```
MODULE_DEVICE_TABLE( pci, my_id_table );
```

Аналогичные структуры определены для драйверов USB и HID драйверов (см. ниже).

```
/*
 * Device table entry for "new style" table-driven USB drivers.
 * User mode code can read these tables to choose which modules to load.
 * Declare the table as a MODULE_DEVICE_TABLE.
 *
 * A probe() parameter will point to a matching entry from this table.
 * Use the driver_info field for each match to hold information tied
 * to that match:  device quirks, etc.
 *
 * Terminate the driver's table with an all-zeroes entry.
 * Use the flag values to control which fields are compared.
```

Запись в таблице устройств для USB-драйверов с табличным управлением «новый стиль». Код пользовательского режима может прочитать эти таблицы, чтобы выбрать, какие модули загрузить. Объявите таблицу как MODULE_DEVICE_TABLE.

Параметр probe () будет указывать на соответствующую запись из этой таблицы. Используйте поле driver_info для каждого совпадения для хранения информации, связанной с этим совпадением: причуды устройства и т. Д. Уточните таблицу драйверов записью со всеми нулями. Используйте значения флага, чтобы контролировать, какие поля сравниваются.

```
 */

/**
 * struct usb_device_id - identifies USB devices for probing and hotplugging
```

определяет USB-устройства для проверки и горячего подключения

```
 * @match_flags: Bit mask controlling which of the other fields are used to
 *              match against new devices. Any field except for driver_info may be
 *              used, although some only make sense in conjunction with other fields.
 *              This is usually set by a USB_DEVICE_*() macro, which sets all
 *              other fields in this structure except for driver_info.
 * @idVendor: USB vendor ID for a device; numbers are assigned
 *              by the USB forum to its members.
```

USB-идентификатор производителя для устройства; номера назначаются форумом USB своим членам.

```
 * @idProduct: Vendor-assigned product ID.
```

Назначенный поставщиком идентификатор продукта.

```
 * @bcdDevice_lo: Low end of range of vendor-assigned product version numbers.
 *              This is also used to identify individual product versions, for
 *              a range consisting of a single device.
 * @bcdDevice_hi: High end of version number range.  The range of product
 *              versions is inclusive.
 * @bDeviceClass: Class of device; numbers are assigned
 *              by the USB forum.  Products may choose to implement classes,
 *              or be vendor-specific.  Device classes specify behavior of all
 *              the interfaces on a device.
 * @bDeviceSubClass: Subclass of device; associated with bDeviceClass.
 * @bDeviceProtocol: Protocol of device; associated with bDeviceClass.
 * @bInterfaceClass: Class of interface; numbers are assigned
 *              by the USB forum.  Products may choose to implement classes,
 *              or be vendor-specific.  Interface classes specify behavior only
 *              of a given interface; other interfaces may support other classes.
 * @bInterfaceSubClass: Subclass of interface; associated with bInterfaceClass.
 * @bInterfaceProtocol: Protocol of interface; associated with bInterfaceClass.
 * @bInterfaceNumber: Number of interface; composite devices may use
 *              fixed interface numbers to differentiate between vendor-specific
 *              interfaces.
```

```
 * @driver_info: Holds information used by the driver.  Usually it holds
 *              a pointer to a descriptor understood by the driver, or perhaps
 *              device flags.
 *
 * In most cases, drivers will create a table of device IDs by using
 * USB_DEVICE(), or similar macros designed for that purpose.
 * They will then export it to userspace using MODULE_DEVICE_TABLE(),
 * and provide it to the USB core through their usb_driver structure.
 *
 * See the usb_match_id() function for information about how matches are
 * performed.  Briefly, you will normally use one of several macros to help
 * construct these entries.  Each entry you provide will either identify
 * one or more specific products, or will identify a class of products
 * which have agreed to behave the same.  You should put the more specific
 * matches towards the beginning of your table, so that driver_info can
 * record quirks of specific products.
```

В большинстве случаев драйверы создают таблицу идентификаторов устройств с помощью USB_DEVICE () или аналогичных макросов, разработанных для этой цели.

Затем они экспортируют его в пользовательское пространство с помощью MODULE_DEVICE_TABLE () и предоставляют его ядру USB через свою структуру usb_driver.

*

* См. Функцию usb_match_id () для получения информации о том, как выполняются совпадения. Вкратце, вы обычно будете использовать один из нескольких макросов для создания этих записей. Каждая предоставленная вами запись будет либо идентифицировать один или несколько конкретных продуктов, либо будет определять класс продуктов, которые согласились вести себя одинаково. Вы должны поместить более конкретные совпадения в начало вашей таблицы, чтобы driver_info могла записывать причуды определенных продуктов.

```
 */
struct usb_device_id {
        /* which fields to match against? */
        __u16           match_flags;

        /* Used for product specific matches; range is inclusive */
        __u16           idVendor;
        __u16           idProduct;
        __u16           bcdDevice_lo;
        __u16           bcdDevice_hi;

        /* Used for device class matches */
        __u8            bDeviceClass;
        __u8            bDeviceSubClass;
        __u8            bDeviceProtocol;

        /* Used for interface class matches */
        __u8            bInterfaceClass;
        __u8            bInterfaceSubClass;
        __u8            bInterfaceProtocol;

        /* Used for vendor-specific interface matches */
        __u8            bInterfaceNumber;

        /* not matched against */
        kernel_ulong_t  driver_info
                        __attribute__((aligned(sizeof(kernel_ulong_t))));
};
```

Последняя структура связана с HID драйверами. Важно помнить, что драйвер HID обрабатывает те устройства (или фактически те интерфейсы на каждом устройстве), которые соответствуют спецификации HID (англ. Human Interface Device (HID) specification). Однако спецификация HID ничего не говорит о том, что драйвер HID должен делать с информацией, полученной от устройства HID, или откуда поступает информация, отправляемая на устройство, поскольку это, очевидно, зависит от того, каким устройством должно быть делать, и что это за операционная система. Linux (на уровне ядра операционной системы)

поддерживает четыре интерфейса для устройства HID - клавиатуру, мышь, джойстик и универсальный интерфейс, известный как интерфейс событий.

```
struct hid_device_id {
        __u16 bus;
        __u16 group;
        __u32 vendor;
        __u32 product;
        kernel_ulong_t driver_data;
};
```

Типы драйверов

Необходимость драйверов устройств в операционной системе объясняется тем, что каждое отдельное устройство воспринимает только свой строго фиксированный набор специализированных команд, с помощью которых этим устройством можно управлять [3]. Причем команды эти чаще всего предназначены для выполнения низкоуровневых операций. Если бы каждое приложение вынуждено было использовать только эти команды, писать приложения было бы очень сложно, да и размер их был бы очень велик. Поэтому приложения обычно используют команды высокого уровня, предоставляемый ОС, а о преобразовании этих команд в управляющие последовательности для конкретного устройства заботится драйвер этого устройства. Поэтому каждое отдельное устройство, будь то дисковод, клавиатура, мышь или принтер, должно иметь свой программный драйвер, который выполняет роль связующего звена между аппаратной частью устройства и программными приложениями, использующими это устройство.

В Linux драйверы устройств бывают трех типов.

Драйверы первого типа являются частью программного кода ядра (встроены в ядро). Соответствующие устройства автоматически обнаруживаются системой и становятся доступны для приложений. Обычно таким образом обеспечивается поддержка тех устройств, которые необходимы для монтирования корневой файловой системы и запуска компьютера. Примерами таких устройств являются стандартный видеоконтроллер VGA, контроллеры IDE-дисков, материнская плата, последовательные и параллельные порты.

Драйверы второго типа представлены загружаемыми модулями ядра. Они оформлены в виде отдельных файлов и для их подключения (на этапе загрузки или впоследствии) необходимо выполнить отдельную команду подключения модуля. Если необходимость в использовании устройства отпала, модуль можно выгрузить из памяти (отключить). Поэтому использование модулей обеспечивает большую гибкость, так как каждый такой драйвер может быть переконфигурирован без остановки системы. Модули часто используются для управления такими устройствами как SCSI-адаптеры, звуковые и сетевые карты. Разработчики с помощью таких драйверов могут изменять функциональность внешних устройств.

Файлы модулей ядра располагаются в подкаталогах каталога /lib/modules. Обычно при инсталляции системы задается перечень модулей, которые будут автоматически подключаться на этапе загрузки. Список загружаемых модулей хранится в файле /etc/modules. А в файле /etc/modules.conf находится перечень опций для таких модулей. Редактировать этот файл "вручную" не рекомендуется, для этого существуют специальные скрипты (типа update-modules).

