TTY SYSTEM

April 20, 2019 (Http://Www.Mysixue.Com/?P=127) JL (Http://Www.Mysixue.Com/?Author=1)

1. Concept introduction : terminal

In the Linux system , the concept associated with the terminal makes it easy for people confused . First of all there is the concept of the terminal , and then there are various types of terminals (serial terminal , pseudo-terminals , console terminals , control terminal), there is a concept called console.

So what is a terminal ? What is a console terminal ? What is a console?
In order to clarify these questions , let's introduce these concepts in turn .

1.1 terminal

As we all know , the original computer because the price is expensive , and therefore , a computer is generally used by more than one person at the same time . Specifically have in the past that can be connected to a computer device , only the monitor and keyboard, as well as simple processing The circuit itself does not have the ability to process computer information. It is responsible for connecting to a normal computer (usually through a serial port) , then logging in to the computer and operating the computer. Of course, the computer operating systems at that time were all multi-tasking and multi-user operating systems. Such a device with only a monitor and keyboard that can be connected to a computer through a serial port is called a **terminal** .

The main purpose is to provide human-computer interaction terminal interface , so that others can use the terminal to control the unit .
In the Linux system , tty is the terminal subsystem . Tty a word derived from the Teletypes, is a terminal device first appeared, similar to the teletype. tty-core is the core of the terminal subsystem . tty-core layer is the character device drivers , through the character device drivers , terminal subsystem in the / dev create various directory tty node , the following will specifically describe these nodes . There With these nodes , you can control the machine through the terminal .

How to control it ?
Imagine that there is a Raspberry Pi board . After the system is started , a node is created under /dev . Then there is a program that provides the ability to control the machine , such as getty, which runs on the board . Getty first prompts the user Login , for example, it will output a "login:" string to the terminal node , and then the string enters the tty-core through the node . Note that at this time, "login:" only exists in the Linux kernel on the board , there is nothing People can see it . What to do after tty-core receives the string ? It needs to send the string to the user , how to send it ? You can choose the serial port of the Raspberry Pi , and then the user can use some other machine such as an XP computer On , through a tool such as SecureCRT open the serial port , you can see the "login:" a . The user then enters the user login name , will be along the same route back to the getty program , getty verify the user name is a valid name , then prompts the user to enter a password ... … , In this way, human-computer interaction and the function of controlling the machine are realized .

What role does the serial port play in such a process ? It is a carrier for transmitting data . According to the different carriers , terminals can be divided into serial port terminals , pseudo terminals , and console terminals .

1.2 console

After understanding the concept of a terminal , let's take a look at what a console is .
Simply put , the console is the display subsystem + input subsystem of Linux .

Raspberry Pi board or in an example , if the board pick a HDMI display , plug in a USB mouse and keyboard , then HDMI + mouse and keyboard is the board of the console .

The console, like the terminal , has input / output functions . For example, the system can print / get information through the serial port ; it can also print information through the display system of the console , and get information through the input system .

After understanding the concept of the console , it is easy to understand the console terminal behind .

In addition , there is also a concept in the Linux kernel called the console subsystem . Although console is translated as console , in the semantic environment of this article , please distinguish between console and console : console is related to the printk mechanism in the kernel , and The console refers to the (display + input) subsystem .

1.3 Different types of terminals

According to different carriers , terminals can be divided into multiple types . The following are introduced separately

Serial port terminal (/dev/ttySn)

Is well understood , it is the carrier is a serial terminal . Device node name is usually / dev / ttyS0 the like , there are USB to serial types of terminals , the node name is usually / dev / ttyUSB0 the like .

Console terminal (/dev/console, /dev/tty0, /dev/tty1 …)

Above we understand what a console is . The console itself has the function of receiving input and displaying output , but its input is generally the input subsystem (keyboard , mouse, etc.). Its output is generally the display system . Console terminal It is to use the input / output function of the console as a carrier and use it to create a terminal .

The node names of the console terminal are : /dev/tty0, /dev/tty1 … , /dev/tty6

Take the Raspberry Pi board as an example , connect a USB mouse , HDMI display , and start the system . After the system is started , you will see a login interface on HDMI . This login interface is created by the getty program on /dev/tty1 . press Alt + F1 - F6, you can see 6 landfall interface , the login screen is getty , respectively, in / dev / tty1-6 created on . / etc / inittab will control getty program which landed on the console terminal .

/ dev / tty0 it may be understood . 1 link , which is linked to the console terminal is currently being used , for example, now by Alt + F2 to switch to the / dev / tty2 corresponding terminal console , and then enter the command echo test> / dev / tty0 , You will see test on the console terminal corresponding to /dev/tty2 . If you use Alt+F3 to switch to another terminal and do the same action , you will also see test on the terminal corresponding to F3 . But no matter where In which terminal , enter the command echo test> /dev/tty2, you will only see test on the terminal corresponding to Alt+F2 . No matter which console terminal the current system uses , " system related information " will be sent to/dev/tty0. Only the " system " or super user root can write to /dev/tty0 .

/dev/console can also be understood as a link . You can log in on /dev/console only in single-user mode .
Through bootargs , you can tell the Linux kernel where the printk information will be printed during the system startup phase .
On the Raspberry Pi , if console=/dev/ttyS0, 115200 console=/dev/tty1 , the information printed by printk will be seen on the serial port and HDMI .

However, when the Linux kernel is started, when an application goes to open /dev/console , the last value passed in is obtained. Such as the example is / dev / tty1. Above embodiment is based on whether you enter any terminal echo test> / dev / console, eventually in HDMI of tty1 displayed on the terminal. After the kernel is started, some programs (such as ssh, alsa-lib, etc.) will be initialized during the startup process of the file system . At this time, the output information of these programs will be located on /dev/console , which is why we can only The reason for seeing this information on HDMI .

Pseudo terminal (pty)

The pseudo terminal is mainly used to control the machine through the network .

Take telnet as an example . On the Raspberry Pi board , there will be a telnet daemon , which communicates with other machines through the network (TCP/IP protocol) , and monitors whether other machines want to connect to this machine through telnet . when the connection is received , daemon fork illustrating a sub-process , running the control program of the machine on the sub-processes (such as getty). Next getty opens a pseudo-terminal node (/ dev / ttyp2), we have the The node is called the slave node (s2). Then getty will send a "login:" string to s2 . When this string is passed toAfter the tty-core is in , where should the next step be sent ? If it is a serial port terminal , it can be sent through the serial port . However, in the pseudo-terminal , the string will be sent to another node (/dev/ptyp2), we This node is called the master device node (m2).The telnet daemon will read the data in m2 , and then send it to other machines through the TCP/IP protocol .

Therefore : the pseudo terminal is a logical device (s2/m2) that appears in pairs , and the carrier of the pseudo terminal is not real hardware , but a logical device (m2) written in software .

Pseudo-terminal and terminal in front of that form of expression biggest difference , is that it always appears in pairs , rather than a single. It is divided into " pseudo terminal master device (/dev/ptyMN)" and " pseudo terminal slave device " . (/dev/ttyMN) . Among them, M and N are named as follows:

M: pqrstuvwxyzabcde total 16 th

N: 0 1 2 3 4 5 6 7 8 9 abcdef total 16 th

In this way, the default support is 256 at most. This naming method has some problems, and the maximum number of terminals is also limited. Therefore, the Linux kernel introduces a new naming method: UNIX98_PTYS

UNIX98_PTYS

Under this naming method, there is a device node (/dev/ptmx) as the master device of all pseudo-terminals. When a process opens /dev/ptmx , a corresponding slave device will be generated in the /dev/pts/ directory. At this time, the master device (1) and the slave device (N) have a one-to-many relationship .

Control terminal (tty)

The terminal /dev/tty does not have any carrier, it can be understood as a link, which will link to the actual terminal opened by the current process. Enter tty in the command line of the current process to view the terminal corresponding to /dev/tty . For example, the program getty runs on the slave device /dev/pts/5 which is the terminal , then when the tty command is entered , the display is /dev/pts/5

1.4 Understand the terminals present in the system

/proc/ tty/ drivers :

showing the name of the driver, the default node name, the major number for the driver, the range of minors used by the driver, and the type of the tty driver

cat /proc/tty/drivers

name of the driver	default node name	major number	range of minors	type of the tty driver
/dev/tty	/dev/tty	5	0	system:/dev/tty
/dev/console	/dev/console	5	1	system:console
/dev/ptmx	/dev/ptmx	5	2	system
/dev/vc/0	/dev/vc/0	4	0	system:vtmaster
usbserial	/dev/ttyUSB	188	0-254	serial
serial	/dev/ttyS	4	64-67	serial
pty_slave	/dev/pts	136	0-255	pty:slave
pty_master	/dev/ptm	128	0-255	pty:master
pty_slave	/dev/ttyp	3	0-255	pty:slave
pty_master	/dev/pty	2	0-255	pty:master
unknown	/dev/tty	4	1-63	console

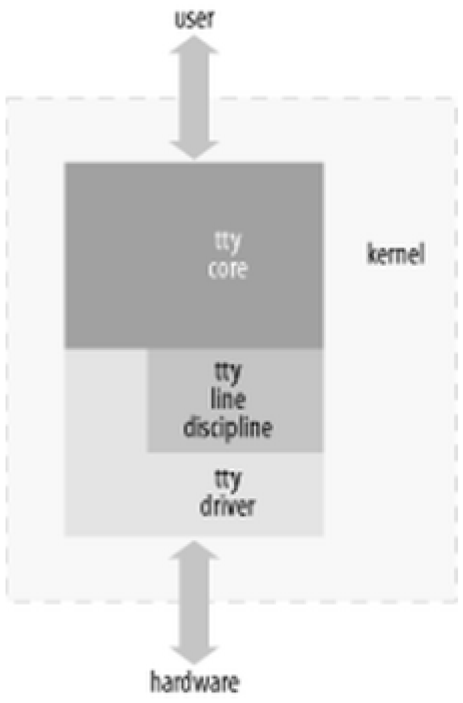
/proc/ tty/ driver/ files
contains individual files for some of the tty drivers, if they implement that functionality. The default serial driver creates a file in this directory that shows a lot of serial-port-specific information about the hardware

/sys/class/ tty
All of the tty devices currently registered and present in the kernel have their own subdirectory under /sys/class/tty. Within that subdirectory, there is a “dev” file that contains the major and minor number assigned to that tty device. If the driver tells the kernel the locations of the physical device and driver associated with the tty device, it creates symlinks back to them

2. Introduction to tty subsystem architecture

All terminal nodes are character device drivers , so the uppermost layer is **character device drivers .**

Below the character device driver is the tty subsystem , first paste a picture



The tty core is an abstraction of the concept of the terminal , which implements the general functions of various different types of terminals
The tty driver is the **driver** of the carrier. For example, if we use the serial port as the carrier, the tty driver is the driver of the serial port.

