EQUIPMENT MODEL

April 1, 2019 (Http://Www.Mysixue.Com/?P=62) JL
(Http://Www.Mysixue.Com/?Author=1)

1. Introduction to document structure

First of all , Chapter 2 will give the overall framework of the entire device model .
Then , Chapter 3 introduces sysfs, and understanding it helps us understand what the device model is doing .
Next , Chapters 4, 5, and 6 describe the basic concepts of the device model.Basically , these basics will not be seen in the driver development , but we must first understand them in order to better understand the subsequent content .
Then , Chapter 7 briefly introduces how some top-level directories under /sys/ are created, and you can browse them .
Next , Chapters 8, 9, 10, and 11 describe the 4 most important concepts in the device model :Bus, Class, Device, Driver .
The first 12 chapters describes the platform system , which is based on the Bus, Class, Device, Driver abstracted from a more upper layer of things , after understanding the contents of the front , look platfrom system will be very easy .
The last chapter describes the devres, it is easy to understand , but also very practical . Teach you in driving the development process , how eliminating " GOTO release resource troubles " .

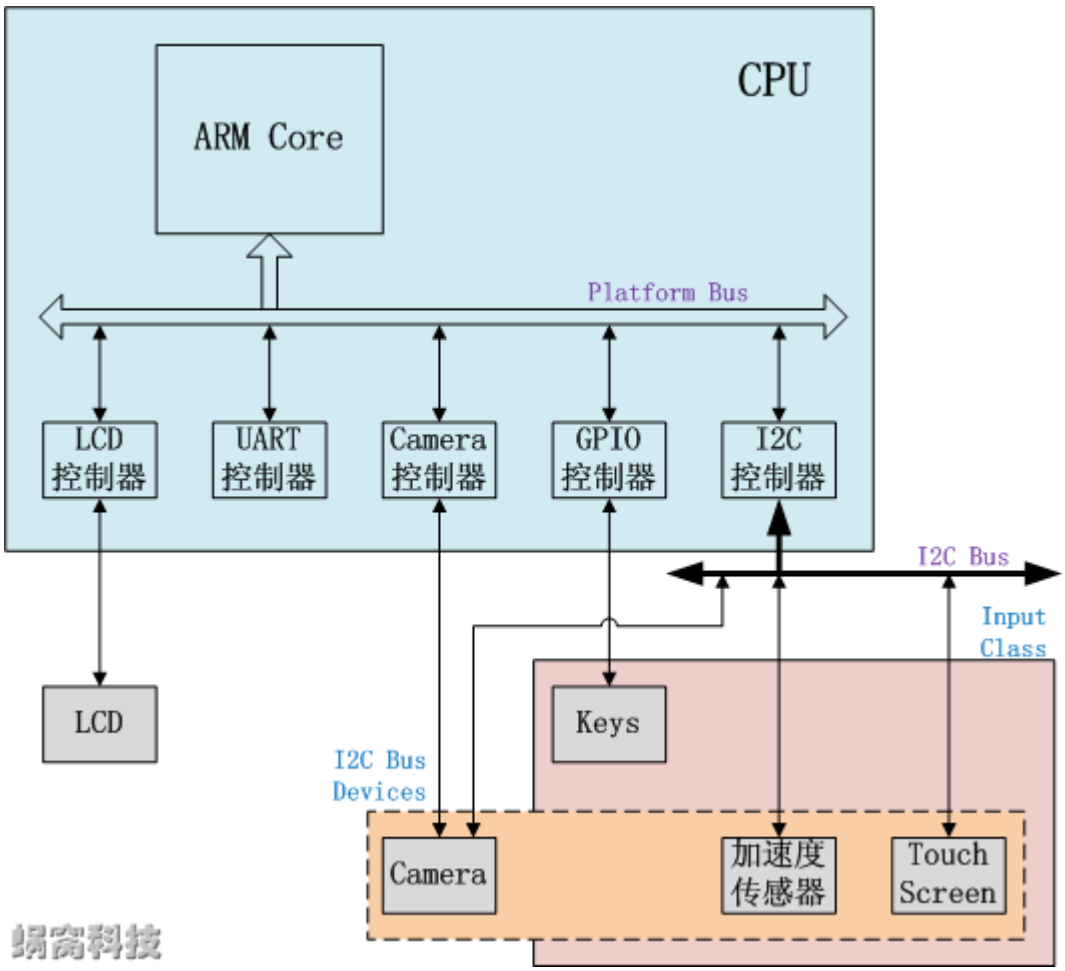
2. Architecture Introduction

Since Linux supports almost all hardware devices with different functions in the world (this is the advantage of Linux) , half of the code in the Linux kernel is device drivers , and with the rapid upgrade of hardware , the amount of code for device drivers is also Rapid growth . In order to reduce the complexity of Linux driver development caused by device diversity , as well as device hot plug processing, power management, etc. , the Linux kernel proposes the concept of device model (also known as Driver Model) . The device model combines hardware devices induction, classification , and a standard set of abstract data structures and interfaces . driven development , simplifies the filling and to implement data structures defined in the kernel .