И, наконец, **третий тип драйверов.** В этих драйверах устройств программный код драйвера поделен между ядром и специальной утилитой, предназначенной для управления

данным устройством. Например, для драйвера принтера ядро отвечает за взаимодействие с параллельным портом, а формирование управляющих сигналов для принтера осуществляет демон печати lpd.

Но надо специально отметить, что во всех трех случаях непосредственное взаимодействие с устройством осуществляет ядро или какой-то модуль ядра. А пользовательские программы взаимодействуют с драйверами устройств через специальные файлы, расположенные в каталоге /dev и его подкаталогах. То есть взаимодействие прикладных программ с аппаратной частью компьютера в ОС Linux осуществляется по следующей схеме:

устройство ‹-› ядро ‹-› специальный файл устройства ‹-› программа пользователя

Такая схема обеспечивает единый подход ко всем устройствам, которые с точки зрения приложений выглядят как обычные файлы.

Драйверы

При разработке драйверов устройств необходимо ориентироваться на структуры, определенные в ядре. Рассмотрим основные действия, которые должен выполнять драйвер устройства демонстрируются на примере простого драйвера USB.

Известно, что USB драйверы пишутся для интерфейсов устройств, а не для самого устройства. Интерфейс имеет дескриптор типа struct usb_interface (см. приложение 2).

```
int (*probe) (struct usb_interface *intf,  const struct usb_device_id *id);
void (*disconnect) (struct usb_interface *intf);
```

USB драйвер является драйвером устройства, т.е. он должен подключиться к реальному устройству в пространстве аппаратных средств. USB поддерживает «горячее» (plug’n’play) соединение с динамически загружаемым и выгружаемым драйвером. Пользователь не заботится ни о терминировании, ни об IRQ и адресах портов. Загрузка подходящего драйвера осуществляется по комбинации PID/VID (Product ID/Vendor ID).

Интерфейсы API для ядра USB выглядят следующим образом (прототип в <linux/usb.h>):

```
int usb_register(struct usb_driver *driver);
void usb_deregister(struct usb_driver *);
```

В структуре usb_driver в соответствующих полях должны быть указаны имя устройства - name, идентификационная таблица – id_table, используемая для автоматического обнаружения конкретного устройства, и две функции обратного вызова – probe и disconnect, которые вызываются ядром USB при горячем подключении и отключении устройства, соответственно.

Код простейшего драйвера имеет следующий вид:

```
#include <linux/module.h>
```

```
#include <linux/kernel.h>
#include <linux/usb.h>

static int skel_probe(struct usb_interface *interface, const struct usb_device_id *id)
{
   printk(KERN_INFO "Skel drive plugged\n");
   return 0;
}
static void skel_disconnect(struct usb_interface *interface)
{
   printk(KERN_INFO "Skel drive removed\n");
}
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);

static struct usb_driver pen_driver =
{
   .name = "skel",
   .id_table = skel_table,
   .probe = skel_probe,
   .disconnect = skel_disconnect,
};

static int __init usb_skel_init(void)
{
   int result;
   result =  usb_register(&skel_driver);
   if(result<0)
      {
          err("usb_register failed for the "__FILE__"driver. Error number %d",result);
          return -1;
       }
   return 0;
}

static void __exit usb_skel_exit(void)
{
   usb_deregister(&pen_driver);
}

module_init(usb_skel_init);
module_exit(usb_skel_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("XXX");
MODULE_DESCRIPTION("USB Skeleton Registration Driver");
```

Если необходимо написать полноценный драйвер USB для Linux, нужно ознакомьтесь со спецификацией протокола USB. Он объясняет, как структурирована USB-подсистема Linux, и описывает концепцию USB urbs (USB Request Blocks) [8], которые необходимы для драйверов USB.

Одна из главных концепций USB заключается в том, что в USB-системе может быть только один мастер. Им является host-компьютер. USB - устройства всегда отвечают на запросы host-компьютера, но они никогда не могу посылать информацию самостоятельно.

Передача данных выполняется между буфером в памяти хост компьютера и конечной точкой универсальной последовательной шины USB устройства. Перед передачей данные организуются в пакеты. Используемый тип передачи зависит от канала, по которому выполняется передача.

На логическом уровне USB устройство поддерживает транзакции приема и передачи данных. Хост всегда является мастером, а обмен данными должен осуществляться в обоих направлениях:

* OUT - отсылая пакет с флагом OUT, хост отсылает данные устройству
* IN - отсылая пакет с флагом IN, хост отправляет запрос на прием данных из устройства.

В составе USB-функции, то есть в устройстве с интерфейсом USB, имеется периферийный контроллер USB. Как показано на рисунке 4, этот контроллер имеет две основные функции: он взаимодействует с USB-системой (соединяясь с хостом или хабом) и содержит в себе буферы в количестве от одного до шестнадцати, называемые конечными точками.

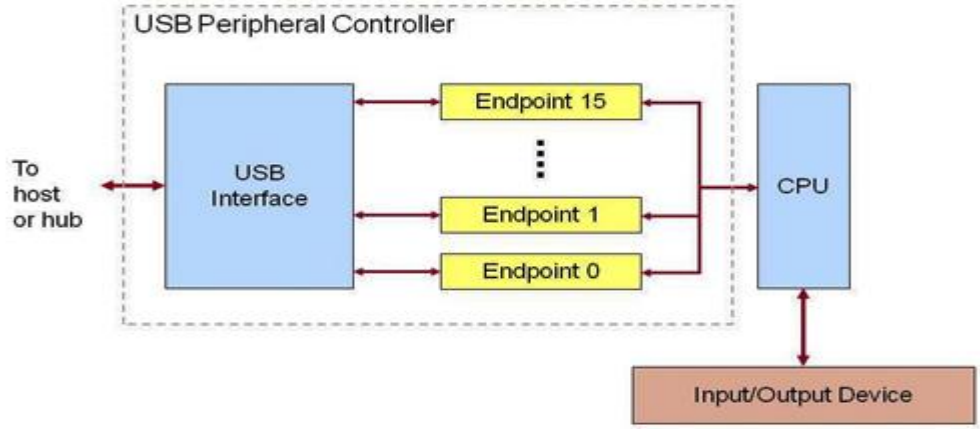

Рис. 4

Конечные точки (endpoints). Это базовый объект связи интерфейса USB. Устройство может иметь до 16 конечных точек, нумерация начинается с 0 и заканчивается 15. Каждая конечная точка может включать в себя два буфера (адреса): входной и выходной. То есть устройство может обладать 32 адресами конечных точек. Каждая USB-функция должна содержать как минимум одну (нулевую) конечную точку с входным и выходным буфером.

Каналы (pipes). USB это протокол, построенный по принципу master/slave. Все общение инициализируется хостом. Хост определяет каналы, которые связаны с конечными точками функции. В отличие от конечной точки, которая имеет физическую сущность в нашем мире, канал является всего лишь логической концепцией, правилом. После установки канала, становится определенным и тип передачи данных, который он поддерживает.

Передачи (transfers). Данные отправляются и принимаются посредством передач или сообщений, состоящих из ряда транзакций, каждая из которых в свою очередь состоит из пакетов. В любой момент времени система USB может поддерживать несколько передач и связанные с ними транзакции. Существует 4 типа передач:

Следующая таблица показывает типы передачи, которые могут использовать драйверы USB устройств:

№п.п	Тип передачи	Описание
1	control	Передача типа control является двунаправленным и предназначен для обмена с устройством короткими пакетами типа «вопрос-ответ». Обеспечивает гарантированную доставку данных. Используется системным ПО USB для выдачи определенных общих команд на USB устройство и позволяет ПО ОС прочитать информацию об устройстве, такую как коды производителя и модели (PID/VID). Передача типа control обычно осуществляется конечной точкой 0 USB устройства, но могут использоваться другие конечные точки.
2	isochronous	Изохронный канал имеет гарантированную пропускную способность (N пакетов за один период шины) и обеспечивает непрерывную передачу данных. Используются для устройств с очень большим объемом данных, где синхронизация по времени является более критической, чем точность передаваемых данных. Передача осуществляется без подтверждения приема. Используется для приложений реального времени, например для передачи аудио и видео информации
3	interrupt	Канал прерывания позволяет доставлять короткие пакеты без гарантии доставки и без подтверждений приема, но с гарантией времени доставки – пакет будет доставлен не позже, чем через N миллисекунд. Например, используется в устройствах ввода таких, как клавиатура, мышь или джойстики.
4	bulk	Поточная или сплошная передача используется устройствами, отправляющими и принимающими большое количество данных. Канал дает гарантию доставки каждого пакета. Поддерживает автоматическую приостановку передачи данных. Однако, не дает гарантии скорости и задержки доставки. Bulk пакеты передаются в последнюю очередь, т.к. имеет самый низкий приоритет передачи и занимают всю свободную полосу пропускания шины.