The driver only cares about how to send data to the hardware (such as the serial port , which is the sending register) and how to receive data from the hardware. The core will consider how to interact with the user space in a unified form, and what the interactive data format is. The data format here refers to the concept of software, which can be understood as a protocol, such as whether the header needs to be encapsulated and what the header information is.

When the core receives the data, it will pass it to the tty line discipline , which will send it to the driver . The driver converts the data into a format acceptable to the hardware and sends it from the hardware. Conversely, when the hardware receives the data, the driver will write the data to a buffer, and then push the data in the buffer to the discipline buffer, and the user space will read the data from the discipline buffer through the read interface .

The core can also directly interact with the driver without going through the discipline . But usually there is a discipline .

The main purpose of **tty line discipline** is to decapsulate / encapsulate the transmitted data on some protocols , such as PPP or Bluetooth .

From the driver 's point of view, it doesn't know whether the data is directly given to it by the core or given to it after the discipline . The driver only knows to send the received data to the hardware and read the data from the hardware. It is not clear whether the data encapsulates some protocols. This design is also logical. The hardware only knows the transmission and reception data of one bit by one bit , regardless of the meaning of the transmitted data.

After understanding these three concepts , we know that if you want to add a serial port terminal, you need to make a serial port driver. This driver must conform to the tty driver specification, that is , implement the necessary interface functions according to the tty driver 's requirements. Then register with the tty core , and everything will be fine. The same is true for other types of carriers, such as virtual terminals or console terminals, just implement a tty driver and register it.

In the embedded SOC , the serial port is generally called UART or USART , and there is generally a chapter in the data manual of each chip to describe this module. Different chip manufacturers, such as Atmel and TI , have somewhat different UART modules, but most of them are the same, such as start/stop bit , baud rate, etc. Therefore , another concept is abstracted in the Linux kernel : Serial core

Serial core : Serial core is under the tty driver , it abstracts some common things of the serial device, so that for the UART modules of different manufacturers , there is no need to completely implement the interface required by the tty driver from beginning to end , just define A simple UART driver , and then register with Serial Core , then Serial Core will encapsulate itself into the form of a tty driver , register with the tty core , and complete the action of adding a serial port terminal. Simplifies the development of serial port terminal drivers.

Next , we first introduce the tty driver, which is a connecting module:
Pair , it tty core interaction
The next , it provides an interface to the serial core

Then we are introducing tty core, then serial core, and finally tty line discipline.

3. tty driver

3.1 Introduction

If you want to write a terminal driver , you need to follow the steps below :

- First , create a struct **tty_driver** structure .

  The kernel code provides an API (alloc_tty_driver) specifically used to create this structure and allocate memory to the structure .

  ```
  struct tty _driver  tiny_tty_driver = alloc_tty_driver(TINY_TTY_MINORS);
  ```

- Then , define a **tty_operations** structure , and preparation of the corresponding functions implemented :

  ```
  static struct tty_operations serial_ops = {
          .open = tiny_open,
          .close = tiny_close,
          .write = tiny_write,
          .write_room = tiny_write_room,
          .set_termios = tiny_set_termios,
  };
  ```

- Then , initialize the tiny_tty_driver just created

  ```
      /* initialize the tty driver */
          tiny_tty_driver->owner = THIS_MODULE;
          tiny_tty_driver->driver_name = "tiny_tty";
          tiny_tty_driver->name = "ttty";
          tiny_tty_driver->devfs_name = “tts/ttty%d”;
          tiny_tty_driver->major = TINY_TTY_MAJOR,
          tiny_tty_driver->type = TTY_DRIVER_TYPE_SERIAL,
          tiny_tty_driver->subtype = SERIAL_TYPE_NORMAL,
          tiny_tty_driver->flags = TTY_DRIVER_REAL_RAW | TTY_DRIVER_NO_DEVFS,
          tiny_tty_driver->init_termios = tty_std_termios;
          tiny_tty_driver->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL | CLOCAL;
       tty_set_ operations( tiny_tty_driver, &serial_ops);
  ```

- Then , call the API provided by the tty driver subsystem to register the driver .

  ```
      /* register the tty driver */
      retval = tty_register_driver(tiny_tty_driver);
  ```

When the registration is completed without errors , you will find the following changes :

- /dev/

  There may be more than one under /dev/The device node starting with tiny_tty_driver->name , such as /dev/ttty0, /dev/ttty1. Why is it possible but not certain ? How many nodes will appear under /dev/ ? Why is ittiny_tty_driver-> name beginning ? These questions will find answers in this section provides a detailed analysis .

- /proc/tty/drivers

  There will be an extra line in the file

  ```
   $ cat /proc/tty/drivers
  tiny_tty /dev/ttty 240 0-3 serial
  ```

  The data in this row are : tiny_tty_driver->driver_name , tiny_tty_driver->name , major device number , minor device number range , tiny_tty_driver->type

- /sys/class/tty

  There will be multiple subdirectories in this directory , and the names of the subdirectories start with tiny_tty_driver->name . Again , the rest of this section will explain why .

The above are the basic steps to write a terminal driver . Whether you are writing a serial terminal driver , a virtual terminal driver , or a console terminal driver , you should follow the above steps .

However, for the serial terminal driver , serial core has already completed the above steps for you , you only need to register with the serial core subsystem .

3.2 Main data structure

According to the introduction in the previous section , we have extracted several main data structures and introduced them respectively : tty_driver , tty_operations .

tty _driver

In the tty subsystem , tty_driver is used to describe a tty driver . To write a terminal driver , you must define a tty_driver structure . Then use this structure to register with the tty subsystem .

Header file :include/linux/tty_driver.h

struct tty_driver	**Comment**
int magic	magic number for this structure
struct kref kref	Reference count
struct cdev *cdevs	Describe the structure of a character device driver . As mentioned at the b eginning of this article , all terminal nodes (/dev/ttyxx) are character devi ce drivers
struct module *owner	
const char *driver_name	Will appear in the first column of /proc/tty/drivers . The driver_name variable should be set to something short, descriptive, and **unique** among all tty drivers in the kernel
const char*name	会出现在/dev/ , /sys/class/tty/ , /proc/tty/drivers的第二列.
intname_base	name的起始编号, 一般情况下默认是0 /dev/下的节点名和/sys/class/tty/下的目录名是由(name+name_base)组成的. 例如name=ttty, name_base=0, 组合之后就是ttty0
intmajor	该driver的主设备号. 与字符设备驱动相关.
intminor_start	该driver的次设备号的起始值. 与字符设备驱动相关. major+minor_start就是该driver的起始设备号
unsigned intnum	表示该driver在注册字符设备驱动的时候, 可以注册几个次设备. 次设备的设备号从(major+minor_start)开始递增. 假如num = 3, 如果使能了创建设备节点, 则/dev/下会多出来3个节点, /sys/class/tty/下也会多出来3个文件夹

shorttype	该driver的类型, 以下几种类型之一: include/linux/tty_driver.h /* tty driver types */ #define TTY_DRIVER_TYPE_SYSTEM0x0001 #define TTY_DRIVER_TYPE_CONSOLE0x0002 #define TTY_DRIVER_TYPE_SERIAL 0x0003 #define TTY_DRIVER_TYPE_PTY 0x0004 #define TTY_DRIVER_TYPE_SCC 0x0005 /* scc driver */ #define TTY_DRIVER_TYPE_SYSCONS 0x0006
short subtype	The driver 's subtype , one of the following subtypes include/linux/tty_driver.h /* system subtypes (magic, used by tty_io.c) */ #define SYSTEM_TYPE_TTY 0x0001 #define SYSTEM_TYPE_CONSOLE 0x0002 #define SYSTEM_TYPE_SYSCONS 0x0003 #define SYSTEM_TYPE_SYSPTMX 0x0004 /* pty subtypes (magic, used by tty_io.c) */ #define PTY_TYPE_MASTER 0x0001 #define PTY_TYPE_SLAVE 0x0002 /* serial subtype definitions */ #define SERIAL_TYPE_NORMAL 1
struct ktermios init_termios	If the user space needs to configure a terminal's baud rate , start / stop bit , parity and other parameters , generally a termios structure will be prepared , and then this structure will be set into the kernel driver . init_termios represents the initial termios of the driver
unsigned long flags	The driver 's flags, can be used with several types or (|) flags decide in the tty when registration subsystem , the system will take what action , such as whether to create / dev / node and so on . flags are defined in the include / Linux / tty_driver.h , for each flag means , the document also detailed notes : #define TTY_DRIVER_INSTALLED 0x0001 #define TTY_DRIVER_RESET_TERMIOS 0x0002 #define TTY_DRIVER_REAL_RAW 0x0004 #define TTY_DRIVER_DYNAMIC_DEV 0x0008 #define TTY_DRIVER_DEVPTS_MEM 0x0010 #define TTY_DRIVER_HARDWARE_BREAK 0x0020 #define TTY_DRIVER_DYNAMIC_ALLOC 0x0040 TTY_DRIVER_UNNUMBERED_NODE #define 0x0080
struct proc_dir_entry *proc_entry	proc system related , used to generate files under /proc/ tty/driver /
struct tty_driver *other	only used for the PTY driver
struct tty_struct **ttys	Pointer , pointing tty_struct structure , tty_struct will tty core detail section
struct tty_port **ports	Pointer , pointing tty_port structure , tty_port will tty core detail section

struct ktermios **termios	Used to link all termios related to the driver
void *driver_state	
const struct tty_operations *ops	The ops associated with the driver , this structure will be specifically intro duced later
struct list_head tty_drivers	tty driver subsystem will have a global list head , mount all registered driv er. The tty_drivers here are used to hook themselves to the global linked list head

tty_operations

tty_operations is used to describe a tty_driver operation function .