2.1 The basic concept of the equipment model

Concepts of Bus, Class, Device and Device Driver

The following figure is an example of a common hardware topology in embedded systems:



The hardware topology describes three of the four important concepts in the Linux device model: Bus , Class and Device (the fourth is Device Driver , which will be described later) :

- Bus (bus)

  Linux believes that the bus is a channel for information interaction between the CPU and one or more devices . In order to facilitate the abstraction of the device model , all devices should be connected to the bus

- Class (classification)

  In the Linux device model , Class object-oriented programming concept is very similar to that in Class (class) , it is primarily a collection of functions or devices with similar properties , so that you can abstract a plurality of devices can be shared between the data structures and interface functions . Accordingly dependent same Class driver device , there is no need to repeat the definition of these common resources , directly from the Class inheritance to the

- Device (device)

  Abstract all hardware devices in the system , describing its name, attributes, subordinate Bus , subordinate Class and other information

- Device Driver (driver)

  Linux device model with Driver Driver abstract hardware , which includes device initialization, the associated power management interface . The Linux driver development kernel , basically around the abstractions (interface function to achieve specified)

3. sysfs

Quickly experience what sysfs is

A very perceptual realization is , just find a Linux machine or board , and then see if there is a folder called sys under its root directory (/) ? Yes , the /sys/ directory is the sysfs mentioned here .

What is the use of sysfs

ls /sys/, you will see several familiar folders , bus, class, devices. Are these folders the same as the conceptual names in the device model . Guess whether sysfs has anything to do with the device model ? Yes , sysfs is closely related to the device model .

The role of sysfs is mainly reflected in two aspects . From the perspective of kernel and user space , these two aspects are reflected in :

- sysfs to **show something to the user kernel space** : it shows the various components of the equipment module **level** relations , the hierarchy is a tree structure . Top-level directories , subdirectories , subdirectories , property files . User Space You can use ls to read these directories , or use cat to read the property files .
- sysfs can **get some information from user space and pass it to the kernel** : There may be some files in a certain directory of sysfs . We call this file a **property** file . User space can modify these property files . After the files are modified , sysfs can pass Some mechanism informs the kernel .

  Note: Remember the article in the kernel module , we have introduced , the kernel module can have module parameters , we can also sysfs read / modify module parameters . However, the kernel module will not receive a notification parameters are modified , Modules only take the initiative to read parameters , the previous values do the right ratio , before we know whether the parameters are modified .

  There will be a question here . The module parameters are also modified through sysfs . Why did you not receive the modification notification ? The specific reasons did not go into detail . However, we will introduce in the following chapters how sysfs notifies the kernel that a certain attribute file has been modified a . after understanding this mechanism , if interested in tracking the above questions , you can read the details of its own source code module parameters .

Detailed understanding of sysfs

sysfs is a RAM- based virtual file system

Briefly explain what a virtual file system is .

To explain the virtual file system , first look at what is the file system . FAT/FAT32, this is the most common file system , our U disk generally uses this file system . In addition to these two , there are many files System : EXT2, EXT4, JIFFS2, UBIFS ...

Linux kernel to user space for a unified interface , on top of the actual file system , encapsulates one virtual file system (VFS). In this way the user space to see is the VFS unified interface provided , and between the real file system the difference , then the encapsulation layer shielding off by the kernel , user space do not have to pay

attention to these differences . by providing the core register_filesystem function , will be able to VFS registered a virtual file system .

sysfs is a virtual file system , its code path in fs / sysfs. Its initial function is sysfs_init , in this function , will call register_filesystem , the VFS registration virtual file system .
It based RAM mean , sysfs is built in RAM on , the user space all on sysfs increase / delete / modify operation , gone after the restart . For example, you sysfs created a file , after the restart , the file It's gone .

The difference between sysfs proc devfs udev devtmpfs

First of all , except for udev, the others are all virtual file systems .

devfs , in Linux 2.4 introduced , it is called device file system , mainly related to / dev device node directory about . once spoke highly of many engineers . However, in Linux 2.6 are udev replaced , because devfs done The work is believed to be done in user mode , and the maintainer of devfs stopped maintaining the code .

udev , introduced in Linux 2.6 , works entirely in user mode . It is a daemon that uses the hot plug events sent by the kernel when devices are added or removed to manage the creation and deletion of device nodes in the /dev directory . This article will later There is a chapter dedicated to how the kernel sends hotplug events .

devtmpfs , can be understood as an enhanced version of devfs, it experienced beginning in the design of some debate , it was agreed , was opposed , details refer to here (http://lwn.net/Articles/331818/) . But in the end or in the 2.6.31 Linux to the kernel when . Its main purpose is to optimize start time , some embedded devices require a higher start-up time , and udev for some reason will slow down the start-up time . devtmpfs will eventually be mount to the / dev directory .

proc, a virtual file system used to provide process and status information

sysfs, introduced in Linux 2.6 , is used to show the hierarchical relationship and attributes between various parts in the device model .

Why here describes these file system , on the one hand it is sysfs have a comparative understanding , on the other hand , this paper describes in later chapters uevent event time , require background knowledge in this section .

4. kobject & ktype &attribute

4.1 Introduction

Kobject

An emotional understanding is : kobject correspondence / sys / directory under the (folder) or subdirectory . But not / sys / all directory corresponds kobject, and some other forms directory is created .
Kobject is the cornerstone of the Linux device model :

Linux kernel device model is used Bus , Class , Device , Driver four core data structure , a large number of different functions of the hardware device (and method of driving the hardware device) , in the form of a tree structure , summarize, abstract , So as to facilitate the unified management of Kernel ;
The number and types of hardware devices are very large , which determines that there will be a large number of data structures related to the device model in the Kernel . These data structures must have some common functions , which need to be abstracted and implemented in a unified manner , otherwise it will be impossible Avoid redundant code ;
These are the two backgrounds of the birth of Kobject , so the main role of Kobject can also be summarized into two aspects .