Блоки запросов URB

Код USB в ядре Linux взаимодействует со всеми устройствами USB с помощью URB (англ. USB request block – блок запроса USB). Блок запроса USB (URB) описывается структурой struct urb (см. приложение 3).

URB используется для передачи или в или из заданной конечной точки USB на заданное USB устройство в асинхронном режиме. В зависимости от потребностей, драйвер USB устройства может выделить для одной конечной точки много блоков или может повторно использовать один urb для множества разных конечных точек. Каждая конечная точка в устройстве может обрабатывать очередь блоков так, что перед тем, как очередь опустеет, к одной конечной точке может быть направлено множество urb. Типичный жизненный цикл содержит следующие шаги:

- Создание urb драйвером USB.
- Назначение urb в определенную точку конечную точку.
- Передача драйвером USB устройства в USB ядро.
- Передача urb драйвером USB в заданный драйвер контроллера USB узла для указанного устройства
- Обработка драйвером контроллера USB узла, который выполняет передачу в USB устройство.
- После завершения работы с urb драйвер контроллера USB узла уведомляет драйвер USB устройства

URB может быть отменен драйвером, который передал urb или драйвером USB. URB создается динамически и содержит счетчик ссылок, что позволяет освободить urb автоматически, когда освобождается блок последним пользователем.

Функции URB

Имеется четыре функции ядра USB, которые обрабатывают URB **:**

• struct urb *usb_alloc_urb(int iso_packets, int mem_flags);

Всякий раз, когда требуется структура URB, эта функция должна вызываться. Параметр iso-packets определяет число изохронных пакетов, которое должен содержать этот urb. Второй параметр mem_flags эквивалентен параметру, который передается в kmalloc() для указания того, как выделять буфера. В случае успеха возвращаемое значение является указателем на структуру URB, предварительно установленную в ноль, в противном случае возвращается указатель NULL.

Например,

```
mouse->irq = usb_alloc_urb(0, GFP_KERNEL);
if (!mouse->irq)
{
    usb_free_coherent(dev, 0, mouse->data, mouse->data_dma);
    input_free_device(input_dev);
    kfree(mouse);
    returen error;
}
```

```
void usb_free_coherent(struct usb_device *dev, size_t size, void *addr,
                       dma_addr_t dma)
{
        if (!dev || !dev->bus)
```

```
                return;
        if (!addr)
                return;
        hcd_buffer_free(dev->bus, size, addr, dma);
}
EXPORT_SYMBOL_GPL(usb_free_coherent);
```
Эта функция освобождает буфер ввода / вывода, позволяя использовать его повторно.

• void usb_free_urb (struct urb *urb);

Сообщает ядру, что драйвер закончил работу с urb.

• int usb_submit_urb(struct urb *urb, int mem_flags);

Эта функция асинхронно отправляет запрос на передачу в ядро USB. Аргумент purb является указателем на ранее выделенную и инициализированную структуру URB. В случае успеха возвращаемое значение равно 0, в противном случае возвращается соответствующий код ошибки. Функция всегда возвращает неблокирование, и можно запланировать несколько URB для разных конечных точек без ожидания. На изохронных конечных точках даже можно запланировать больше URB для одной конечной точки. Это ограничение вызвано механизмами обработки ошибок и повторных попыток протокола USB.

• int usb_unlink_urb(struct urb *urb); или usb_kill_urb(struct urb *urb);

Эта функция отменяет запланированный запрос до его завершения. Функцию можно вызывать синхронно или асинхронно в зависимости от флага передачи USB_ASYNC_UNLINK. Синхронно вызываемая функция ожидает 1 мс и не должна вызываться из обработчика прерываний или завершения. Возвращаемое значение равно 0, если функция завершается успешно. Асинхронно вызванная функция сразу возвращается. Возвращаемое значение равно -EINPROGRESS, если функция была успешно запущена. При вызове usb unlink urb обработчик завершения вызывается после завершения функции. Состояние URB помечается как -ENOENT (синхронно вызывается) или -ECONNRESET (асинхронно вызывается). Функция usb_kill_urb срабатывает, когда устройство отключается.

Следующий пример является более развитым, поскольку обеспечивает передачу данных. Обратите внимание, что флэш-накопитель относится к классу устройств USB массового хранения данных, для работы с которыми предполагается использование команд, похожих на команды SCSI, которые предназначены для передачи данных в источник / приемник данных массовой памяти (bulk endpoints). Поэтому в случае, если данные не отформатированы должным образом, команды read/write, который показаны ниже в листинге кода, могут, в действительности, не передавать данные так, как это от них ожидается. Но все же, в этом коде собран весь драйвер USB. Чтобы получить представление о реальной передаче данных через USB простым и элегантным способом, может потребоваться воспользоваться некоторым специально настроенным устройством USB, похожим на то, что показано здесь [16].

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/usb.h>

#define MIN(a,b) (((a) <= (b)) ? (a) : (b))
```

```
#define BULK_EP_OUT 0x01
#define BULK_EP_IN 0x82
#define MAX_PKT_SIZE 512

static struct usb_device *device;
static struct usb_class_driver class;
static unsigned char bulk_buf[MAX_PKT_SIZE];

static int pen_open(struct inode *i, struct file *f)
{
   return 0;
}
static int pen_close(struct inode *i, struct file *f)
{
   return 0;
}
static ssize_t pen_read(struct file *f, char __user *buf, size_t cnt, loff_t *off)
{
   int retval;
   int read_cnt;
   /* Read the data from the bulk endpoint */
   retval = usb_bulk_msg(device, usb_rcvbulkpipe(device, BULK_EP_IN),
         bulk_buf, MAX_PKT_SIZE, &read_cnt, 5000);
   if (retval)
   {
      printk(KERN_ERR "Bulk message returned %d\n", retval);
      return retval;
   }
   if (copy_to_user(buf, bulk_buf, MIN(cnt, read_cnt)))
   {
      return -EFAULT;
   }
   return MIN(cnt, read_cnt);
}
static ssize_t pen_write(struct file *f, const char __user *buf, size_t cnt, loff_t *off)
{
   int retval;
   int wrote_cnt = MIN(cnt, MAX_PKT_SIZE);
   if (copy_from_user(bulk_buf, buf, MIN(cnt, MAX_PKT_SIZE)))
   {
      return -EFAULT;
   }
   /* Write the data into the bulk endpoint */
   retval = usb_bulk_msg(device, usb_sndbulkpipe(device, BULK_EP_OUT),
         bulk_buf, MIN(cnt, MAX_PKT_SIZE), &wrote_cnt, 5000);
   if (retval)
   {
      printk(KERN_ERR "Bulk message returned %d\n", retval);
      return retval;
   }
   return wrote_cnt;
}
```

```
static struct file_operations fops =
{
   .open = pen_open,
   .release = pen_close,
   .read = pen_read,
   .write = pen_write,
};

static int pen_probe(struct usb_interface *interface, const struct usb_device_id *id)
{
   int retval;
   device = interface_to_usbdev(interface);
   class.name = "usb/pen%d";
   class.fops = &fops;
   if ((retval = usb_register_dev(interface, &class)) < 0)
   {
      /* Something prevented us from registering this driver */
      err("Not able to get a minor for this device.");
   }
   else
   {
      printk(KERN_INFO "Minor obtained: %d\n", interface->minor);
   }

   return retval;
}

static void pen_disconnect(struct usb_interface *interface)
{
   usb_deregister_dev(interface, &class);
}

/* Table of devices that work with this driver */
static struct usb_device_id pen_table[] =
{
   { USB_DEVICE(0x058F, 0x6387) },
   {} /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, pen_table);

static struct usb_driver pen_driver =
{
   .name = "pen_driver",
   .probe = pen_probe,
   .disconnect = pen_disconnect,
   .id_table = pen_table,
};

static int __init pen_init(void)
{
   int result;
```

```
    /* Register this driver with the USB subsystem */
    if ((result = usb_register(&pen_driver)))
    {
        err("usb_register failed. Error number %d", result);
    }
    return result;
}

static void __exit pen_exit(void)
{
    /* Deregister this driver with the USB subsystem */
    usb_deregister(&pen_driver);
}