Careful observation of these operating parameters of the function , will find that they are associated with struct tty_struct related to this structure . Tty_struct is for describing a tty device is open later , associated with the status of all . In other words , tty_struct is a run-time phase The data structure . We will introduce this structure in detail in the tty core section .

Header file : include/linux/tty_driver.h

struct tty_operations	**Comment**
struct tty_struct * (* **lookup**)(struct tty_driver *driv er, struct inode *inode, int idx)	The meaning of these interface functions will not be introduced in detail f or now , just a brief list
int (* **install**)(struct tty_driver *driver, struct tty_stru ct *tty)	
void (* **remove**)(struct tty_driver *driver, struct tty_ struct *tty)	
int (* **open**)(struct tty_struct * tty, struct file * filp)	
void (* **close**)(struct tty_struct * tty, struct file * filp)	
void (* **shutdown**)(struct tty_struct *tty)	
void (* **cleanup**)(struct tty_struct *tty)	
int (* **write**)(struct tty_struct * tty, const unsigned c har *buf, int count)	
int (* **put_char**)(struct tty_struct *tty, unsigned char ch)	
void (* **flush_chars**)(struct tty_struct *tty)	
int (* **write_room**)(struct tty_struct *tty)	
int (* **chars_in_buffer**)(struct tty_struct *tty)	
int (* **ioctl**)(struct tty_struct *tty, unsigned int cmd, unsigned long arg)	
long (* **compat_ioctl**)(struct tty_struct *tty, unsigne d int cmd, unsigned long arg)	
void (* **set_termios**)(struct tty_struct *tty, struct kte rmios * old)	

void (* **throttle**)(struct tty_struct * tty)	
void (* **unthrottle**)(struct tty_struct * tty)	
void (* **stop**)(struct tty_struct *tty)	
void (* **start**)(struct tty_struct *tty)	
void (* **hangup**)(struct tty_struct *tty)	
int (* **break_ctl**)(struct tty_struct *tty, int state)	
void (* **flush_buffer**)(struct tty_struct *tty)	
void (* **set_ldisc**)(struct tty_struct *tty)	
void (* **wait_until_sent**)(struct tty_struct *tty, int timeout)	
void (* **send_xchar**) (struct tty_struct * TTY, char CH)	
int (* **tiocmget**)(struct tty_struct *tty)	
int (* **tiocmset**)(struct tty_struct *tty, unsigned int set, unsigned int clear)	
int (* **resize**)(struct tty_struct *tty, struct winsize *ws)	
int (* **set_termiox**)(struct tty_struct *tty, struct termiox *tnew)	
int (* **get_icount**)(struct tty_struct *tty, struct serial_icounter_struct *icount)	
#ifdef CONFIG_CONSOLE_POLL int (* **poll_init**)(struct tty_driver *driver, int line, char *options); int (* **poll_get_char**)(struct tty_driver *driver, int line); void (* **poll_put_char**)(struct tty_driver *driver, int line, char ch); #endif	
const struct file_operations *proc_fops	

3.3 Main API description

According to the introduction in Section 3.1 , we know the most important API: tty_register_driver .
The API which will call several other API: tty_register_device , proc_tty_register_driver
Below we introduce them separately .

tty_register_driver

If you want to write a terminal driver , you will first prepare the tty_driver and tty_operations structures , and then call tty_register_driver to register with the tty driver subsystem .

Here we analyze in detail , tty_register_driver which in the end what had been done thing .

Header file : include/linux/tty.h

Implementation file : drivers/tty/tty_io.c

int tty_register_driver(struct tty_driver *driver)

- Assign major and minor device numbers in a dynamic / static manner , and assign them to driver-> major , driver-> minor_start .
- if (driver->flags & TTY_DRIVER_DYNAMIC_ALLOC) , then call tty_cdev_add to register a character device driver to the character device driver subsystem .

  tty_cdev_add encapsulates some APIs of character device drivers . Let's look at the details of this function :

  ```
  static int tty_cdev_add ( struct tty_driver * driver , dev_t dev ,
          unsigned int index , unsigned int count )
  {
      /* init here, since reused cdevs cause crashes */
      cdev_ init ( & driver -> cdevs [ index ], & tty_fops );
      driver -> cdevs [ index ]. owner = driver -> owner ;
      return cdev_add (& driver -> cdevs [ index ], dev , count );
  }
  ```

  When cdev_add returns successfully , the character device driver has been successfully registered . However, please note that the device node will not be automatically created in /dev/ at this time . There will be other codes to create the device node later .

  In addition, you need to pay special attention to the structure of tty_fops , which is a structure defined by the kernel system (indrivers / tty / tty_io.c in). assume / dev / down has created a device node , when we call in user space open / read / write / close operations such time , it will eventually be mapped to tty_fops this structure .
- list_add(&dri ver->tty_drivers, &tty_drivers) ;

  tty_drivers is a global linked list header defined in drivers/tty/tty_io.c . Here, " driver " is linked to this global linked list header .
- if (!(driver->flags & TTY_DRIVER_DYNAMIC_DEV)) , (if flags is not defined TTY_DRIVER_DYNAMIC_DEV)

      for (i = 0; i <driver->num; i++)   (for each num)
          Call tty_register_device once .

  tty_register_device will be in / dev / create directory 1 character device driver corresponding node , but also in / sys / class / tty create directory 1 pieces of association subdirectory . for circulating total calls (driver-> NUM) times tty_register_device , so / dev / under will appear (driver-> NUM) a device node , / sys / class / tty can also occur under (driver-> NUM) a sub-directory .

  As for why tty_register_device Why can create device nodes , what node name ? And why in / sys / class / tty create subdirectories , what directory name ? Later would Detailed analysis of the API, you will find the answer .
- proc_tty_register_driver (driver)

  proc_tty_register_driver will create a subdirectory under the /proc/tty/driver/ directory , and the name of the subdirectory is tty_driver->driver_name .

  Later it will be devoted to what proc_tty_register_driver this API.
- driver->flags |= TTY_DRIVER_INSTALLED

  Set flags sign , on behalf of the driver has been correctly registered .

tty_register_device

tty_register_device is mainly used to generate device nodes and subdirectories under /sys/class/tty .

In the preparation of the final drive , when calling tty_register_driver to tty -time Subsystem registration , if not set TTY_DRIVER_DYNAMIC_DEV , it will automatically call tty_register_device; if set TTY_DRIVER_DYNAMIC_DEV , can also manually call back tty_register_device to create the device nodes and class subdirectories .

Let's analyze in detail what exactly is done in tty_register_device .

Header file : include/linux/tty.h
Implementation file : drivers/tty/tty_io.c

```
struct device * tty_register_device (struct tty_driver *driver, unsigned index,
  struct device *device)
{
return tty_register_device_attr(driver, index, device, NULL, NULL);
}
```

Call tty_register_device_attr directly , let's take a look at tty_register_device_attr :

```
struct device * tty_register_device_attr ( struct tty_driver * driver ,
                           unsigned index , struct device * device ,
                           void * drvdata ,
                           const struct attribute_group ** attr_grp )
```

- If (index >= driver->num) , an error will be returned . It means that the passed index parameter cannot be greater than the num number of the driver itself

-
  ```
  if ( driver -> type == TTY_DRIVER_TYPE_PTY )
          pty_line_ name ( driver , index , name );
      else
          tty_line_ name ( driver , index , name )
  ```
  According driver of type, if PTY, then call pty_line_name; if TTY, then call tty_line_name.
  pty_line_name and tty_line_name object name is set , the result is stored in the name variable . They internally transferred using sprintf, formatted output name . Specific details of the code can be seen .
  For example , if the type is TTY, the name may end up as ("%s%d", driver->name, index + driver->name_base) .
  This name is very important.The name of the node under /dev/ and the name of the subdirectory under /sys/class/tty/ are all determined by it .
- IF (! (Driver-> flags & TTY_DRIVER_DYNAMIC_ALLOC)) , is called tty_cdev_add registered character device driver . Here and tty_register_driver inside i f (driver-> flags & TTY_DRIVER_DYNAMIC_ALLOC) chimed in . If tty_register_driver already registered a character device driver , then There is no need to register again here .
- Then allocate a struct device structure , assign values to dev->devt , dev->class, etc. , set the name of dev . Then call device_register(dev) to register the device.
  device_register is described in detail in the article "Device Model" , it will create device nodes and create subdirectories under class . For details, please see the corresponding chapter in "Device Model" .

proc_tty_register_driver & /proc/ tty/ drivers

proc is a Linux system in a sub-module , with sysfs somewhat similar , can be considered a virtual file system . If you ask proc register , after successful registration , the user space can, through / proc / xxx with your " proc Driver " Interactive .

Under normal circumstances , we will provide some debugging information to the user space through the system , so we define proc as a debugging technology and will introduce it in detail in the article ``Debugging Technology'' .

Here , we simply look at what the tty subsystem has registered with proc .

Header file : include/linux/tty.h
Implementation file : fs/proc/proc_tty.c

How is the /proc/tty/drivers file generated ?