Kobject has two main functions :

- Organize the hierarchy in a **tree structure** : through the parent pointer , all Kobjects can be combined in a hierarchical structure
- **reference count** : using a reference count (Reference COUNT) , to record Kobject citation counts , and reference number becomes 0 when it is released

Note1: In Linux , Kobject hardly exists alone. Its main function is to be embedded in a large data structure to provide some underlying functional realization for this data structure
Note2: Linux driver developers rarely use Kobject and the interfaces it provides directly , but use the device model interface built on Kobject .

A ttribute

A perceptual understanding is : Attribute is a file in a certain directory of /sys/ , and this file is called an attribute file .
Attribute and Kobject relationship is also evident , file exists in a directory under , so Attribute also with Kobject linked . We generally say that a Kobject have certain properties .

Attribute has two main parameters :

name : the name , / sys / file name under , is the name
mode : permissions , read and write permissions for the file

ktype

There is no way to give Ktype a perceptual understanding .
Ktype main role is to Kobject of some common things abstract together .
For example, we mentioned above , Kobject provide the capability for reference counting , when the count to 0 Shi , will release resources , so that each Kobject should have a release method to release resources
In the properties file mentioned above , it is possible that several Kobjects have the same properties , and we can define these properties in a Ktype .

Ktype concept of comparative inside the kernel rare understanding , we will be at the end of this chapter , for Kobject, Ktype, Attribute make a whole explanation , for ease of understanding .

Based on the above description , Ktype mainly has the following functions :

- **release** : Kobject 's resource release method

- struct attribute ** **default_attrs** : Note that it is a double pointer , corresponding to a head of the list . Each Attribute corresponding to a file attribute , list header representing attributes have a plurality of files . When a Kobject point the Ktype time , this Kobject All these properties files will be included .
- **sysfs_ops** : the read / write function corresponding to the attribute file

4.2 Main data structure

Kobject

/* Kobject: include/linux/kobject.h line 60 */

struct kobject	**Comment**
const char *name	The Kobject name , but also the sysfs directory name
struct list_head entry	The list_head used to add Kobject to Kset (The relationship between Kobject and Kset will be explained in detail in the next chapter)
struct kobject *parent	Point to parent the kobject , thus forming a hierarchical structure (in sysfs on the perfor mance of the directory structure)
struct kset *kset;	The kobject belongs Kset . May be NULL . If present , and do not specify the parent , wil l the Kset of Kobject as parent (as Kset is a special Kobject)
struct kobj_type *ktype;	The kobj_type that the Kobject belongs to . Each Kobject must have a ktype , otherwise the Kernel will prompt an error
struct sysfs_dirent *sd;	The representation of the Kobject in sysfs
struct kref kref;	"The kref struct " type (in the include / linux / kref.h definition) variable , the reference count may be used as an atomic operation
u nsigned int state_initialized:1	Indicate whether the Kobject has been initialized , so as to perform exception verificatio n during Kobject 's Init , Put , Add, etc. operations
unsigned int state_in_sysfs:1	Indicating that the Kobject is already in sysfs presentation , so that when the automatic l ogout from sysfs removed in
unsigned int state_add_uevent_sent:1	Record whether ADD uevent has been sent to user space
unsigned int state_remove_uevent_se nt:1	Whether the record has been sent to the user space of the REMOVE uevent , if you have sent ADD uevent, and did not send the Remove uevent , at the time of automatic logoff , reissue the REMOVE uevent , to allow users to correctly handle space
unsigned int uevent_suppress	If this field is 1 , it means that all reported uevent events are ignored

Note: UEVENT provides a "user space notification" feature to achieve , through this function , when the kernel has Kobject increase, delete, modify, and other actions , will notify the user space .
Details about this feature , will be in its present text later chapter section described in detail.

A ttribute & bin_attribute

/* attribute : include/linux/ sysfs .h */

struct attribute as a normal attribute , the use of the attribute generated sysfs files , can read and write with a string . Typically such a core to form a variable space exposed to the user , so that the user modify this variable space

struct attribute	**Comment**

const char *name	The Attribute name , but also the sysfs in the file name
umode_t mode	Read / write permissions
#ifdef CONFIG_DEBUG_LOCK_ALLOC bool ignore_lockdep:1; struct lock_class_key *key; struct lock_class_key skey; #endif	CONFIG_DEBUG_LOCK_ALLOC related , don't pay attention here

struct bin_attribute binary properties , the struct attribute based on , increasing the Read , Write and other functions , generates sysfs file can be read in any way . Usually this is used to form a block Memory exposed to the user space , such as the EEPROM storage space exposed to user space , directly in user space read / write EEPROM. This kind of property file will not be analyzed in detail for the time being .

struct bin_attribute	**Comment**
struct attribute attr	Encapsulate struct attribute
size_t size;	
void *private	
ssize_t (*read)(struct file *, struct kobject *, struct bin_attribute *, char *, loff_t , size_t)	Read function
ssize_t (*write)(struct file *, struct kobject *, struct bin_attribute *, char *, loff_t , size_t)	Write function
int (*mmap)(struct file *, struct kobject *, struct bin_attribute *attr, struct vm_area_struct *vma)	map function