module_init(pen_init);
module_exit(pen_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Anil Kumar Pugalia <email_at_sarika-pugs_dot_com>");
MODULE_DESCRIPTION("USB Pen Device Driver");
```

Как видно из кода, здесь используется функция, которая создает bulk urb, отсылает его и ждет завершения:

---

int usb_bulk_msg(struct usb_device * *usb_dev***, unsigned int** *pipe***, void *** *data***, int** *len***, int *** *actual_length***, int** *timeout***);**

usb_dev – указатель на usb устройство для отправки сообщения;

pipe - конечная точка "pipe" для отправки сообщения

data - указатель на данные для отправки

len - длина в байтах данных для отправки

actual_length - указатель на местоположение, чтобы поместить фактическую длину, переданную в байтах

timeout - время в msecs для ожидания завершения сообщения до истечения времени ожидания (если 0, ожидание будет длиться вечно)

Эта функция отправляет простое bulk сообщение в указанную конечную точку и ожидает его завершения или истечения времени ожидания.

Эту функцию нельзя использовать из контекста прерывания, как обработчик нижней половины. Если нужно асинхронное сообщение или нужно отправить сообщение из контекста прерывания, то используйте **usb_submit_urb**. Если поток драйвере использует этот вызов, нужно убедиться, что метод отсоединения может дождаться его завершения. Поскольку нет дескриптора используемого URB, то нельзя отменить запрос.

Поскольку в ioctl нет usb_interrupt_msg и USBDEVFS_INTERRUPT, пользователи вынуждены злоупотреблять этой процедурой, используя ее для отправки URB для конечных точек прерываний.

В случае успеха 0. В противном случае отрицательный номер ошибки. Количество переданных байтов будет сохранено в параметре actual_length.

Используемые источники

1. Linux Device Driver Tutorial Part 6 – Cdev structure and File Operations https://embetronicx.com/tutorials/linux/device-drivers/cdev-structure-and-file-operations-of-character-drivers/

2. O'Reilly Chapter 16. Block Drivers https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch16.html

3. PCI Express Port Bus Driver. https://lwn.net/Articles/116311/

4. 3. PCI Express I/O Virtualization Howto https://www.kernel.org/doc/html/latest/PCI/pci-iov-howto.html

5. USB Topology https://www.usblyzer.com/usb-topology.htm

6. USB 3.0: что нужно знать разработчику? Автор: Mike(admin) от 22-09-2013, 17:45 http://digitrode.ru/articles/37-usb-30-chto-nuzhno-znat-razrabotchiku.html

7. Linux для пользователя. Глава 9 Подключение и настройка аппаратных устройств. 9.1 Драйверы устройств. http://www.uhlib.ru/kompyutery_i_internet/linux_dlja_polzovatelja/p10.php

8. USB Request Block (URB) https://www.kernel.org/doc/html/v5.4/driver-api/usb/URB.html

9. Codebot. Настоящие программисты не исправляют... https://codebot.wordpress.com/

10. Writing a Linux Kernel Module — Part 2: A Character Device http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/

11. Как написать свой первый Linux device driver https://habr.com/ru/post/337946/

12. Обслуживание периферии в коде модулей ядра: Часть 51. Взаимодействие с PCI-устройствами https://www.ibm.com/developerworks/ru/library/l-linux_kernel_51/

13. Device drivers infrastructure. The Basic Device Driver-Model Structures https://www.kernel.org/doc/html/v4.14/driver-api/infrastructure.html

14. Разработка драйвера PCI устройства под Linux. https://habr.com/ru/post/348042/

15. CHAPTER 3 Char Drivers. https://static.lwn.net/images/pdf/LDD3/ch03.pdf

16. Anil Kumar Pugalia Драйверы устройств в Linux. Часть 11: Драйверы USB в Linux

    http://rus-linux.net/MyLDP/BOOKS/drivers/linux-device-drivers-11.html

17. Linux Device Driver Tutorial Part 7 – Linux Device Driver Tutorial Programming/ https://embetronicx.com/tutorials/linux/device-drivers/linux-device-driver-tutorial-programming/

18. Linux Driver Tutorial: How to Write a Simple Linux Device Driver.

    https://www.apriorit.com/dev-blog/195-simple-driver-for-linux-os

19. Introduction to Major and Minor Number.

https://www.embhack.com/introduction-to-major-and-minor-number/

20. Блочные устройства. Создание и регистрация блочных устройств в системе. https://www.ibm.com/developerworks/ru/library/l-linux_kernel_block_devices_01/
21. Programming Guide for Linux USB Device Drivers (c) 2000 by Detlef Fliegl, deti@fliegl.de http://usb.cs.tum.edu. http://lmu.web.psi.ch/docu/manuals/software_manuals/linux_sl/usb_linux_programming_guide.pdf

Приложение 1

```
/*
 * Register a single major with a specified minor range.
 *
 * If major == 0 this function will dynamically allocate an unused major.
 * If major > 0 this function will attempt to reserve the range of minors
 * with given major.
 *
 */
static struct char_device_struct *
__register_chrdev_region(unsigned int major, unsigned int baseminor,
                         int minorct, const char *name)
{
        struct char_device_struct *cd, *curr, *prev = NULL;
        int ret;
        int i;
        if (major >= CHRDEV_MAJOR_MAX) {
                pr_err("CHRDEV \"%s\" major requested (%u) is greater than the maximum
(%u)\n",
                       name, major, CHRDEV_MAJOR_MAX-1);
                return ERR_PTR(-EINVAL);
        }
        if (minorct > MINORMASK + 1 - baseminor) {
                pr_err("CHRDEV \"%s\" minor range requested (%u-%u) is out of range of
maximum range (%u-%u) for a single major\n",
                        name, baseminor, baseminor + minorct - 1, 0, MINORMASK);
                return ERR_PTR(-EINVAL);
        }
        cd = kzalloc(sizeof(struct char_device_struct), GFP_KERNEL);
        if (cd == NULL)
                return ERR_PTR(-ENOMEM);
        mutex_lock(&chrdevs_lock);
        if (major == 0) {
                ret = find_dynamic_major();
                if (ret < 0) {
                        pr_err("CHRDEV \"%s\" dynamic allocation region is full\n",
                               name);
                        goto out;
                }
                major = ret;
        }
        ret = -EBUSY;
        i = major_to_index(major);
        for (curr = chrdevs[i]; curr; prev = curr, curr = curr->next) {
                if (curr->major < major)
                        continue;
```

```
                        if (curr->major > major)
                                break;
                        if (curr->baseminor + curr->minorct <= baseminor)
                                continue;
                        if (curr->baseminor >= baseminor + minorct)
                                break;
                        goto out;
                }
                cd->major = major;
                cd->baseminor = baseminor;
                cd->minorct = minorct;
                strlcpy(cd->name, name, sizeof(cd->name));

                if (!prev) {
                        cd->next = curr;
                        chrdevs[i] = cd;
                } else {
                        cd->next = prev->next;
                        prev->next = cd;
                }
                mutex_unlock(&chrdevs_lock);
                return cd;
out:
                mutex_unlock(&chrdevs_lock);
                kfree(cd);
                return ERR_PTR(ret);
}

static struct char_device_struct *
__unregister_chrdev_region(unsigned major, unsigned baseminor, int minorct)
{
                struct char_device_struct *cd = NULL, **cp;
                int i = major_to_index(major);

                mutex_lock(&chrdevs_lock);
                for (cp = &chrdevs[i]; *cp; cp = &(*cp)->next)
                        if ((*cp)->major == major &&
                            (*cp)->baseminor == baseminor &&
                            (*cp)->minorct == minorct)
                                break;
                if (*cp) {
                        cd = *cp;
                        *cp = cd->next;
                }
                mutex_unlock(&chrdevs_lock);
                return cd;
}

/**
 * register_chrdev_region() - register a range of device numbers
 * @from: the first in the desired range of device numbers; must include
 *        the major number.
 * @count: the number of consecutive device numbers required
 * @name: the name of the device or driver.
 *
 * Return value is zero on success, a negative error code on failure.
 */
```

```
int register_chrdev_region(dev_t from, unsigned count, const char *name)
{
        struct char_device_struct *cd;
        dev_t to = from + count;
        dev_t n, next;

        for (n = from; n < to; n = next) {
                next = MKDEV(MAJOR(n)+1, 0);
                if (next > to)
                        next = to;
                cd = __register_chrdev_region(MAJOR(n), MINOR(n),
                               next - n, name);
                if (IS_ERR(cd))
                        goto fail;
        }
        return 0;
fail:
        to = n;
        for (n = from; n < to; n = next) {
                next = MKDEV(MAJOR(n)+1, 0);
                kfree(__unregister_chrdev_region(MAJOR(n), MINOR(n), next - n));
        }
        return PTR_ERR(cd);
}

/**
 * alloc_chrdev_region() - register a range of char device numbers
 * @dev: output parameter for first assigned number
 * @baseminor: first of the requested range of minor numbers
 * @count: the number of minor numbers required
 * @name: the name of the associated device or driver
 *
 * Allocates a range of char device numbers.  The major number will be
 * chosen dynamically, and returned (along with the first minor number)
 * in @dev.  Returns zero or a negative error code.
        Выделяет диапазон номеров символьных устройств. Старший номер будет выбран динамически и
        возвращен (вместе с первым младшим номером) в @dev. Возвращает ноль или отрицательный код
        ошибки.