This file will be created in the initialization function of proc_tty.c , the code is as follows :

```
/*
* Called by proc_root_ init( ) to initialize the /proc/tty subtree
*/
void __init proc_tty_init ( void )
{
    if (! proc_mkdir ( "tty" , NULL ))
        return ;
    proc_ mkdir ( "tty/ldisc" , NULL ); /* Preserved: it's userspace visible */
    /*
     * /proc/tty/driver/serial reveals the exact character counts for
     * serial links which is just too easy to abuse for inferring
     * password lengths and inter-keystroke timings during password
     * entry .
     */
    proc_tty_driver = proc_mkdir_ mode ( "tty/driver" , S_IRUSR | S_IXUSR , NULL );
    proc_ create ( "tty/ldiscs" , 0 , NULL , & tty_ldiscs_proc_fops );
    proc_ create ( " tty/drivers " , 0 , NULL , & proc_tty_drivers_operations );
}
```

➢✌♦ mentioned above , we can check how many tty drivers are registered in the system through cat /proc/tty/drivers .
How to achieve it ? Cat operation will eventually be mapped to proc_tty_drivers_operations , the operations will eventually scan tty_drivers the **global head of the list** of all under the driver, then their feedback to the user space .

proc_tty_register_driver

This API will be called in tty_register_driver . The code details of this API are as follows :

```
/*
* This function is called by tty_register_ driver( ) to handle
* registering the driver's /proc handler into /proc/ tty/ driver/<foo>
*/
void proc_tty_register_driver ( struct tty_driver * driver )
{
    struct proc_dir_entry * ent ;

    if (! driver -> driver_name || driver -> proc_entry ||
        ! driver -> ops -> proc_fops )
        return ;

    ent = proc_create_data ( driver -> driver_name , 0 , proc_tty_driver ,
                   driver -> ops -> proc_fops , driver );
    driver -> proc_entry = ent ;
}
```

- If tty_driver structure defines the OPS-> proc_fops , will be in / proc / tty / driver / create a file directory , file name is driver-> driver_name. We can cat the file , in order to obtain the necessary information . The cat operation will eventually be mapped to driver->ops->proc_fops .

void *The role of driver_state

4. tty core

In the previous section, we introduced how to register a terminal driver with the tty driver subsystem .
After the driver is successfully registered , the user space can interact with the driver through the interface provided by the tty core subsystem .
In this section , we from the perspective of the user space , take a look at tty core internal logic subsystem .

4.1 Introduction

As mentioned earlier , all terminal devices , from the perspective of user space , are character device drivers .
Registered tty_driver time , tty_register_driver calls tty_cdev_add to register a character device driver , and then tty_register_device will create device nodes .

tty_cdev_add when registering character device driver , using ops is drivers / tty / tty_io.c implemented struct the file_operations tty_fops . user space open / read other operations , will eventually be mapped totty_fops on .

In addition , the introduction tty_driver when this structure , we also mentioned tty_struct , tty_port , ktermios these structures .

We classify the above data structures into the tty core subsystem , and we will introduce them in detail in this section .

4.2 Main data structure

tty_struct

Regarding tty_struct and ktermios, first look at an explanation in the official code :
 * Where all of the state associated with a tty is kept while the tty
* is open. Since the termios state should be kept even if the tty
* has been closed — for things like the baud rate, etc — it is
* not stored here, but rather a pointer to the real state is stored
* here

tty_struct is used to represent a tty device is open state after , when the device is close later , this structure is disappeared . What is it tty_driver difference .
termios in tty_driver the time of registration , will have an initial value of . tty device is open after , you can modify the termios values . device is close after , it will not disappear . For example : Suppose we open a tty device , Then set its baud rate to 9600.If we close the device and then reopen it , the baud rate remains unchanged , which is still 9600.

Header file : include/linux/tty.h

struct tty_ struct	Comment
int magic	magic number for this structure
struct kref kref	Reference count
struct device *dev	A device will be created in tty_register_device , where dev points to the c reated device
struct tty_driver *driver	Corresponding tty_driver
const struct tty_operations *ops	The tty_operations corresponding to tty_driver. You should be able to access it directly through driver->ops , why should it be mentioned separately here ? In tty device is open time , will driver-> ops assigned to tty_struct-> ops. When needed , can be tty_struct-> ops reassigned without having to chan ge driver-> ops.
int index	A tty_driver may correspond tty_driver-> num one device . Here 0 <= index <= tty_driver->num
struct ld_semaphore ldisc_sem; struct tty_ldisc *ldisc	ldisc points to the tty line discipline corresponding to the tty_struct . ldisc_sem is a mutex , for mutual exclusion of ldisc access . For example, suppose we want to change tty_struct->ldisc, we need to ac quire the lock ldisc_sem first

struct mutex atomic_write_lock;
struct mutex legacy_mutex;
struct mutex throttle_mutex;
struct rw_semaphore termios_rwsem;
struct mutex winsize_mutex;
spinlock_t ctrl_lock;
spinlock_t flow_lock;
	It defines the various locks , for exclusive access .
	We introduced various locking mechanisms in the article ``Competition a nd Blocking" , and you can view the original text for details .

/* Termios values are protected by the termios rwse m */ struct ktermios termios, termios_locked; struct termiox *termiox; /* May be NULL for unsupp orted */	The ktermios corresponding to the tty_struct
char name[64]	The tty_struct 's name, its value is s printf(p, “ %s%d “, driver->name , index + driver->name_base)
struct pid *pgrp; /* Protected by ctrl lock */ struct pid *session;	pid related

unsigned long flags	The flag corresponding to the tty_struct , one of the following subtypes Attention flags changes must SET_BIT / clear_bit such atomic operations , in order to avoid various problems caused by concurrent access include/linux/tty .h #define TTY_THROTTLED 0 /* Call unthrottle() at threshold min */ #define TTY_IO_ERROR 1 /* Cause an I/O error (may be no ldisc too) */ #define TTY_OTHER_CLOSED 2 /* Other side (if any) has closed */ #define TTY_EXCLUSIVE 3 /* Exclusive open mode */ #define TTY_DEBUG 4 /* Debugging */ #define TTY_DO_WRITE_WAKEUP 5 /* Call write_wakeup after queuing new */ #define TTY_OTHER_DONE 6 /* Closed pty has completed input processi ng */ #define TTY_LDISC_OPEN 11 /* Line discipline is open */ #define TTY_PTY_LOCK 16 /* pty private */ #define TTY_NO_WRITE_SPLIT 17 /* Preserve write boundaries to driver */ #define TTY_HUPPED 18 /* Post driver->hangup() */ #define TTY_LDISC_HALTED 22 /* Line discipline is halted */
int count	In user space , can a tty device node open times , repeatedly open in kern el space corresponds to only . 1 th tty_struct. count represents open times . In the open / re-open when ++, in close wh en -
struct winsize winsize; /* winsize_mutex */	The size of the window corresponding to this tty_struct . Note that this parameter is not ktermios, in tty_struct disappear when , it will disappear . Why do not you take it out it alone ? Because almost ever y application in open time , will be set winsize, so there is no need to save .
unsigned long stopped:1, /* flow_lock */ flow_stopped:1, unused:BITS_PER_LONG – 2;	The definition of some variables , the specific purpose is not yet clear
int hw_stopped	The specific purpose is not yet clear
unsigned long ctrl_status:8, /* ctrl_lock */ packet:1, unused_ctrl:BITS_PER_LONG – 9;	The definition of some variables , the specific purpose is not yet clear
unsigned int receive_room; /* Bytes free for queue */	Usually there will be a buffer for storing user space to over the data , this parameter should refer to the buffer remaining size.
int flow_change	The specific purpose is not yet clear
struct tty_struct *link	The specific purpose is not yet clear
struct fasync_struct *fasync	Asynchronous notification mechanism is related . Example, the user space may lose some data down , but no longer there w aiting , can continue to execute other programs . After this data has been t ransmitted to the kernel , notify the user space .
int alt_speed; /* For magic substitution of 38400 bps */	A variable , the specific meaning is not yet clear

wait_queue_head_t write_wait;
wait_queue_head_t read_wait;
	Two waiting queues . The waiting queue is described in detail in the articl e "Competition and Blocking"

struct work_struct hangup_work	A work queue
void *disc_data	Related to tty line discipline
void *driver_data	
struct list_head tty_files	In the article ``Character Device Drivers'', we said that for each open ope ration , the kernel space will create a corresponding struct file. However, for multiple open operations of the tty device , the kernel will only have one struct tty_struct. The tty_files here is a linked list header , and all struct files will be linked under this linked list header
int closing	
unsigned char *write_buf	
int write_cnt	
/* If the tty has a pending do_SAK, queue it here – a kpm */ struct work_struct SAK_work;	Work queue , see notes for specific purposes
struct tty_port *port	Point to a tty_port

ktermios

K termios is mainly used to configure tty devices in user space , configure its baud rate , parity, and so on .

Header file : include / uapi / asm-generic / termbits.h

For specific details and the meaning of each field , you can directly view the source code , so I wo n't talk about it here .

```
struct ktermios {
    tcflag_t c_iflag ; /* input mode flags */
    tcflag_t c_oflag ; /* output mode flags */
    tcflag_t c_cflag ; /* control mode flags */
    tcflag_t c_lflag ; /* local mode flags */
    cc_t c_line ; /* line discipline */
    cc_t c_ cc [ NCCS ]; /* control characters */
    speed_t c_ispeed ; /* input speed */
    speed_t c_ospeed ; /* output speed */
};
```

tty_port

Looking back at the structure of tty_operations , you will find that it only has a write function , but no read function . When the user space wants to send data , the write function will be called , and it will operate the hardware (such as the serial port) to send the data . But when the hardware When data is received , how is it passed to user space ?

tty_port role in this , you can simply understand it as a buffer. When the hardware receives the data , it will receive the data stored in the tty_port the buffer inside , then from the user space will tty_port the buffer read data .

Before proceeding to the following article , let us first sort out the relationship between /dev/ device node , tty_driver, tty_struct, tty_port .