K obj_attribute

/* kobj_attribute : include/linux/ kobject.h line 138 */

struct kobj_attribute	**Comment**
struct attribute attr	Indicates that it is an encapsulation of the struct attribute structure
show	Property file reading function Is not it a bit confused ! Attributes of the file read / write operation is not in ktype definition of it ? Here is how the case . The patience to read , the last chapter in a section of " summary " will explain the relationship between them
store	Property file write function

Ktype

/* include/linux/kobject.h, line 108 */

struct kobj_type	**Comment**
void (*release)(struct kobject *kobj)	By this back transfer function , you may be of the type comprising kobject memory data structures relieve

co nst struct sysfs_ops *sysfs_ops	This type of Kobject of sysfs file system interface , is the attribute of the fi le read / write function
struct attribute **default_attrs	This type of Kobject of atrribute list (the so-called attribute , is sysfs a fil e in the file system) . Will Kobject when added to the kernel , along with r egistration to sysfs in
const struct kobj_ns_type_operations *(*child_ns_ty pe)(struct kobject *kobj)	It is related to the namespace of the file system (sysfs) , so I won't explai n it in detail here.
const void *(*namespace)(struct kobject *kobj)	It is related to the namespace of the file system (sysfs) , so I won't explai n it in detail here.

4.3 Main API description

Kobject

Header file： include/linux/kobject.h

Implementation code : lib/kobject.c

kobject API	Comment
void kobject_init (struct kobject *kobj, struct kobj_type *ktype)	We can use kmalloc to allocate a Kobject structure by ourselves , and then use the API to initialize this structure . Note that a parameter of type kobj_type needs to be passed in this API . This parameter i s required and cannot be empty , which confirms what we mentioned above . Every Kob ject needs to have a Ktype. If we call kmalloc to allocate Kobject by ourselves, then we must also allocate and initia lize Ktype by ourselves, and set theKtype passed to Kobject
int kobject_ add(struct kobject *kobj, struct kobject *parent, const char *fmt, …)	The initialization completion of kobject added to the kernel of , parameters including th e need to add kobject , the kobject 's parent (for forming a hierarchical structure , may b e empty), for providing kobject name format strings .
int kobject_ini t_and_add(struct kobje ct *kobj, struct kobj_type *ktype, struc t kobject *parent, const char *fmt, …)	Is a combination of the above two interfaces
extern void kobject_del(struct kobject *kobj);	add corresponding inverse function , the kernel is removed in kobject
extern struct kobject * __must_check kobject_create (void);	The API assigns a Kobject memory space , then call kobject_init initialization Kobject. I nitialization , passed Ktype is a system called their own definition of dynamic_kobj_kty pe of Ktype. If we use this API, so we do not need to own distribution Kobject space , do not need to own to get hold of Ktype, the system will take care of themselves The API returns an initialized good Kobject, then we can call kobject_add add it to the k ernel in
extern struct kobject * __must_check kobject_create_and_add(const char * name, struct kobject *parent);	Automatically add after Create

kobject_set_name_vargs	Set the name of kobject
kobject_set_name	Encapsulation of the previous function , used to set the name of kobject
kobject_rename	Modify the name of kobject
kobject_get	Call kref_get (& kobj-> the kref) , to increase kobject reference count . Generally we use kobject_add to a kobject when added to the kernel , if its parent is not NULL, will be the parent point of kobject reference count by one .
kobject_put	Call kref_put (& kobj-> kref, kobject_release) to reduce kobject reference count . When the count to 0 when , will pass in performing kobject_release function to release the res ources .

Attribute

Header file： include/linux/ sysfs .h

Implementation code : fs/sysfs/file.c

Attribute API	Comment
sysfs_create_file(struct kobject *kobj, const struct attribute *attr)	In a kobject create the next 1 attribute file
sysfs_remove_file(struct kobject *kobj, const struct attribute *attr)	Delete a property file under kobject
sysfs_create_group(struct kobject *kobj, const struct attribute_group *grp)	In kobject created under multiple properties files , attribute_group equivalent to an arra y , which can store multiple properties files . Interestingly, this group can have a name, when the name is NULL when , all the propert ies are stored in files kobject the corresponding directory ; when the name is not NULL when , will kobject generates a corresponding directory under the name directory , all P roperties files are stored in the name directory
sysfs_remove_group(struct kobject *kobj, const struct attribute_group *grp)	Delete an attribute group under kobject
sysfs_create_groups(struct kobject *kobj, const struct attribute_group **groups)	Create multiple attribute groups
sysfs_remove_groups(struct kobject *kobj, const struct attribute_group **groups)	Delete multiple attribute groups

K obj_attribute & __ATTR

Header file： include/linux/ sysfs .h

This is not an API, it is a macro , #define __ATTR(_name, _mode, _show, _store) . This macro is used to initialize the kobj_attribute data structure .

4. 4 key code analysis & DEMO

Imagine a scenario : We want to write a kernel module , loaded into the kernel (remember the kernel module how to write it !), Used in / sys / create a directory , then in the creation directory 1 attribute file ,You can read / write the properties file . Don't read the content below , think about what you would do ?