 */
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,
                        const char *name)
{
        struct char_device_struct *cd;
        cd = __register_chrdev_region(0, baseminor, count, name);
        if (IS_ERR(cd))
                return PTR_ERR(cd);
        *dev = MKDEV(cd->major, cd->baseminor);
        return 0;
}

/**
 * __register_chrdev() - create and register a cdev occupying a range of minors
 * @major: major device number or 0 for dynamic allocation
 * @baseminor: first of the requested range of minor numbers
 * @count: the number of minor numbers required
 * @name: name of this range of devices
 * @fops: file operations associated with this devices
 *
```

```
 * If @major == 0 this functions will dynamically allocate a major and return
 * its number.
 *
 * If @major > 0 this function will attempt to reserve a device with the given
 * major number and will return zero on success.
 *
 * Returns a -ve errno on failure.
 *
 * The name of this device has nothing to do with the name of the device in
 * /dev. It only helps to keep track of the different owners of devices. If
 * your module name has only one type of devices it's ok to use e.g. the name
 * of the module here.
 */
int __register_chrdev(unsigned int major, unsigned int baseminor,
                      unsigned int count, const char *name,
                      const struct file_operations *fops)
{
        struct char_device_struct *cd;
        struct cdev *cdev;
        int err = -ENOMEM;

        cd = __register_chrdev_region(major, baseminor, count, name);
        if (IS_ERR(cd))
                return PTR_ERR(cd);

        cdev = cdev_alloc();
        if (!cdev)
                goto out2;

        cdev->owner = fops->owner;
        cdev->ops = fops;
        kobject_set_name(&cdev->kobj, "%s", name);

        err = cdev_add(cdev, MKDEV(cd->major, baseminor), count);
        if (err)
                goto out;

        cd->cdev = cdev;

        return major ? 0 : cd->major;
out:
        kobject_put(&cdev->kobj);
out2:
        kfree(__unregister_chrdev_region(cd->major, baseminor, count));
        return err;
}

/**
 * unregister_chrdev_region() - unregister a range of device numbers
 * @from: the first in the range of numbers to unregister
 * @count: the number of device numbers to unregister
 *
 * This function will unregister a range of @count device numbers,
 * starting with @from.  The caller should normally be the one who
 * allocated those numbers in the first place...
 */
void unregister_chrdev_region(dev_t from, unsigned count)
{
```

```
        dev_t to = from + count;
        dev_t n, next;

        for (n = from; n < to; n = next) {
                next = MKDEV(MAJOR(n)+1, 0);
                if (next > to)
                        next = to;
                kfree(__unregister_chrdev_region(MAJOR(n), MINOR(n), next - n));
        }
}

/**
 * __unregister_chrdev - unregister and destroy a cdev
 * @major: major device number
 * @baseminor: first of the range of minor numbers
 * @count: the number of minor numbers this cdev is occupying
 * @name: name of this range of devices
 *
 * Unregister and destroy the cdev occupying the region described by
 * @major, @baseminor and @count.  This function undoes what
 * __register_chrdev() did.
 */
void __unregister_chrdev(unsigned int major, unsigned int baseminor,
                         unsigned int count, const char *name)
{
        struct char_device_struct *cd;

        cd = __unregister_chrdev_region(major, baseminor, count);
        if (cd && cd->cdev)
                cdev_del(cd->cdev);
        kfree(cd);
}
```

Приложение 2

```
/**
 * struct usb_driver - identifies USB interface driver to usbcore
 * @name: The driver name should be unique among USB drivers,
 *        and should normally be the same as the module name.
 * @probe: Called to see if the driver is willing to manage a particular
 *        interface on a device.  If it is, probe returns zero and uses
 *        usb_set_intfdata() to associate driver-specific data with the
 *        interface.  It may also use usb_set_interface() to specify the
 *        appropriate altsetting.  If unwilling to manage the interface,
 *        return -ENODEV, if genuine IO errors occurred, an appropriate
 *        negative errno value.
```

Вызывается, чтобы узнать, готов ли драйвер управлять определенным интерфейсом на устройстве. Если это так, probe возвращает ноль и использует usb_set_intfdata (), чтобы связать специфичные для драйвера данные с интерфейсом. Он также может использовать usb_set_interface () для указания подходящего altsetting. Если вы не хотите управлять интерфейсом, верните -ENODEV, если произошли подлинные ошибки ввода-вывода, соответствующее отрицательное значение errno.

```
 * @disconnect: Called when the interface is no longer accessible, usually
 *        because its device has been (or is being) disconnected or the
 *        driver module is being unloaded.
 * @unlocked_ioctl: Used for drivers that want to talk to userspace through
 *        the "usbfs" filesystem.  This lets devices provide ways to
```

```
 *		expose information to user space regardless of where they
 *		do (or don't) show up otherwise in the filesystem.
 * @suspend: Called when the device is going to be suspended by the
 *		system either from system sleep or runtime suspend context. The
 *		return value will be ignored in system sleep context, so do NOT
 *		try to continue using the device if suspend fails in this case.
 *		Instead, let the resume or reset-resume routine recover from
 *		the failure.
 * @resume: Called when the device is being resumed by the system.
 * @reset_resume: Called when the suspended device has been reset instead
 *		of being resumed.
 * @pre_reset: Called by usb_reset_device() when the device is about to be
 *		reset.  This routine must not return until the driver has no active
 *		URBs for the device, and no more URBs may be submitted until the
 *		post_reset method is called.
 * @post_reset: Called by usb_reset_device() after the device
 *		has been reset
 * @id_table: USB drivers use ID table to support hotplugging.
 *		Export this with MODULE_DEVICE_TABLE(usb,...).  This must be set
 *		or your driver's probe function will never get called.
 * @dev_groups: Attributes attached to the device that will be created once it
 *		is bound to the driver.
 * @dynids: used internally to hold the list of dynamically added device
 *		ids for this driver.
 * @drvwrap: Driver-model core structure wrapper.
 * @no_dynamic_id: if set to 1, the USB core will not allow dynamic ids to be
 *		added to this driver by preventing the sysfs file from being created.
 * @supports_autosuspend: if set to 0, the USB core will not allow autosuspend
 *		for interfaces bound to this driver.
 * @soft_unbind: if set to 1, the USB core will not kill URBs and disable
 *		endpoints before calling the driver's disconnect method.
 * @disable_hub_initiated_lpm: if set to 1, the USB core will not allow hubs
 *		to initiate lower power link state transitions when an idle timeout
 *		occurs.  Device-initiated USB 3.0 link PM will still be allowed.
 *
 * USB interface drivers must provide a name, probe() and disconnect()
 * methods, and an id_table.  Other driver fields are optional.
 *
 * The id_table is used in hotplugging.  It holds a set of descriptors,
 * and specialized data may be associated with each entry.  That table
 * is used by both user and kernel mode hotplugging support.
 *
 * The probe() and disconnect() methods are called in a context where
 * they can sleep, but they should avoid abusing the privilege.  Most
 * work to connect to a device should be done when the device is opened,
 * and undone at the last close.  The disconnect code needs to address
 * concurrency issues with respect to open() and close() methods, as
 * well as forcing all pending I/O requests to complete (by unlinking
 * them as necessary, and blocking until the unlinks complete).
```

Драйверы интерфейса USB должны предоставлять методы name, probe () и disconnect () и id_table. Другие поля драйвера являются необязательными. id_table используется для горячего подключения. Он содержит набор дескрипторов, и специализированные данные могут быть связаны с каждой записью. Эта таблица используется поддержкой горячего подключения как в режиме пользователя, так и в режиме ядра.