- a tty_driver corresponding to (tty_driver-> NUM) a device node
- a device node is in the open after , corresponding to a tty_struct
- a tty_struct corresponds to a tty_port

The main code and data structure related to tty_port are as follows :

Header file : include/linux/tty.h
The details of the struct tty_port data structure will not be introduced in detail , just put the code here .

```
struct tty_port_operations {
    /* Return 1 if the carrier is raised */
    int (* carrier_raised )( struct tty_port * port );
    /* Control the DTR line */
    void (* dtr_rts )( struct tty_port * port , int raise );
    /* Called when the last close completes or a hangup finishes
       IFF the port was initialized. Do not use to free resources. Called
       under the port mutex to serialize against activate/shutdowns */
    void (* shutdown )( struct tty_port * port );
    /* Called under the port mutex from tty_port_open, serialized using
       the port mutex */
        /* FIXME: long term getting the tty argument *out* of this would be
           good for consoles */
    int (* activate )( struct tty_port * port , struct tty_struct * tty );
    /* Called on the final put of a port */
    void (* destruct )( struct tty_port * port );
};

struct tty_port {
    struct tty_bufhead buf ; /* Locked internally */
    struct tty_struct   * tty ; /* Back pointer */
    struct tty_struct   * itty ; /* internal back ptr */
    const struct tty_port_operations * ops ; /* Port operations */
    spinlock_t       lock ; /* Lock protecting tty field */
    int          blocked_open ; /* Waiting to open */
    int          count ; /* Usage count */
    wait_queue_head_t open_wait ; / * Open waiters */
    wait_queue_head_t close_wait ; /* Close waiters */
    wait_queue_head_t delta_msr_wait ; /* Modem status change */
    unsigned long        flags ; /* TTY flags ASY_*/
    unsigned char        console : 1 , /* port is a console */
                low_latency : 1 ; / * optional: tune for latency */
    struct mutex mutex ; /* Locking */
    struct mutex buf_mutex ; /* Buffer alloc lock */
    unsigned char * xmit_buf ; /* Optional buffer */
    unsigned int         close_delay ; /* Close port delay */
    unsigned int         closing_wait ; /* Delay for output */
    int          drain_delay ; /* Set to zero if no pure time
                              based drain is needed else
                              set to size of fifo */
    struct kref kref ; /* Ref counter */
};
```

Source file :

drivers/tty/tty_port.c: This C file provides some APIs for tty_port , including :

- void tty_port_init(struct tty_port *port) : The *port parameter points to a space that has been allocated , and this API is used to initialize this space
- tty_port_link_device : used to associate tty_port with tty_driver , that is, let tty_driver-> ports[index] = tty_port
- tty_port_register_device
- tty_port_register_device_attr
- …… .

drivers / tty / tty_buffer.c: the C file is provided an operation tty_port-> buf Some of the API, mainly of the buffer process , the C file corresponding to the file header mainly the include / Linux / tty_flip. H :

- void tty_buffer_init (struct * tty_port Port) : mainly used for the initial of tty_port-> buf , Note that no a buffer allocated space
- tty_buffer_request_room : it calls tty_buffer_alloc , to buffer allocated storage space
- tty_buffer_set_limit : set the size limit of the buffer
- tty_insert_flip_char : insert a character into the buffer
- tty_insert_flip_string : insert a string into the buffer
- tty_buffer_space_avail : get the remaining space of the buffer
- tty_flip_buffer_push : Move the data from tty_port->buf to tty line discipline . As mentioned earlier , the user space will read the data received by the hardware from the tty_port , which is actually read from the tty line discipline

4.3 Main API description

The tty core submodule does not seem to provide any interface to other submodules of the kernel . Its main function is to provide a character device driver interface to the user space . Therefore, in this section we will mainly look at the interface functions of these character device drivers .

Since the details of the implementation of these interface functions are too cumbersome , and our main work in the project is focused on the driver , we only need to have a general understanding of the tty core , so this section will only give a rough introduction to roughly clarify the code logic .

core tty character device driver provided , the most important is the following ops:

Code path : drivers/tty/tty_io.c

```
static const struct file_operations tty_fops = {
    . llseek     = no_llseek ,
    . read       = tty_read ,
    . write      = tty_write ,
    . poll       = tty_poll ,
    . unlocked_ioctl = tty_ioctl ,
    . compat_ioctl   = tty_compat_ioctl ,
    . open       = tty_open ,
    . release    = tty_release ,
    . fasync     = tty_fasync ,
};
```

We mainly analyze the functions of open/read/write/release .

open

When we open a tty device node in user space , the open function here will be called .

The main function of the open function is to create and initialize the tty_struct structure .
If a user repeatedly open space with a tty device node , open the function does not create a new tty_struct, will perform tty_reopen operation , in reopen inside , tty_struct-> count ++.
There is also a special case : the /dev/tty device node , remember its role ? If you perform an open operation on this device node , will the kernel space create a new tty_struct structure ? (The answer is no). and kernel space directly by (current-> signal-> tty) Gets already created tty_struct. remember the current it ? it is struct task_struct type of pointer , by current, we can get detailed information about processes .

Let's take a look at the code details of this function :

```
static int tty_open ( struct inode * inode , struct file * filp )
{
   struct tty_struct * tty ;
   struct tty_driver *driver = NULL;
   dev_t device = inode->i_rdev;

    ...

    tty = tty_open_current_tty ( device , filp );
    if (! tty ) {
        mutex_ lock ( & tty_mutex );
        driver = tty_lookup_driver ( device , filp , & noctty , & index );
        if ( IS_ERR ( driver )) {
            retval = PTR_ERR ( driver );
            goto err_unlock ;
        }

        /* check whether we're reopening an existing tty */
        tty = tty_driver_lookup_tty ( driver , inode , index );
        if ( IS_ERR ( tty )) {
            retval = PTR_ERR ( tty );
            goto err_unlock ;
        }

        if ( tty ) {
            mutex_ unlock ( & tty_mutex );
            tty_ lock ( tty );
            /* safe to drop the kref from tty_driver_lookup_ tty( ) */
            tty_kref_ put ( tty );
            retval = tty_reopen ( tty );
            if ( retval < 0 ) {
                tty_ unlock ( tty );
                tty = ERR_PTR ( retval );
            }
        } else { / * Returns with the tty_lock held for now */
            tty = tty_init_dev ( driver , index );
            mutex_ unlock ( & tty_mutex );
        }

        tty_driver_kref_ put ( driver );
    }

...

}
```

- tty_open_current_tty : When the open operation is performed on the /dev/tty node , this function will work , and it will call get_current_tty , get the created tty_struct from (current->signal->tty)
- tty_lookup_driver : found through device number corresponding tty_driver. Guessed implement logic it ? Also very simple , first of all we have known device number , and then all of tty_driver are mounted to the global head of the list tty_drivers below the , one by one scan head of the list below All tty_drivers mounted, compare the device number , you can find the corresponding tty_driver.
- tty_driver_lookup_tty : Check if tty_deriver-> ttys[idx] is NULL, if it is not empty , it proves to open the same tty device node repeatedly , just execute the `tty_reopen` operation directly , without creating a new `tty_struct`.
- tty_init_dev : Create a tty_struct structure . It will call alloc_tty_struct to allocate and initialize tty_struct.

read

When we perform a read operation in user space , the read function here will be called . The main purpose of read is to return data from kernel space to user space .
As we said earlier , when our hardware (such as the serial port) receives data , will be stored in the tty_portInside tty line discipline . The read operation here is to get data from tty_ldisc and then return it to user space .

Post it directly bar code :

```
static ssize_t tty_read ( struct file * file , char __user * buf , size_t count ,
            loff_t * ppos )
{
    int i ;
    struct inode * inode = file_inode ( file );
    struct tty_struct * tty = file_tty ( file );
    struct tty_ldisc * ld ;

    if ( tty_paranoia_check ( tty , inode , "tty_read" ))
        return - EIO ;
    if (! tty || ( test_bit ( TTY_IO_ERROR , & tty -> flags )))
        return - EIO ;

    /* We want to wait for the line discipline to sort out in this
       situation */
    ld = tty_ldisc_ref_wait ( tty );
    if ( ld -> ops -> read )
        i = ld -> ops -> read ( tty , file , buf , count );
    else
        i = - EIO ;
    tty_ldisc_ DEREF ( LD );

    if ( i > 0 )
        tty_update_ time ( & inode -> i_atime );

    return i ;
}
```
The code is very simple , call ld->ops->read to read the data . This is simple because the main logic is processed in the tty line discipline , we will describe it in detail when we introduce this section .

write

When we perform a write operation in user space , the write function here will be called.The main purpose of write is to transfer data from user space to kernel space , and then send it out through hardware .

write logic is very simple , after receiving the user data space , call discipline tty line to send data , tty line discipline will call tty_driver-> ops-> write a function of the data sent by the hardware .

The code is as follows :

```
static ssize_t tty_write ( struct file * file , const char __user * buf ,
                        size_t count , loff_t * ppos )
{
    struct tty_struct * tty = file_tty ( file );
    struct tty_ldisc * ld ;
    ssize_t ret ;

    ...

    ld = tty_ldisc_ref_wait ( tty );
    if (! ld -> ops -> write )
```

```
        ret = - EIO ;
    else
        ret = do_tty_write ( ld -> ops -> write , tty , file , buf , count );
    tty_ldisc_ DEREF ( LD );
    return ret ;
}
```
do_tty_write will call ld->ops->write, and ld->ops->write will eventually call the tty_driver->ops->write function to send the data through the hardware .

release

When the user space performs a close operation, it will cause the release function to be called .
However, not every close operation will cause the release function to be called . If the device node is opened multiple times , only the last close operation will cause the release function to be called . This logic is controlled by the character device driver . You can look back for the specific details. See the article "Character Device Drivers" .

So once here the release function is executed , on behalf of all the open operations are already close up . Release function which will release tty_struct this structure .

The code will not be posted .

5. tty line discipline

5.1 Introduction

The role of tty line discipline (hereinafter referred to as ldis) has been roughly introduced in Chapter 2 , and now we will come back to its position in the tty subsystem .
In the tty subsystem , the tty core is responsible for interacting with the user space in the form of a character device driver ; the tty driver is responsible for operating the underlying hardware , and sending and receiving data through the hardware in a bit- by - bit manner .
When the user space wants to send data, it will first pass the data to the tty core, and the tty core will transfer the data to ldis; ldis can do some encapsulation of the data at this time (usually the encapsulation of the header on the software protocol), ldis encapsulation After the data is completed , the data will be handed over to the tty driver to send out through the hardware ; the tty driver does not care about the specific content of the data , it only sends out the data one bit by one .
Conversely , when after the hardware has received data , tty driver in data storage will tty_port-> buffer in , and then at some point , in the tty_port-> buffer move data into LDIS in ; LDIS received data , Some decapsulation processing is performed on the data (usually the decapsulation of the header on the software protocol); when the user space calls the read interface to obtain the data , the tty core will fetch the data from the ldis and then hand it to the user space .

Taken together , the role of ldis is to process data at the software protocol level , which is not related to specific hardware , and is mainly related to the protocol . Therefore, a ldis is used to implement a protocol , such as PPP or Bluetooth .

After understanding the status of ldis , let's analyze the architecture at the software level .
First of all , you can ldis understood as a pool , you can ldis provide sub-module API add to this pool (registered) ldis. This pool will store a lot more ldis, each ldis correspond to implement a protocol , when we want to use ldis time , will select a pool from the inside ldis.