First of all , we need we kobject create a directory , there are two ways to create a table of contents :

- manually create : the meaning of the manually created with kmalloc discretionary kobject, defined Ktype, tune with kobject_init initialization , and then call kobject_add added to the kernel
- Automatic creation : Automatic creation means to use kobject_create or kobject_create _add to get everything done .

Then , we need in kobject create a properties file directory , create a properties file a number of ways :

- ➢✋➶ it is a manually created kobject directory , then
  - ■➱❒☐◘♏❒♦♓♏♦ file can be defined by ♲☐◆❒♦♏●➶
  - ■ may be in Ktype define a default list of attributes which (reference Ktype data structure), when the kobject when added to the core , these properties files will automatically appear .
- ➢✋➶ it is automatically created , then
  - ■❄♒♏ attribute file can only be added by calling the Attribute API , and the default attribute list cannot be defined . The reason is that the system will process the Ktype internally when it is automatically created , and we cannot add the default attribute list .

Create Kobject manually

Analyze several key functions

kobject_init

Initialize the struct kobject pointer obtained through kmalloc and other memory allocation functions . The main execution logic is

- ➢ 💣♋&♏ sure that the parameter ktype is not empty , that is , when initializing kobject , you must give it a Ktype
- ➢ If the pointer has been initialized (judgment kobj-> state_initialized) , print an error message and stack information (but not a fatal error , it can continue)
- ➢ call kobject_init_internal initialization kobj internal parameters , including the reference count, List , various signs, etc.
  - ■ When initializing the reference count , the function call is kref_init , **the function will reference count is initialized to 1** .
- ➢ according to the input parameters , the ktype pointer imparting kobj-> ktype

kobject_add

The initialization completion of kobject added to the kernel of , parameters including the need to add kobject , the kobject 's parent (for forming a hierarchical structure , may be empty), for providing kobject name format string . The main logic is performed :

- ➢ Confirm that kobj is not empty , confirm that kobj has been initialized , otherwise exit with an error
- ➢✍♋●● ♦♒♏ internal interface kobject_add_varg to complete the add operation

- kobject_add_varg , parse the formatted string , assign the result to kobj->name , and then call the kobject_add_internal interface to complete the actual addition operation
  - kobject_add_internal : add kobject to the kernel . The main execution logic is
    - check kobj and kobj-> name of legitimacy , if not legality print an error message and exit
    - call kobject_get increase the kobject the parent of the reference count (if parent any)
    - If there kset (ie kobj-> kset not empty) , then call kobj_kset_join interface is added kset . Meanwhile , if the kobject no parent , but there kset , then do two things : to kobject the parent set kset of kobject (kset is a special kobject) , and increase the reference count of kset -> kobjecct

      NOTE: This chapter does not consider kset for the time being, kset is NULL, the next chapter is dedicated to kset
    - By create_dir interface calls sysfs related interfaces in sysfs created under kobject corresponding directory
    - If the creation fails, the subsequent implementation of the rollback , otherwise kobj-> state_in_sysfs set to 1

kobject_get

Call kref_get to increase the reference count

kobject_put

In a system-defined interfaces kobject_release parameters , call kref_put . The kref time zero reference count module , call kobject_release

- kobject_release : by kref structure , get kobject pointer , and call kobject_cleanup interfaces continue
- kobject_cleanup : Responsible for releasing the space occupied by kobject , the main execution logic is as follows
  - Check that the kobject whether there ktype , if not , print a warning message
  - ■☝➚ the kobject sends ADD uevent to user space but does not send REMOVE uevent , reissue REMOVE uevent
  - ■☝➚ the kobject is registered in the sysfs file system , call the kobject_del interface to delete its registration in sysfs
  - calling the kobject of ktype the release interfaces , free up memory space
  - ■☼♏●♏♋♦♏ the memory space occupied by the kobject 's name

DEMO

https://gitlab.com/study-linux/linux_driver_model/blob/master/manual_kobject_attribute.c
(https://gitlab.com/study-linux/linux_driver_model/blob/master/manual_kobject_attribute.c)

NOTE 1 : Why practice creating kobject manually
When writing the real kernel driver , we basically do not go directly manipulate Kobject. Instead of using the upper level of the interface . For example, we know the device model, there is a device concept , when we need to add a device time , calling the system The provided device_add interface , this function will eventually handle Kobject related things .
We can be understood as device encapsulates kobject, and this packaging operation , the processing kobject time , are used in the manual mode mentioned above .
So we practice the manual method here in order to better understand the following chapters (device, device_driver, bus, class, they are the real focus).

Create Kobject automatically

Analyze several key functions

kobject_create

The interface kobj allocate memory space , and is built into the system dynamic_kobj_ktype parameters , call kobject_init interfaces to accomplish the subsequent initialization operation

kobject_create_and_add

Combination of kobject_create and kobject_add

DEMO

https://gitlab.com/study-linux/linux_driver_model/blob/master/auto_kobject_attribute.c
(https://gitlab.com/study-linux/linux_driver_model/blob/master/auto_kobject_attribute.c)

4.5 Details of sysfs read and write property files

In the previous article, we introduced sysfs, knowing that it is a virtual file system .
In the introduction Kobject time , we also mentioned Attribute, know Kobject represents a directory , and Attribute representatives of files in that directory , we can sysfs to read / write the property file .

sysfs is a virtual file system , virtual file system is the file system , Linux file system will provide open, read, write interfaces to the user space , which is the Linux everything is a file concept .
When we want to sysfs time to read and write attributes of the file , first call is certainly sysfs the open function , then the sysfs the read / write function . Then what ? Read / write what is specific to the operation of the properties file it ? Struct attribute The data structure is very simple , I haven't seen any interface function to respond to the read/write interface of sysfs !