```
 *
```

* Методы probe () и disconnect () вызываются в контексте, в котором они могут спать, но они должны избегать злоупотребления привилегией. Большая часть работы по подключению к

устройству должна выполняться, когда устройство открыто, и отменено при последнем закрытии. Код разъединения должен разрешать проблемы параллелизма в отношении методов open () и close (), а также принудительно завершать все ожидающие запросы ввода-вывода (отсоединяя их по мере необходимости и блокируя до завершения отмены ссылок).
 */
```
struct usb_driver {
        const char *name;

        int (*probe) (struct usb_interface *intf,
                      const struct usb_device_id *id);
        void (*disconnect) (struct usb_interface *intf);
        int (*unlocked_ioctl) (struct usb_interface *intf, unsigned int code,
                               void *buf);
        int (*suspend) (struct usb_interface *intf, pm_message_t message);
        int (*resume) (struct usb_interface *intf);
        int (*reset_resume)(struct usb_interface *intf);
        int (*pre_reset)(struct usb_interface *intf);
        int (*post_reset)(struct usb_interface *intf);
        const struct usb_device_id *id_table;
        const struct attribute_group **dev_groups;
        struct usb_dynids dynids;
        struct usbdrv_wrap drvwrap;
        unsigned int no_dynamic_id:1;
        unsigned int supports_autosuspend:1;
        unsigned int disable_hub_initiated_lpm:1;
        unsigned int soft_unbind:1;
};



/**
 * struct usb_interface - what usb device drivers talk to
 * @altsetting: array of interface structures, one for each alternate
 *      setting that may be selected.  Each one includes a set of
 *      endpoint configurations.  They will be in no particular order.
 * @cur_altsetting: the current altsetting.
 * @num_altsetting: number of altsettings defined.
 * @intf_assoc: interface association descriptor
 * @minor: the minor number assigned to this interface, if this
 *      interface is bound to a driver that uses the USB major number.
 *      If this interface does not use the USB major, this field should
 *      be unused.  The driver should set this value in the probe()
 *      function of the driver, after it has been assigned a minor
 *      number from the USB core by calling usb_register_dev().
 * @condition: binding state of the interface: not bound, binding
 *      (in probe()), bound to a driver, or unbinding (in disconnect())
 * @sysfs_files_created: sysfs attributes exist
 * @ep_devs_created: endpoint child pseudo-devices exist
 * @unregistering: flag set when the interface is being unregistered
 * @needs_remote_wakeup: flag set when the driver requires remote-wakeup
 *      capability during autosuspend.
 * @needs_altsetting0: flag set when a set-interface request for altsetting 0
 *      has been deferred.
 * @needs_binding: flag set when the driver should be re-probed or unbound
 *      following a reset or suspend operation it doesn't support.
 * @authorized: This allows to (de)authorize individual interfaces instead
 *      a whole device in contrast to the device authorization.
```

```
 * @dev: driver model's view of this device
 * @usb_dev: if an interface is bound to the USB major, this will point
 *           to the sysfs representation for that device.
 * @reset_ws: Used for scheduling resets from atomic context.
 * @resetting_device: USB core reset the device, so use alt setting 0 as
 *           current; needs bandwidth alloc after reset.
 *
 * USB device drivers attach to interfaces on a physical device.  Each
 * interface encapsulates a single high level function, such as feeding
 * an audio stream to a speaker or reporting a change in a volume control.
 * Many USB devices only have one interface.  The protocol used to talk to
 * an interface's endpoints can be defined in a usb "class" specification,
 * or by a product's vendor.  The (default) control endpoint is part of
 * every interface, but is never listed among the interface's descriptors.
 *
 * The driver that is bound to the interface can use standard driver model
 * calls such as dev_get_drvdata() on the dev member of this structure.
 *
 * Each interface may have alternate settings.  The initial configuration
 * of a device sets altsetting 0, but the device driver can change
 * that setting using usb_set_interface().  Alternate settings are often
 * used to control the use of periodic endpoints, such as by having
 * different endpoints use different amounts of reserved USB bandwidth.
 * All standards-conformant USB devices that use isochronous endpoints
 * will use them in non-default settings.
 *
 * The USB specification says that alternate setting numbers must run from
 * 0 to one less than the total number of alternate settings.  But some
 * devices manage to mess this up, and the structures aren't necessarily
 * stored in numerical order anyhow.  Use usb_altnum_to_altsetting() to
 * look up an alternate setting in the altsetting array based on its number.
```

Драйверы USB-устройств подключаются к интерфейсам на физическом устройстве. Каждый интерфейс инкапсулирует одну высокоуровневую функцию, такую как подача аудиопотока на динамик или сообщение об изменении в регуляторе громкости. Многие USB-устройства имеют только один интерфейс. Протокол, используемый для связи с конечными точками интерфейса, может быть определен в спецификации usb «class» или поставщиком продукта. Конечная точка управления (по умолчанию) является частью каждого интерфейса, но никогда не указывается в дескрипторах интерфейса.

```
 *
```

 * Драйвер, связанный с интерфейсом, может использовать стандартные вызовы модели драйвера, такие как dev_get_drvdata () для члена dev этой структуры. Каждый интерфейс может иметь альтернативные настройки. Первоначальная конфигурация устройства устанавливает altsetting 0, но драйвер устройства может изменить этот параметр, используя usb_set_interface (). Альтернативные настройки часто используются для управления использованием периодических конечных точек, например, когда разные конечные точки используют разное количество зарезервированной полосы пропускания USB. Все совместимые со стандартами USB-устройства, которые используют изохронные конечные точки, будут использовать их в нестандартных настройках.

```
 *
```

 * В спецификации USB указано, что номера альтернативных настроек должны быть от 0 до единицы меньше, чем общее количество альтернативных настроек. Но некоторым устройствам удается все испортить, и структуры все равно не обязательно хранятся в числовом порядке. Используйте usb_altnum_to_altsetting (), чтобы найти альтернативный параметр в массиве altsetting на основе его номера.

```
 */
struct usb_interface {
        /* array of alternate settings for this interface,
```

```
         * stored in no particular order */
        struct usb_host_interface *altsetting;

        struct usb_host_interface *cur_altsetting;      /* the currently
                                         * active alternate setting */
        unsigned num_altsetting;        /* number of alternate settings */

        /* If there is an interface association descriptor then it will list
         * the associated interfaces */
        struct usb_interface_assoc_descriptor *intf_assoc;

        int minor;                      /* minor number this interface is
                                         * bound to */
        enum usb_interface_condition condition;         /* state of binding */
        unsigned sysfs_files_created:1;         /* the sysfs attributes exist */
        unsigned ep_devs_created:1;     /* endpoint "devices" exist */
        unsigned unregistering:1;       /* unregistration is in progress */
        unsigned needs_remote_wakeup:1;     /* driver requires remote wakeup */
        unsigned needs_altsetting0:1;   /* switch to altsetting 0 is pending */
        unsigned needs_binding:1;       /* needs delayed unbind/rebind */
        unsigned resetting_device:1;    /* true: bandwidth alloc after reset */
        unsigned authorized:1;          /* used for interface authorization */

        struct device dev;              /* interface specific device info */
        struct device *usb_dev;
        struct work_struct reset_ws;    /* for resets in atomic context */
};
```

Для USB драйверов в ядре Linux определена структура **usb_device_driver.** Драйверы USB должны содержать все поля, перечисленные в структуре, кроме drvwrap. Struct usb_device_driver определяет драйвер USB-устройства для usbcore.

```
struct usb_device_driver {
        const char *name; /* Имя драйвера должно быть уникальным среди драйверов
                             USB и обычно должно   совпадать с именем модуля*/
        int (*probe) (struct usb_device *udev);
        void (*disconnect) (struct usb_device *udev);
        int (*suspend) (struct usb_device *udev, pm_message_t message);
        int (*resume) (struct usb_device *udev, pm_message_t message);
        const struct attribute_group **dev_groups;
        struct usbdrv_wrap drvwrap;
        unsigned int supports_autosuspend:1;
};
```

* @probe: функция вызывается, чтобы узнать, готов ли драйвер управлять конкретным устройством. Если это так, probe возвращает ноль и использует dev_set_drvdata (), чтобы связать данные, относящиеся к драйверу, с устройством. Если управление устройством невозможно, то возвращается отрицательное значение errno.
* @disconnect: вызывается, когда устройство больше недоступно, обычно потому, что оно было (или отключается) или модуль драйвера выгружается.
* @suspend: вызывается, когда устройство будет приостановлено системой.
* @resume: вызывается, когда устройство возобновляется системой.
* @dev_groups: атрибуты, прикрепленные к устройству, которое будет создано после его привязки к   драйверу.

Приложение 3

```
/**
 * struct urb - USB Request Block
 * @urb_list: For use by current owner of the URB.
 * @anchor_list: membership in the list of an anchor
 * @anchor: to anchor URBs to a common mooring
 * @ep: Points to the endpoint's data structure.  Will eventually
 *      replace @pipe.
 * @pipe: Holds endpoint number, direction, type, and more.
 *      Create these values with the eight macros available;
 *      usb_{snd,rcv}TYPEpipe(dev,endpoint), where the TYPE is "ctrl"
 *      (control), "bulk", "int" (interrupt), or "iso" (isochronous).
 *      For example usb_sndbulkpipe() or usb_rcvintpipe().  Endpoint
 *      numbers range from zero to fifteen.  Note that "in" endpoint two
 *      is a different endpoint (and pipe) from "out" endpoint two.
 *      The current configuration controls the existence, type, and
 *      maximum packet size of any given endpoint.
```

Содержит номер конечной точки, направление, тип и многое другое. Создайте эти значения с помощью восьми доступных макросов; usb_ {snd, rcv} TYPEpipe (dev, конечная точка), где TYPE - «ctrl» (контроль), «bulk», «int» (прерывание) или «iso» (изохронный). Например, usb_sndbulkpipe () или usb_rcvintpipe (). Числа конечных точек варьируются от нуля до пятнадцати. Обратите внимание, что конечная точка «in» два - это другая конечная точка (и канал) от конечной точки «out» два. Текущая конфигурация контролирует существование, тип и максимальный размер пакета любой заданной конечной точки.