So , the next question is who will be responsible for selecting from this pool inside ldis it ? The answer is tty core, because it will pass ldis sending data , from ldis receive data , it must know which with ldis.

There is also a problem , tty core is in when selecting ldis it ? By default , when a user space of the calling open when operating open a device node , tty core of the open function will be called , in the tty core of the open function inside , will be from LDIS select a pond inside ldis. Further , the user can also space ioctl specified tty core using one ldis.

The problem continues , tty core know which use ldis up , then it should take this ldis stored , for later use at any time . Where is it stored ? You should already know , the answer is tty_struct-> ldisc , the tty core of the open function Inside , the tty_struct structure will be created , then the default ldis will be selected , and the obtained ldis will be stored in tty_struct-> ldisc .

For this design the kernel , so what's to say the ?
Can only say that the design of very good! The one hand, reflects the software hierarchical thinking , different sub-modules responsible for different things ; on the other hand , before the module with low coupling module , tty core and ldis and are not necessarily linked , it you can choose any one ldis, user space provided tty core used ldis.
Such logical design , for example, a user wants to send a space character A, the user can select the space via Bluetooth (wherein a LDIS) sent , you can be selected by other means (other LDIS) sent .

5.2 Main data structure

tty_ ldiscs[NR_LDISCS]

Implementation file : drivers / tty /tty_ldisc.c

```
/* Line disc dispatch table */
static struct tty_ldisc_ops * tty_ldiscs [ NR_LDISCS ];
```

It is what we said earlier that pool , in fact, a global array of structures . The size of the pool is the size of the array , which is NR_LDISCS .

When we use the tty ldis sub-module provides API to register a ldis time , to be registered ldis is stored inside the array .

NR_LDISCS in the include / uapi / Linux /tty.h defined . Uapi description will refer to this header the user space file , the user space object of this reference is to the first document set tty core which use LDIS, when the set , specify a user space ID can be , this ID corresponds to an element of the array .

```
#define NR_LDISCS 30


/* line disciplines */
```

```
#define N_TTY 0
#define N_SLIP 1
#define N_MOUSE 2
#define N_PPP 3
#define N_STRIP 4
#define N_AX25 5
#define N_X25 6   /* X.25 async */
#define N_6PACK 7
#define N_MASC 8   /* Reserved for Mobitex module <kaz@cafe.net> */
#define N_R3964 9   /* Reserved for Simatic R3964 module */
#define N_PROFIBUS_ FDL 10 /* Reserved for Profibus */
#define N_IRDA      11  / * Linux IrDa - http://irda.sourceforge.net/ */
#define N_ SMSBLOCK 12 /* SMS block mode - for talking to GSM data */
                /* cards about SMS messages */
#define N_HDLC      13  / * synchronous HDLC */
#define N_SYNC_ PPP 14 /* synchronous PPP */
#define N_HCI       15  / * Bluetooth HCI UART */
#define N_GIGASET_ M101 16 /* Siemens Gigaset M101 serial DECT adapter */
#define N_SLCAN     17  / * Serial / USB serial CAN Adaptors */
#define N_PPS       18  / * Pulse per Second */
#define N_V253      19  / * Codec control over voice modem */
#define N_CAIF 20      /* CAIF protocol for talking to modems */
#define N_GSM0710   21  / * GSM 0710 Mux */
#define N_TI_WL     22  / * for TI's WL BT, FM, GPS combo chips */
#define N_TRACESINK 23  / * Trace data routing for MIPI P1149.7 */
#define N_TRACEROUTER   24  / * Trace data routing for MIPI P1149.7 */
```

- #define NR_LDISCS 30 , illustrate the pond can store up to 30 Ge ldis
- Further kernel has most of predetermined ldis corresponding ID, for example #define N_PPP 3 , described implement PPP this protocol ldis corresponding ID is 3.

tty_ldisc

If we want to write yourself a ldis, you must implement a tty_ldisc _ops structure , then the tty ldis registered sub-modules . Registration process , tty ldis sub-module creates a tty_ldisc structure , and this structure into tty_ldiscs [NR_LDISCS] A position in this array .

Header file : include / linux / tty_ldisc.h

```
struct tty_ldisc {
    struct tty_ldisc_ops * ops ;
    struct tty_struct * tty ;
};
```

This structure is very simple , *tty is used to point to the tty_struct structure , mainly to implement the `tty_ldisc_ops` structure .

tty_ldisc_ops

Header file : include / linux / tty_ldisc.h

```
struct tty_ldisc_ops {
    int magic ;
    char * name ;
    int num ;
    int flags ;

    /*
     * The following routines are called from above.
     */
    int (* open )( struct tty_struct *);
    void (* close )( struct tty_struct *);
    void (* flush_buffer )( struct tty_struct * tty );
    ssize_t (* chars_in_buffer ) ( struct tty_struct * TTY );
    ssize_t (* read )( struct tty_struct * tty , struct file * file ,
            unsigned char __user * buf , size_t nr );
    ssize_t (* write )( struct tty_struct * tty , struct file * file ,
             const unsigned char * buf , size_t nr );
    int (* ioctl )( struct tty_struct * tty , struct file * file ,
             unsigned int cmd , unsigned long arg );
    long (* compat_ioctl )( struct tty_struct * tty , struct file * file ,
                unsigned int cmd , unsigned long arg );
    void (* set_termios )( struct tty_struct * tty , struct ktermios * old );
    unsigned int (* poll )( struct tty_struct *, struct file *,
                 struct poll_table_struct *);
    int (* hangup )( struct tty_struct * tty );

    /*
     * The following routines are called from below.
     */
    void (* receive_buf )( struct tty_struct *, const unsigned char * cp ,
                   char * fp , int count );
    void (* write_wakeup )( struct tty_struct *);
    void (* dcd_change )( struct tty_struct *, unsigned int );
    void (* fasync )( struct tty_struct * tty , int on );
    int (* receive_buf2 )( struct tty_struct *, const unsigned char * cp ,
                char * fp , int count );

    struct   module * owner ;

    int refcount ;
};
```

The meaning of each interface function has been explained in tty_ldisc.h , and we will post it directly :

/*

* This structure defines the interface between the tty line discipline

* implementation and the tty routines. The following routines can be
* defined; unless noted otherwise, they are optional, and can be
* filled in with a null pointer.
*
* int (* **open**)(struct tty_struct *);
*
* This function is called when the line discipline is associated
* with the tty. The line discipline can use this as an
* opportunity to initialize any state needed by the ldisc routines.
*
* void (* **close**)(struct tty_struct *);
*
* This function is called when the line discipline is being
* shutdown , either because the tty is being closed or because
* the tty is being changed to use a new line discipline
*
* void (* **flush_buffer**)(struct tty_struct *tty);
*
* This function instructs the line discipline to clear its
* buffers of any input characters it may have queued to be
* delivered to the user mode process.
*
* **ssize_t** (* **chars_in_buffer**)(struct tty_struct *tty);
*
* This function returns the number of input characters the line
* discipline may have queued up to be delivered to the user mode
* process .
*
* ssize_t (* **read**)(struct tty_struct * tty, struct file * file,
*   unsigned char * buf, size_t nr);
*
* This function is called when the user requests to read from
* the tty. The line discipline will return whatever characters
* it has buffered up for the user. If this function is not
* defined, the user will receive an EIO error.
*
* ssize_t (* **write**)(struct tty_struct * tty, struct file * file,
*   const unsigned char * buf, size_t nr);
*
* This function is called when the user requests to write to the
* tty . The line discipline will deliver the characters to the
* low-level tty device for transmission, optionally performing
* some processing on the characters first. If this function is
* not defined, the user will receive an EIO error.
*
* int (* **ioctl**)(struct tty_struct * tty, struct file * file,
* unsigned int cmd, unsigned long arg);

```
*
* This function is called when the user requests an ioctl which
* is not handled by the tty layer or the low-level tty driver.
* It is intended for ioctls which affect line discpline
* operation . Note that the search order for ioctls is (1) tty
* layer , (2) tty low-level driver, (3) line discpline. So a
* low-level driver can “grab” an ioctl request before the line
* discpline has a chance to see it.
*
* long (* compat_ioctl )(struct tty_struct * tty, struct file * file,
*        unsigned int cmd, unsigned long arg);
*
* Process ioctl calls from 32-bit process on 64-bit system
*
* void (* set_termios )( struct tty_struct *tty, struct ktermios * old);
*
* This function notifies the line discpline that a change has
* been made to the termios structure.
*
* int (* poll )(struct tty_struct * tty, struct file * file,
*   poll_table *wait);
*
* This function is called when a user attempts to select/poll on a
* tty device. It is solely the responsibility of the line
* discipline to handle poll requests.
*
* void (* receive_buf )( struct tty_struct *, const unsigned char *cp,
*       char *fp, int count);
*
* This function is called by the low-level tty driver to send
* characters received by the hardware to the line discpline for
* processing. < cp > is a pointer to the buffer of input
* character received by the device. < fp > is a pointer to a
* pointer of flag bytes which indicate whether a character was
* received with a parity error, etc. <fp> may be NULL to indicate
* all data received is TTY_NORMAL.
*
* void (* write_wakeup )( struct tty_struct *);
*
* This function is called by the low-level tty driver to signal
* that line discpline should try to send more characters to the
* low-level driver for transmission. If the line discpline does
* not have any more data to send, it can just return. If the line
* discipline does have some data to send, please arise a tasklet
* or workqueue to do the real data transfer. Do not send data in
* this hook, it may leads to a deadlock.
*
```

```
* int (* hangup )(struct tty_struct *)
*
* Called on a hangup. Tells the discipline that it should
* cease I/O to the tty driver. Can sleep. The driver should
* seek to perform this action quickly but should wait until
* any pending driver I/O is completed.
*
* void (* fasync )( struct tty_struct *, int on)
*
* Notify line discipline when signal-driven I/O is enabled or
* disabled .
*
* void (* dcd_change )( struct tty_struct *tty, unsigned int status)
*
* Tells the discipline that the DCD pin has changed its status.
* Used exclusively by the N_PPS (Pulse-Per-Second) line discipline.
*
* int (* receive_buf2 )(struct tty_struct *, const unsigned char *cp,
* char *fp, int count);
*
* This function is called by the low-level tty driver to send
* characters received by the hardware to the line discpline for
* processing. < cp > is a pointer to the buffer of input
* character received by the device. < fp > is a pointer to a
* pointer of flag bytes which indicate whether a character was
* received with a parity error, etc. <fp> may be NULL to indicate
* all data received is TTY_NORMAL.
* If assigned, prefer this function for automatic flow control.
*/
```

5.3 Main API description

tty_register_ldisc

Header file : include/linux/tty.h
Implementation file : drivers / tty /tty_ldisc.c
int **tty_register_ldisc** (int disc, struct tty_ldisc_ops *new_ldisc)

- ➢ code logic is very simple , the new_ldisc stored tty_ldiscs array , the parameter disc Representative new_ldisc in the array ID.

tty_set_ldisc

Header file : include/linux/tty.h
Implementation file : drivers / tty /tty_ldisc.c
int **tty_set_ldisc** (struct tty_struct *tty, int ldisc)

- ➢ When the user space of the calling ioctl set ldis time , the function is called .