Next , let's briefly sort out this process :
All file systems , will define a struct file_operations variable , used to describe the operation of the interface system of the present document , the sysfs no exception . Defined in FS / kernfs /file.c (http://lxr.free-electrons.com/source/fs/kernfs/file.c?v=3.18#L880) .
In the next step , the code will enter the kernfs_ops (http://lxr.free-electrons.com/source/fs/sysfs/file.c?v=3.18#L182) in fs/sysfs/ (http://lxr.free-electrons.com/source/fs/sysfs/file.c?v=3.18#L182) file.c , there are many kernfs_ops , generally for the property file issysfs_file_kfops_rw (against binary attributes sysfs_bin_kfops_rw , here not elaborate).
Next sysfs_file_kfops_rw will find kobject (each file will correspond to a particular attribute kobject, when we tried to open when a property file , we already know that it's kobject a), to obtain kobject-> ktype.
Next , ktype->sysfs_ops.

To sum up a bit , the above process : VFS -> file_operations -> sysfs(kernfs_ops) -> (kobject->ktype) -> (ktype->sysfs_ops).

Next , if we define ktype ourselves, the above process will eventually call ktype->sysfs_ops directly , as demonstrated by the DEMO of manually creating Kobject ;

If it is a system-defined ktype, the above process will add another link , (ktype->sysfs_ops) -> (kobj_attribute->show/store), as demonstrated by the DEMO that automatically creates Kobject .

There is one more question to think about :

By sysfs reading properties files , meaning you need to user space buffer write data , through sysfs write property files , meaning that you need from the user space buffer to read the data .

In the article on the kernel module, we discussed that the kernel cannot directly access the buffer in the user space and needs to use copy_to_user/copy_from_user.

If you pay close attention to the code in the DEMO , you will find that the store/show function we wrote does not handle this buffer . Why ?

The reason is that the file_operations part of the system helped us deal with these problems , and it also considered the issue of mutual exclusion . If you are interested, you can take a look at its code .

4.6 kobj_attribute, device_attribute, driver_attribute, bus_attribute, class_attribute

Several primary kernel data structures associated with the model of the device , there is a corresponding Attribute. Their manifestation is similar to kobj_attribute . When reading and writing attribute files , the kernel system will automatically handle the (xxx_attribute->show/store) link for you . You only need to use the API (or macro) provided by them to define The properties file is fine .

__ATTR :

```
#define __ ATTR ( _name, _mode, _show, _store) { \
.attr = {.name = __stringify(_name), \
.mode = VERIFY_OCTAL_PERMISSIONS(_mode) }, \
.show = _show, \
.store = _store, \
}

static struct kobj_attribute foo_attribute =
    __ ATTR ( foo,0666, foo_show, foo_store);
```

DEVICE_ATTR

```
#define DEVICE_ ATTR( _name, _mode, _show, _store) \
struct device_attribute dev_attr_##_name = __ ATTR( _name, _mode, _show, _store)
```

DRIVER_ATTR

```
#define DRIVER_ ATTR( _name, _mode, _show, _store) \
struct driver_attribute driver_attr_##_name = __ ATTR( _name, _mode, _show, _store)

#define BUS_ ATTR ( _name, _mode, _show, _store) \
struct bus_attribute bus_attr_##_name = __ ATTR( _name, _mode, _show, _store)

#define CLASS_ ATTR ( _name, _mode, _show, _store) \
struct class_attribute class_attr_##_name = __ ATTR( _name, _mode, _show, _store)
```

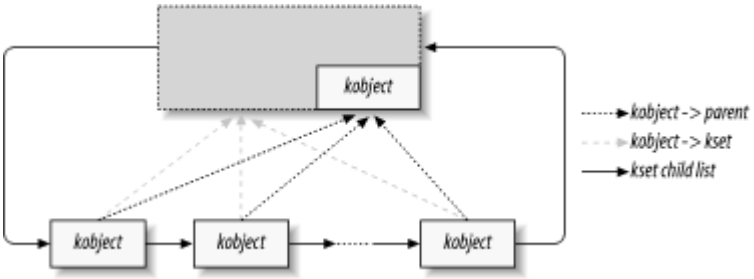
5. kset

5.1 Introduction

After understanding kobject, it is much easier to look at kset . You can read this section in conjunction with the kset data structure in the next section .

Kset is a special Kobject (so it will also appear in the form of a directory in /sys/) , it is equivalent to a container , and multiple kobjects can be mounted below . It is used to aggregate similar Kobjects (these Kobjects can have the same the ktype, may also have different of ktype) .

As mentioned above , Kobject in / sys / performance for a directory , who this directory is the parent directory , by kobject-> parent decision .
When a kobject belong to a kset when , in most cases , the kobject-> parent will point kset of kobject. In this case , we can kset simply seen as a parent directory , but belongs to kset of kobject all subdirectories . like this, like the following diagram :



One more thing , note that in most cases , even if a kobject belongs to a kset, it does not necessarily mean that the kobject->parent must point to the kobject of the kset . Correspondingly , there is no parent / child in this case Directory relationship .