```
 * @stream_id: the endpoint's stream ID for bulk streams
 * @dev: Identifies the USB device to perform the request.
```

Определяет USB-устройство для выполнения запроса.

```
 * @status: This is read in non-iso completion functions to get the
 *      status of the particular request.  ISO requests only use it
 *      to tell whether the URB was unlinked; detailed status for
 *      each frame is in the fields of the iso_frame-desc.
 * @transfer_flags: A variety of flags may be used to affect how URB
 *      submission, unlinking, or operation are handled.  Different
 *      kinds of URB can use different flags.
 * @transfer_buffer:  This identifies the buffer to (or from) which the I/O
 *      request will be performed unless URB_NO_TRANSFER_DMA_MAP is set
 *      (however, do not leave garbage in transfer_buffer even then).
 *      This buffer must be suitable for DMA; allocate it with
 *      kmalloc() or equivalent.  For transfers to "in" endpoints, contents
 *      of this buffer will be modified.  This buffer is used for the data
 *      stage of control transfers.
 * @transfer_dma: When transfer_flags includes URB_NO_TRANSFER_DMA_MAP,
 *      the device driver is saying that it provided this DMA address,
 *      which the host controller driver should use in preference to the
 *      transfer_buffer.
 * @sg: scatter gather buffer list, the buffer size of each element in
 *      the list (except the last) must be divisible by the endpoint's
 *      max packet size if no_sg_constraint isn't set in 'struct usb_bus'
 * @num_mapped_sgs: (internal) number of mapped sg entries
 * @num_sgs: number of entries in the sg list
 * @transfer_buffer_length: How big is transfer_buffer.  The transfer may
 *      be broken up into chunks according to the current maximum packet
 *      size for the endpoint, which is a function of the configuration
 *      and is encoded in the pipe.  When the length is zero, neither
 *      transfer_buffer nor transfer_dma is used.
 * @actual_length: This is read in non-iso completion functions, and
```

```
*		it tells how many bytes (out of transfer_buffer_length) were
*		transferred.  It will normally be the same as requested, unless
*		either an error was reported or a short read was performed.
*		The URB_SHORT_NOT_OK transfer flag may be used to make such
*		short reads be reported as errors.
* @setup_packet: Only used for control transfers, this points to eight bytes
*		of setup data.  Control transfers always start by sending this data
*		to the device.  Then transfer_buffer is read or written, if needed.
* @setup_dma: DMA pointer for the setup packet.  The caller must not use
*		this field; setup_packet must point to a valid buffer.
* @start_frame: Returns the initial frame for isochronous transfers.
* @number_of_packets: Lists the number of ISO transfer buffers.
* @interval: Specifies the polling interval for interrupt or isochronous
*		transfers.  The units are frames (milliseconds) for full and low
*		speed devices, and microframes (1/8 millisecond) for highspeed
*		and SuperSpeed devices.
* @error_count: Returns the number of ISO transfers that reported errors.
* @context: For use in completion functions.  This normally points to
*		request-specific driver context.
* @complete: Completion handler. This URB is passed as the parameter to the
*		completion function.  The completion function may then do what
*		it likes with the URB, including resubmitting or freeing it.
* @iso_frame_desc: Used to provide arrays of ISO transfer buffers and to
*		collect the transfer status for each buffer.
*
* This structure identifies USB transfer requests.  URBs must be allocated by
* calling usb_alloc_urb() and freed with a call to usb_free_urb().
* Initialization may be done using various usb_fill_*_urb() functions.  URBs
* are submitted using usb_submit_urb(), and pending requests may be canceled
* using usb_unlink_urb() or usb_kill_urb().
*
* Data Transfer Buffers:
*
* Normally drivers provide I/O buffers allocated with kmalloc() or otherwise
* taken from the general page pool.  That is provided by transfer_buffer
* (control requests also use setup_packet), and host controller drivers
* perform a dma mapping (and unmapping) for each buffer transferred.  Those
* mapping operations can be expensive on some platforms (perhaps using a dma
* bounce buffer or talking to an IOMMU),
* although they're cheap on commodity x86 and ppc hardware.
*
* Alternatively, drivers may pass the URB_NO_TRANSFER_DMA_MAP transfer flag,
* which tells the host controller driver that no such mapping is needed for
* the transfer_buffer since
* the device driver is DMA-aware.  For example, a device driver might
* allocate a DMA buffer with usb_alloc_coherent() or call usb_buffer_map().
* When this transfer flag is provided, host controller drivers will
* attempt to use the dma address found in the transfer_dma
* field rather than determining a dma address themselves.
*
* Note that transfer_buffer must still be set if the controller
* does not support DMA (as indicated by hcd_uses_dma()) and when talking
* to root hub. If you have to trasfer between highmem zone and the device
* on such controller, create a bounce buffer or bail out with an error.
* If transfer_buffer cannot be set (is in highmem) and the controller is DMA
* capable, assign NULL to it, so that usbmon knows not to use the value.
* The setup_packet must always be set, so it cannot be located in highmem.
```

 *
* Эта структура идентифицирует запросы передачи USB. URB должны быть выделены путем вызова usb_alloc_urb () и освобождены с помощью вызова usb_free_urb (). Инициализация может быть выполнена с использованием различных функций usb_fill _ * _ urb (). URB передаются с использованием usb_submit_urb (), и отложенные запросы могут быть отменены с использованием usb_unlink_urb () или usb_kill_urb ().
 *
 * Буферы передачи данных:
 *
 * Обычно драйверы предоставляют буферы ввода / вывода, выделенные с помощью kmalloc () или иным образом взятые из общего пула страниц. Это обеспечивается Transfer_buffer (запросы на управление также используют setup_packet), а драйверы хост-контроллера выполняют отображение (и удаление) dma для каждого переданного буфера. Эти операции отображения могут быть дорогими на некоторых платформах (возможно, с использованием буфера отказов dma или общения с IOMMU), хотя они дешевы на стандартном оборудовании x86 и ppc.
 *
 * В качестве альтернативы драйверы могут передавать флаг передачи URB_NO_TRANSFER_DMA_MAP, который сообщает драйверу контроллера хоста, что такое отображение не требуется для Transfer_buffer, поскольку драйвер устройства поддерживает DMA. Например, драйвер устройства может выделить буфер DMA с помощью usb_alloc_coherent () или вызвать usb_buffer_map (). Когда этот флаг передачи предоставлен, драйверы хост-контроллера будут пытаться использовать адрес dma, найденный в поле Transfer_dma, вместо того, чтобы самим определять адрес dma.
 *
 * Обратите внимание, что Transfer_buffer все еще должен быть установлен, если контроллер не поддерживает DMA (как указано hcd_uses_dma ()) и при общении с корневым концентратором. Если вам нужно переключиться между highmem-зоной и устройством на таком контроллере, создайте буфер возврата или выведите его с ошибкой. Если Transfer_buffer не может быть установлен (находится в highmem), и контроллер поддерживает DMA, присвойте ему значение NULL, чтобы usbmon знал, что не следует использовать это значение. Пакет setup_packet всегда должен быть установлен, поэтому он не может быть помещен в highmem.
 * Initialization:
 *
 * All URBs submitted must initialize the dev, pipe, transfer_flags (may be
 * zero), and complete fields. All URBs must also initialize
 * transfer_buffer and transfer_buffer_length. They may provide the
 * URB_SHORT_NOT_OK transfer flag, indicating that short reads are
 * to be treated as errors; that flag is invalid for write requests.
Все отправленные URB должны инициализировать поля dev, pipe, Transfer_flags (могут быть нулевыми) и заполнить поля. Все URB также должны инициализировать Transfer_buffer и Transfer_buffer_length. Они могут предоставлять флаг передачи URB_SHORT_NOT_OK, указывающий, что короткие чтения должны рассматриваться как ошибки; этот флаг недействителен для запросов на запись.
 *
 * Bulk URBs may
 * use the URB_ZERO_PACKET transfer flag, indicating that bulk OUT transfers
 * should always terminate with a short packet, even if it means adding an
 * extra zero length packet.
Bulk URB могут использовать флаг передачи URB_ZERO_PACKET, указывающий, что массовые передачи OUT должны всегда заканчиваться коротким пакетом, даже если это означает добавление дополнительного пакета нулевой длины.
 *
 * Control URBs must provide a valid pointer in the setup_packet field.
 * Unlike the transfer_buffer, the setup_packet may not be mapped for DMA
 * beforehand.
Control URB должны предоставлять действительный указатель в поле setup_packet. В отличие от Transfer_buffer, setup_packet может не отображаться для DMA заранее.