The logic function is also very simple , according to the parameter ldisc, selected from a pool inside the corresponding LDIS, then replace tty_struct-> ldisc.

tty_ldisc_N_TTY

tty_ldisc_N_TTY is not an API, it is the default ldis of the kernel system . This ldis does not perform additional processing on data , it is like a pipeline , connecting tty core and tty driver.
If the user space does not show which ldis is used for configuration , the default is tty_ldisc_N_TTY .

There are two problems here :

1. Who defines tty_ldisc_N_TTY and when will you add yourself to the pool ?
1. The code logic is how the tty_ldisc_N_TTY as the default ldis of ?

First look at the first question : **Who defines tty_ldisc_N_TTY and who registers** :

Implementation file : drivers / tty / n_tty.c

```
struct tty_ldisc_ops tty_ldisc_N_TTY = {
    . magic           = TTY_LDISC_MAGIC ,
    . name            = "n_tty" ,
    . open            = n_tty_open ,
    . close           = n_tty_close ,
    . flush_buffer    = n_tty_flush_buffer ,
    . chars_in_buffer = n_tty_chars_in_buffer ,
    . read            = n_tty_read ,
    . write           = n_tty_write ,
    . ioctl           = n_tty_ioctl ,
    . set_termios     = n_tty_set_termios ,
    . poll            = n_tty_poll ,
    . receive_buf     = n_tty_receive_buf ,
    . write_wakeup    = n_tty_write_wakeup ,
    . fasync      = n_tty_fasync ,
    . receive_buf2    = n_tty_receive_buf2 ,
};
```

Registration function : drivers/tty/tty_ldisc.c

```
void tty_ldisc_begin ( void )
{
    /* Setup the default TTY line discipline. */
    ( void ) tty_register_ldisc ( N_TTY , & tty_ldisc_N_TTY );
}
```

- The `void __init console_init(void)` function in `drivers/tty/tty_io.c` will call `tty_ldisc_begin` here , and then `tty_ldisc_begin` calls `tty_register_ldisc` to add an `ldis` to the pool

Let's look at the second question : **how is put on the code logic tty_ldisc_N_TTY as the default ldis of** :

We know that when the user space performs an open operation on a tty node , the corresponding open function in the tty core will be called .

The open function of tty core will create a tty_struct structure , and will select the default ldis at this time , and assign this ldis to tty_struct->ldisc.

The specific code flow is :
drivers / tty /tty_ io.c: tty_open -> tty_init_dev -> alloc_tty_struct -> tty_ldisc_init .

tty_ldisc_init is defined in drivers / tty / tty_ldisc.c , the code is as follows :

```
void tty_ldisc_init ( struct tty_struct * tty )
{
    struct tty_ldisc * ld = tty_ldisc_get ( tty , N_TTY );
    if ( IS_ERR ( ld ))
        panic ( "n_tty: init_tty" );
    tty -> ldisc = ld ;
}
```

- tty_ldisc_get(tty, N_TTY) : According to the parameter N_TTY (actually an ID, corresponding to an element in the array), get the corresponding ldis, and then assign it to tty->ldisc .

6. serial core

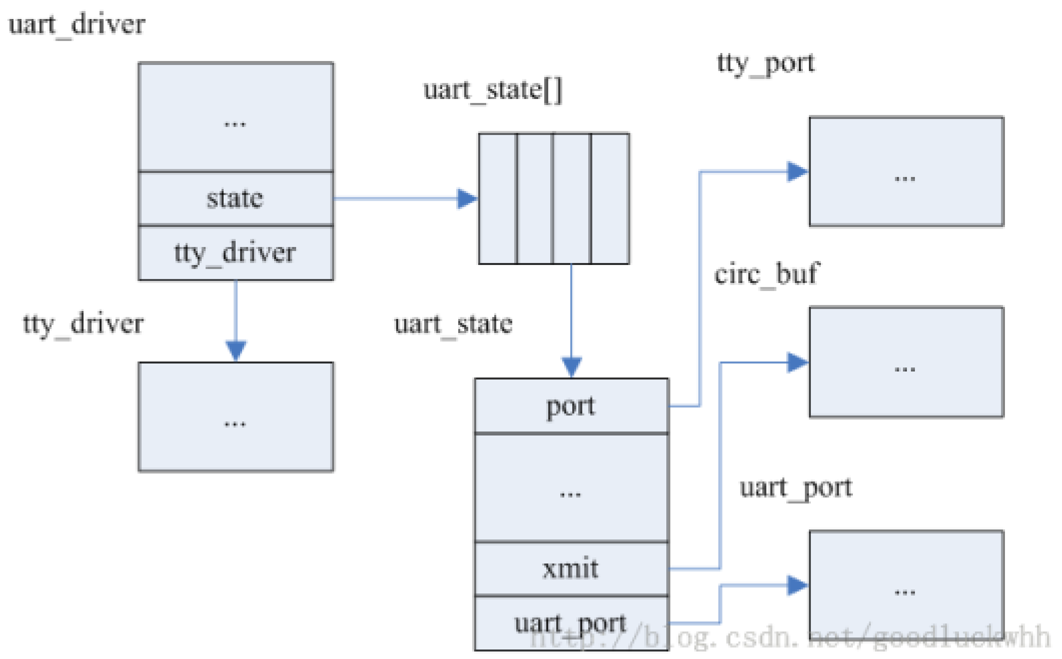
6.1 Introduction

serial core is mainly for the serial driver . the vast majority of ARM 's CPU, has a serial port controller , the CPU chip inside the manual , usually called UART or USART.

Why does the serial core exist ? The main purpose is to make it easier to write serial port drivers .
When you need to write a serial port driver , you only need to register with the serial core subsystem , and the serial core will help you register with the tty driver subsystem . Of course, you can also directly register a serial driver with the tty driver subsystem , so It is equivalent to bypassing the serial core, which is generally not recommended .

The serial core also involves several data structures of its own . In order to clarify the meaning of these data structures , let's take a look at the characteristics of the serial port on the hardware:
A general ARM 's CPU will have more on UART / USART controller , they are commonly called chip handbook on the USART1, USART2, USART3 ...
In Serial core inside , with a struct uart_driver structure representative of a CPU of all USART control ; with a struct uart_state structure representative of CPU on a specific controller (e.g. the USART1) .

Let's first come to a diagram of the data structure , and then we will clarify the correspondence between these data structures :



A lot of data structures ... , do n't worry , let's analyze it slowly :

- a CPU (assuming there are N number USART control) corresponding to a uart_driver, also corresponds to a tty_driver.
  Where uart_driver-> nr = tty_driver->num = N.
- a USART controller corresponds to a uart_state, also corresponds to a tty_port, also corresponds to a uart_port.
  tty_driver-> major + tty_driver-> minor_start defines the starting device number , a controller also corresponds to a device number , and it also corresponds to a node under /dev/ .

After understanding the corresponding relationship described above , it will be much easier to look at the following introduction to the data structure .

6.2 Main data structure

uart_driver

Header file : include/linux/ serial_core.h

struct uart _driver	Comment
struct module *owner	
const char *driver_name	It will eventually be assigned to tty_driver-> driver_name . If you forget t he role of this field , please look back at the introduction of the tty_driver data structure
const char *dev_name	Eventually it will be assigned to tty_driver-> name . If you forget the role of this field , please look back at the introduction of the tty_driver data str ucture
int major	Will eventually be assigned to tty_driver->major
int minor	It will eventually be assigned to tty_driver-> minor_start
int nr	Will eventually be assigned to tty_driver->num
struct console *cons	Not yet clear about the specific role

/* * these are private; the low level driver should not * touch these; they should be initialised to NULL */ struct uart_state *state	Pay attention to the code comments , when we intend to implement a uart _driver structure , this field should be NULL. The serial core will be responsible for helping us allocate / release uart_st ate space . Note , a uart_driver under can " hook " more uart_state
struct tty_driver *tty_driver	Similarly , this field should be set to NULL The serial core will be responsible for calling alloc_tty_driver to create a tty_driver structure .

uart_state

Header file : include/linux/ serial_core.h
uart_state, as mentioned earlier , it corresponds to a specific controller on the CPU , such as USART1.

struct uart _ state	**Comment**
struct tty_port port	A uart_state corresponds to a tty_port, which means that each USART con troller corresponds to a tty_port. tty_port effect when the hardware data is received , the data storage tty_p ort inside , then transferred to ldis inside . Each USART controller , from a hardware point of view , can receive their data , so each controller requires a tty_port, otherwise the number of hard ware to a tty_port which store data there will be data confusion
struct uart_port *uart_port	A uart_state also corresponds to a uart_port. Are you thinking about the relationship between uart_port and tty_port ? Is uart_port similar to tty_port and is responsible for sending the data rec eived by the hardware to ldis? The answer is no , in uart_state in , have tty_port This structure is respon sible for the hardware data received forwarded to ldis, no need to design a uart_port the same thing to do . The main function of uart_port is to deal with the hardware controller . T here will be multiple USARTs on the CPU . Each USART is somewhat differ ent . At least the register address of each USART is different , and the inter rupt number is different . And different USARTs may also have different b aud rates, etc. , so that each hardware controller , requires a uart_port stru cture to describe it . Because uart_port is responsible with the specific deal hardware controll er , and therefore send data to hardware controllers and data acquisition h ardware controller , you have to go through uart_port. Uart_port data is r eceived after , will pass uart_port-> uart_state-> tty_port (see it , uart_p ort not available directly to tty_port, can also be seen from here , it tty_p ort belong to the same level , are subordinate uart_state) found tty_port, and then to store data tty_port in . The uart_port structure will be introduced in detail later

struct circ_buf xmit	A uart_state also corresponds to an xmit. xmit is a ring buffer , the size is generally (#define UART_XMIT_SIZE PAG E_SIZE) xmit action is invoked when a user space write data is transmitted to the s erial operation , tty subsystem calls through the layers , the data store will xmit inside , and then notifies the hardware starts sending data . The reason for using the buffer is that the serial port uses serial transmissi on , which is slower . If the buffer is used, the user space will not be block ed .
enum uart_pm_state pm_state	Sleep related , not analyzed yet

uart_port

A specific USART hardware controller corresponds to a uart_port.
uart_port is used to describe the hardware , the main role is responsible for dealing with the hardware controller , control hardware to send data , receive data from the hardware and the like .