Kset is mainly related to Uevent event reporting, and Uevent will be explained in detail in the next chapter .
Mentioned above kset used to group together similar kobject, this means that similar kset can decide it under kobject which can be reported uevent events ; what uevent information is subordinate to all kobject be unified reported .

5.2 Main data structure

Kset

/* include/linux/kobject.h, line 159 */

struct kset	Comment
struct list_head list	Used to link the kset all under kobject the list head
spinlock_t list_lock	Lock , used for mutual exclusion to add / delete kobjects to the linked list
struct kobject kobj	kset's own kobject (kset is a special kobject , which will also be reflected in the form of a directory in sysfs)

const struct kset_uevent_ops *uevent _ops	The kset of uevent operating set of functions . When any Kobject needs to be reported u event time , should call it belongs kset of uevent_ops , adding environment variables , or filter event (kset can decide which event can be reported) . Therefore , if a kobject n ot belong to any kset time , it is not allowed to send uevent of

kset_uevent_ops

/* include/linux/kobject.h, line 123 */

struct kset_uevent_ops	Comment
int (* const filter)(struct kset *kset, str uct kobject *kobj)	filter , if any Kobject needs to be reported uevent time , it belongs kset can filter through the interface , to prevent undesired reported Event , so as to achieve the purpose of man aging as a whole
const char *(* const name)(struct kset *kset, struct kobject *kobj)	name , this interface can return the name of the kset . If a kset does not have a legal na me , all Kobjects under it will not be allowed to report u e vent
int (* const uevent)(struct kset *kset, s truct kobject *kobj, struct kobj_uevent _env *env);	uevent , when any Kobject needs to be reported uevent time , it belongs kset through th e unified interface for these event to add environment variables . Very often reported ue vent environment variables are the same , and therefore can be made kset unified treatm ent , do not need Let each Kobject add independently

kobj_uevent_env represents a specific event , which is explained in detail in the Uevent chapter

5.3 Main API description

Kset is a special kobject , so the initialization, registration and other operations will also call kobject related interfaces , in addition , will have its unique part . In addition , and Kobject the same , kset memory allocation , by the upper software kmalloc allocates itself , or it can be allocated by the kset module , as follows

kset

Header file： incl ude/linux/kobject.h, line 166

Implementation file : lib/kobject.c

kset API	Comment
extern void kset_init(struct kset *kset)	This interface is used to initialize the allocated kset , which mainly includes calling kobject_init_internal to initialize its kobject , and then initialize the kset 's linked li st head and lock . It should be noted that if this interface is used , the upper software must provide th e ktype of the kobject in the kset
extern int __must_check kset_register(stru ct kset *kset)	Call kset_init first Then call kobject_add_internal to add its kobject to the kernel Then send a KOBJ_ADD uevent event
extern void kset_unregister(struct kset *kse t)	Directly call kobject_put release the kset of kobject . When it kobject reference co unt is 0 when , i.e. calling ktype the release interface release kset space occupied

extern struct kset * __must_check kset_create_and_add(const char *name, const struct kset_uevent_ops *u, struct kobject *parent_kobj)	The internal interface kset_create is called to dynamically create a kset , and kset_register is called to register it with the kernel . kset_create is an internal interface (static function), the interface uses kzalloc assigned a kset space , and defines a kset_ktype type ktype , for releasing it from all assigned kset space

5.4 DEMO

Imagine this scenario : We want to / sys / build under 1 parent of the directory , there will be under the parent directory 1 attribute file , 1 subdirectories , there will be a subfolder 1 attribute file . You can read this 2 Ge Properties file . Think about what you would do ?

First , we need to create a kset and add the kset to the kernel , so that a directory will appear under /sys/ . You can create it manually (kmalloc -> kset_register), or you can create it automatically (kset_create_and_add).
Then , define a property file by yourself and add it to this directory .
Next , create a kobject, let kobject->parent=NULL, kobject->kset=kset. Then add the kobject to the kernel , so that the kobject will automatically become a subdirectory of kset .
Finally , call the Attribute API to create an attribute file under the kobject .

DEMO: https://gitlab.com/study-linux/linux_driver_model/blob/master/manual_kset_kobject_attribute.c (https://gitlab.com/study-linux/linux_driver_model/blob/master/manual_kset_kobject_attribute.c)
We made a DEMO, kset is manually created , kobject also is the use of manually (only manually , if you look at kobject the API, you will find , only manually , can be specified kobject of kset) .

It is recommended that when you read this , **you try** to complete the above scenario by automatically creating a kset .