```
 *
 * Interrupt URBs must provide an interval, saying how often (in milliseconds
 * or, for highspeed devices, 125 microsecond units)
 * to poll for transfers.  After the URB has been submitted, the interval
 * field reflects how the transfer was actually scheduled.
 * The polling interval may be more frequent than requested.
 * For example, some controllers have a maximum interval of 32 milliseconds,
 * while others support intervals of up to 1024 milliseconds.
 * Isochronous URBs also have transfer intervals.  (Note that for isochronous
 * endpoints, as well as high speed interrupt endpoints, the encoding of
 * the transfer interval in the endpoint descriptor is logarithmic.
 * Device drivers must convert that value to linear units themselves.)
```

Interrupt URB должны предоставлять интервал, указывающий, как часто (в миллисекундах или для высокоскоростных устройств 125 микросекундных единиц) запрашивать передачи. После того, как URB был представлен, поле интервала отражает, как передача была фактически запланирована. Интервал опроса может быть более частым, чем требуется. Например, некоторые контроллеры имеют максимальный интервал 32 миллисекунды, в то время как другие поддерживают интервалы до 1024 миллисекунд. Изохронные URB также имеют интервалы передачи. (Обратите внимание, что для изохронных конечных точек, а также конечных точек высокоскоростных прерываний кодирование интервала передачи в дескрипторе конечной точки является логарифмическим. Драйверы устройств должны сами преобразовывать это значение в линейные единицы.)

```
 *
 * If an isochronous endpoint queue isn't already running, the host
 * controller will schedule a new URB to start as soon as bandwidth
 * utilization allows.  If the queue is running then a new URB will be
 * scheduled to start in the first transfer slot following the end of the
 * preceding URB, if that slot has not already expired.  If the slot has
 * expired (which can happen when IRQ delivery is delayed for a long time),
 * the scheduling behavior depends on the URB_ISO_ASAP flag.  If the flag
 * is clear then the URB will be scheduled to start in the expired slot,
 * implying that some of its packets will not be transferred; if the flag
 * is set then the URB will be scheduled in the first unexpired slot,
 * breaking the queue's synchronization.  Upon URB completion, the
 * start_frame field will be set to the (micro)frame number in which the
 * transfer was scheduled.  Ranges for frame counter values are HC-specific
 * and can go from as low as 256 to as high as 65536 frames.
```

Если изохронная очередь конечных точек еще не запущена, хост-контроллер запланирует запуск нового URB, как только позволит использование полосы пропускания. Если очередь работает, то новый URB будет запланирован для запуска в первом интервале передачи после окончания предыдущего URB, если этот интервал еще не истек. Если интервал истек (что может произойти, если доставка IRQ задерживается на длительное время), поведение планирования зависит от флага URB_ISO_ASAP. Если флаг снят, тогда URB будет запланирован для запуска в истекшем интервале, подразумевая, что некоторые из его пакетов не будут переданы; если флаг установлен, тогда URB будет запланирован в первом не истекшем временном интервале, нарушая синхронизацию очереди. После завершения URB поле start_frame будет установлено на (микро) номер кадра, в котором была запланирована передача. Диапазоны значений счетчика кадров зависят от HC и могут варьироваться от 256 до 65536 кадров.

```
 *
 * Isochronous URBs have a different data transfer model, in part because
 * the quality of service is only "best effort".  Callers provide specially
 * allocated URBs, with number_of_packets worth of iso_frame_desc structures
 * at the end.  Each such packet is an individual ISO transfer.  Isochronous
 * URBs are normally queued, submitted by drivers to arrange that
 * transfers are at least double buffered, and then explicitly resubmitted
 * in completion handlers, so
 * that data (such as audio or video) streams at as constant a rate as the
```

```
 * host controller scheduler can support.
 *
```
Изохронные URB имеют другую модель передачи данных, отчасти потому, что качество обслуживания - это только «наилучшее усилие». Вызывающие абоненты предоставляют специально выделенные URB, в конце которых стоит число number_of_packets структур iso_frame_desc. Каждый такой пакет является отдельной передачей ISO. Изохронные URB обычно ставятся в очередь, представляются драйверами для обеспечения того, чтобы передачи были по меньшей мере с двойной буферизацией, а затем явно повторно передаются в обработчиках завершения, чтобы потоки данных (такие как аудио или видео) передавались с такой же постоянной скоростью, которую может поддерживать планировщик хост-контроллера ,
```
 * Completion Callbacks: Завершение обратных вызовов:
 *
 * The completion callback is made in_interrupt(), and one of the first
 * things that a completion handler should do is check the status field.
 * The status field is provided for all URBs.  It is used to report
 * unlinked URBs, and status for all non-ISO transfers.  It should not
 * be examined before the URB is returned to the completion handler.
```
Обратный вызов завершения выполняется в in_interrupt (), и одна из первых вещей, которые должен сделать обработчик завершения, - это проверка поля состояния. Поле статуса предоставляется для всех URB. Он используется для сообщения о несвязанных URB и статусе для всех передач не-ISO. Его не следует проверять до того, как URB будет возвращен обработчику завершения.
```
 *
 * The context field is normally used to link URBs back to the relevant
 * driver or request state.
```
Поле контекста обычно используется для связи URB с соответствующим драйвером или состоянием запроса.
```
 *
 * When the completion callback is invoked for non-isochronous URBs, the
 * actual_length field tells how many bytes were transferred.  This field
 * is updated even when the URB terminated with an error or was unlinked.
```
Когда обратный вызов завершения вызывается для неизохронных URB, поле actual_length сообщает, сколько байтов было передано. Это поле обновляется, даже если URB завершен с ошибкой или не был связан.
```
 *
 * ISO transfer status is reported in the status and actual_length fields
 * of the iso_frame_desc array, and the number of errors is reported in
 * error_count.  Completion callbacks for ISO transfers will normally
 * (re)submit URBs to ensure a constant transfer rate.
```
Статус передачи ISO сообщается в полях status и actual_length массива iso_frame_desc, а количество ошибок - в error_count. Обратные вызовы завершения для передач ISO обычно (повторно) передают URB для обеспечения постоянной скорости передачи.
```
 *
 * Note that even fields marked "public" should not be touched by the driver
 * when the urb is owned by the hcd, that is, since the call to
 * usb_submit_urb() till the entry into the completion routine.
```
Обратите внимание, что даже поля, помеченные как public, не должны затрагиваться драйвером, когда urb принадлежит hcd, то есть с момента вызова usb_submit_urb () до входа в подпрограмму завершения.
```
 */
struct urb {
        /* private: usb core and host controller only fields in the urb */
        struct kref kref;               /* reference count of the URB */
        int unlinked;                           /* unlink error code */
        void *hcpriv;                           /* private data for host controller */
        atomic_t use_count;             /* concurrent submissions counter */
        atomic_t reject;                /* submissions will fail */
```

```
        /* public: documented fields in the urb that can be used by drivers */
        struct list_head urb_list;      /* list head for use by the urb's
                                         * current owner */
        struct list_head anchor_list;   /* the URB may be anchored */
        struct usb_anchor *anchor;
        struct usb_device *dev;                 /* (in) pointer to associated device */
        struct usb_host_endpoint *ep;  /* (internal) pointer to endpoint */
        unsigned int pipe;              /* (in) pipe information */
        unsigned int stream_id;                 /* (in) stream ID */
        int status;                     /* (return) non-ISO status */
        unsigned int transfer_flags;    /* (in) URB_SHORT_NOT_OK | ...*/
        void *transfer_buffer;                  /* (in) associated data buffer */
        dma_addr_t transfer_dma;        /* (in) dma addr for transfer_buffer */
        struct scatterlist *sg;         /* (in) scatter gather buffer list */
        int num_mapped_sgs;                     /* (internal) mapped sg entries */
        int num_sgs;                            /* (in) number of entries in the sg list */
        u32 transfer_buffer_length;     /* (in) data buffer length */
        u32 actual_length;              /* (return) actual transfer length */
        unsigned char *setup_packet;    /* (in) setup packet (control only) */
        dma_addr_t setup_dma;                   /* (in) dma addr for setup_packet */
        int start_frame;                /* (modify) start frame (ISO) */
        int number_of_packets;                  /* (in) number of ISO packets */
        int interval;                           /* (modify) transfer interval
                                                 * (INT/ISO) */
        int error_count;                /* (return) number of ISO errors */
        void *context;                          /* (in) context for completion */
        usb_complete_t complete;        /* (in) completion routine */
        struct usb_iso_packet_descriptor iso_frame_desc[0];
                                                /* (in) ISO ONLY */
};
```