Since uart_port is used to describe the hardware , what information should it describe the hardware ?
First of all , we must have the hardware register address , interrupt number and so on .
Secondly , there must be interface functions for operating the hardware , such as operating the hardware to send data , receiving data from the hardware, etc. , this part of the function is actually described by uart_ops .

Let's take a look at the data structure of uart_port , it is quite long , we only intercept a few important points :
Header file : include/linux/ serial_core.h

struct uart _ port	**Comment**
spinlock_t lock	/* port lock */
unsigned long iobase unsigned char __iomem *membase	Hardware register address , we can use the in/out and read/write method s provided by the I/O Port standard to access the hardware
resource_size_t mapbase resource_size_t mapsize	In addition to the above methods , we can also access hardware registers t hrough I/O Memory . In fact , I / O Port ways X86 popular architecture , the ARM in , more often use the I / O Memory way
unsigned int irq; /* irq number */ uns igned long irqflags; /* irq flags */ (*handle_irq)(struct uart_port *)	Interrupt related Interrupt number , interrupt flag Interrupt handler
(*handle_break)(struct uart_port *)	Handling the break signal
struct serial_rs485 rs485 (*rs 485_config)(struct uart_port *, struct serial_rs4 85 *rs485)	485 related 485 flag , (whether 485 is enabled, etc.) 485 configuration function
const struct uart_ops *ops	Point to uart_ops
…… ..	There are many , not Yi Yi elaborate

(*serial_in)(struct uart_port *, int) (*serial_out)(struct uart_port *, int, int) (*set_termios)(struct uart_port *, … ..) (*set_mctrl)(struct uart_port *, unsigned int) (*startup)(struct uart_port *port) (*shutdown)(struct uart_port *port) (*throttle)(struct uart_port *port) (*unth rottle)(struct uart_port *port) (*pm)(struct uart_port *, unsigned int state, ..)	The functions of these interface functions are duplicated in uart_ops . Bas ically, they use the functions in uart_ops , which are not used here.

uart_ops

uart_ops be uart_port substructure , which is used to describe how a USART hardware controller to send and receive data .

Header file : include/linux/ serial_core.h

```
/*
* This structure describes all the operations that can be done on the
* physical hardware. See Documentation/serial/driver for details.
*/
struct uart_ops {
    unsigned int (* tx_empty )( struct uart_port *);
    void (* set_mctrl )( struct uart_port *, unsigned int mctrl );
    unsigned int (* get_mctrl )( struct uart_port *);
    void (* stop_tx )( struct uart_port *);
    void (* start_tx )( struct uart_port *);
    void (* throttle )( struct uart_port *);
    void (* unthrottle )( struct uart_port *);
    void (* send_xchar ) ( struct uart_port *, char CH );
    void (* stop_rx )( struct uart_port *);
    void (* enable_ms )( struct uart_port *);
    void (* break_ctl )( struct uart_port *, int ctl );
    int (* startup )( struct uart_port *);
    void (* shutdown )( struct uart_port *);
    void (* flush_buffer )( struct uart_port *);
    void (* set_termios )( struct uart_port *, struct ktermios * new ,
                       struct ktermios * old );
    void (* set_ldisc )( struct uart_port *, struct ktermios *);
    void (* pm )( struct uart_port *, unsigned int state ,
                  unsigned int oldstate );

    /*
     * Return a string describing the type of the port
     */
    const char *(* type )( struct uart_port *);
    /*
     * Release IO and memory resources used by the port.
     * This includes iounmap if necessary.
     */
    void (* release_port )( struct uart_port *);

    /*
     * Request IO and memory resources used by the port.
     * This includes iomapping the port if necessary.
     */
    int (* request_port )( struct uart_port *);
    void (* config_port )( struct uart_port *, int );
    int (* verify_port )( struct uart_port *, struct serial_struct *);
    int (* ioctl )( struct uart_port *, unsigned int , unsigned long );
#ifdef CONFIG_CONSOLE_POLL
    int (* poll_init )( struct uart_port *);
    void (* poll_put_char )( struct uart_port *, unsigned char );
    int (* poll_get_char )( struct uart_port *);
#endif
};
```

ops call flow comb

From the beginning of Chapter 2 to the present , we have introduced many kinds of ops, let's sort out their calling process .

From user space to hardware :
User space -> tty core(file_operations tty_fops) -> tty line discipline(tty_ldisc_ops) -> tty driver(tty_operations) -> uart port(uart_ops)

From hardware to user space :
The above process can be reversed

6.3 Main API description

uart_register_driver

If we want to write a serial port driver , we need to prepare the uart_driver structure , and then call the API to register with the serial core .
The API will be further registered with the tty driver subsystem , let's analyze the implementation details of the API

Header file : include/linux/ serial_core.h
Implementation file : drivers / tty / serial /serial_core.c
int **uart_register_driver** (struct uart_driver *drv)

- drv->state = kzalloc(sizeof(struct uart_state) * drv->nr, GFP_KERNEL)

  Distribution drv-> nr months uart_state space

- normal = alloc_tty_driver (drv->nr)

  Apply for a tty_driver structure normal , and then initialize normal. Let's look at a few important points :

  - normal->flags = TTY_DRIVER_REAL_RAW | TTY_DRIVER_DYNAMIC_DEV

    flags set TTY_DRIVER_DYNAMIC_DEV flag , set the standard of what it means ? do not remember you can look back at the first 3 chapters . It means that when you call tty_register_driver the tty driver when registering subsystem , and does not create a character device driver , it will not Generate device nodes , and will not create subdirectories under /sys/class/tty .

    Why ? Recall the function of the uart_driver structure : it does not correspond to a specific USART hardware controller . It is like a container in which multiple hardware controllers (that is, uart_port) can be stored . The device node generally represents a certain one Hardware controllers , for example, /dev/ttyS0 corresponds to the USART0 controller , and /dev/ttyS1 corresponds to the USART1 controller , so the device node should not be created at this time , but should be created when uart_port is registered .

  - tty_set_operations (normal, &uart_ops)

    Set normal of tty_operations, is uart_ops. Uart_ops in serial_core.c implemented in . Tty line discipline is to this uart_ops deal .

- for (i = 0; i <drv->nr; i++) {

      … .
      tty_port_ init (port);
      port ->ops = &uart_port_ops;
  }

  Initialization tty_port, and set its ops to uart_port_ops , uart_port_ops also in serial_core.c implementations . When the hardware receives the data , that is, by this uart_port_ops dump the data tty line discipline inside

- retval = tty_register_driver (normal)

  Register with the tty driver subsystem

uart_add_one_port

For one specific USART hardware controller , when we are ready uart_port and uart_port-> uart_ops after the structure , you can call the API to serial core to add a sub- port of .

Header file : include/linux/ serial_core.h
Implementation file : drivers / tty / serial /serial_core.c
int **uart_add_one_port** (struct uart_driver *drv, struct uart_port *uport)

- As for uart_port After doing a series of initialization , eventually calls tty_port_register_device_attr , which is tty driver a subsystem API, the API will eventually lead to registration character device drivers , device nodes , create / sys / class / tty under Subdirectory

7. Drive design guidelines

In most cases , our driver development work is mainly to develop serial drivers , that is, register with the serial core subsystem . Because this section mainly introduces the main steps of writing serial drivers .

In Section 3.1 , we introduced how to write a tty driver.In fact, writing a serial port driver is very similar to writing a tty driver .
But when you write tty driver when , directly to the tty driver were registered subsystem . The preparation of serial driver , is the serial core registration subsystem , then serial core again tty driver subsystem to register .

. 7 .1 code file structure

To write a serial port driver , you must reference the serial_core.h header file .
File path : include/linux/ serial_core.h , this header file defines the data structure you need to implement and defines the API provided to you by the serial core subsystem .

At the same time , you may need to look at the Drivers / TTY / Serial /serial_core.c the C file , in order to assist in understanding the API significance .

7 .2 serial driver authoring process

1. **Register uart_driver**
   Define a struct uart_driver structure , assign values to the corresponding variables of the structure , and then call uart_register_driver to register .

   Consider a question ? When is uart_register_driver called , is it a platform_driver, and then uart_register_driver is called in the driver 's probe function ? The answer is no .
2. **Add a uart_port**
   Define a struct uart_port and struct uart_ops structure , and then call uart_add_one_port to register .

   Similarly , think about a question , when to call uart_add_one_port , will there be platform_driver?
   The answer is yes . Because uart_port is used to describe a particular USART controller . USART controller belonging to ARM CPU peripherals on chip , peripheral pieces are all mounted on the platform bus below , there is a paltform device described in this outer Set the hardware resources , there is also a platform driver to describe how to operate this peripheral .
   Therefore , for each USART controller , there will be a with respect to the platfrom device, which device share the same platfrom driver, the driver 's probe function inside , call uart_add_one_port the serial core add subsystem USART controller .

   But uart_register_driver not in the probe inside the function is called , because it does not correspond to one specific on-chip peripherals , will not be with them for the platform device. Generally, we define a module_init function , in module_init call the function inside uart_register_driver