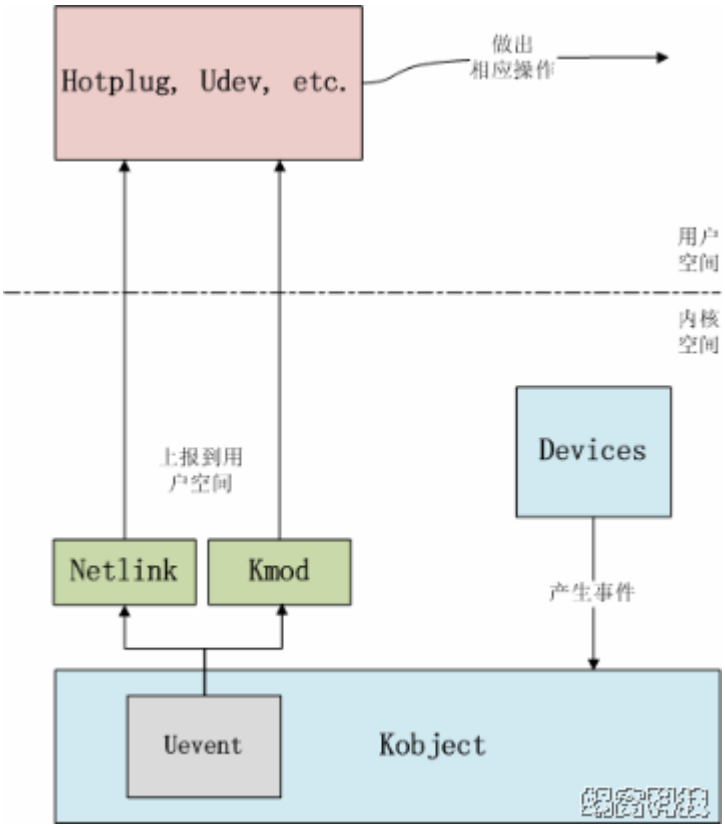
6. Uevent

6.1 Introduction

Uevent is for Kobject of , for Kobject when the state changes , such as adding, removing, etc. , notify the user space program . After the user-space program received such an event , it will be handled accordingly .

This mechanism is typically used to support hot plug devices , for example, U after the disk is inserted , the USB driver software associated with a dynamically created representing U disk device structure (also including therein a corresponding the kobject) , and inform the user space program , for the U to create a dynamic disk / dev / device node directory , further , may notify other applications , the U disk device mount into the system , so that the dynamic support device .

The following picture describes the location of the Uevent module in the kernel



It can be seen , Uevent mechanism is relatively simple , device model in any equipment when events need to be reported , will trigger Uevent interface provided .

Uevent After the module is ready reporting format of the event , can report the incident to the user space in two ways: one is by kmod module , directly call the executable file in the user space (referred to uevent Helper) ; the other is through netlink communication Mechanism to pass events from kernel space to user space

These two methods can exist at the same time , which means that as long as the kernel enables both methods at the same time , an event will be reported through these two methods at the same time .

6.2 Main data structure

kobject_action

/* include/linux/kobject.h, line 50 */

kobject_action defines the type of event , including :

enum kobject_action	Comment
KOBJ_ADD	Add event of kobject or upper data structure (embedded kobject)
KOBJ_REMOVE	Removal event of kobject or upper data structure (embedded kobject)
KOBJ_CHANGE	The status or content of kobject or the upper data structure (embedded with kobject) has changed If the event that the device driver needs to report is not within the scope of the event defined by k object_action , or is a custom event , you can use the event and carry the corresponding paramete rs
KOBJ_MOVE	kobject or upper layer data structure (embedded kobject) change the name or change the Parent (mean sysfs changes to the directory structure)
KOBJ_ONLINE	
KOBJ_OFFLINE	

KOBJ_MAX	

kobj_uevent_env

Kobj_uevent_env can be simply understood as a large block of memory . The size of this block of memory is 2048. Whenever a uevent event needs to be reported , such a block of memory will be generated . Then use the add_uevent_var function to add information about the event to this block of memory (ACTION= %s, DEVPATH=%s , SUBSYSTEM=%s). These messages are all strings .

Adding get away pieces of information later , will be the event reporting user space , the one mentioned , the kernel sends uevent event in two ways :

- ♑♒♏ Kmod method directly calls the executable program in the user space : In this way, kobj_uevent_env will also save the path of the executable program (for example, /sbin/hotplug), and the parameters passed to the executable program . After preparing these contents, it will be uevent reported .
- NetLink mode : In this mode , will kobj_uevent_env save the event information copy to a call sk_buff cache , and then through the socket reporting mechanism .

/* include/linux/kobject.h, line 31 */
#define UEVENT_NUM_ENVP 32 /* number of env pointers */
#define UEVENT_BUFFER_SIZE 2048 /* buffer for the variables */

/* include/linux/kobject.h, line 123 */

struct kobj_uevent_env	Comment
char *argv[3]	argv[0]: Save the executable path of uevent helper (e.g. /sbin/hotplug) argv [1]: passed to uevent helper first parameter , a string , generally kobject belongs kset nam e argv[2]: The second parameter passed to uevent helper , generally NULL
char *envp[UEVENT_NUM_ENV P]	Pointer array An event will have multiple information , each of which is a string (see above in this section), and all the information is stored in buf[UEVENT_BUFFER_SIZE] in turn . envp saves the starting address in buf[UEVENT_BUFFER_SIZE] of each message . You can save a total of UEVENT_NUM_ENVP a message address
int envp_idx;	Every time a message is added , envp_idx++ Record how many messages are added in total
char buf[UEVENT_BUFFER_SIZ E]	The buf to save the information , the size is UEVENT_BUFFER_SIZE
int buflen	Indicates how much space has been used in buf the sizeof (buf) - buflen, is the size of the remaining space

kset_uevent_ops

kset tailor-made data structure , convenient kset unified management under kobject of uevent event .
The specific details have been introduced in the main data structure of kset above

6.3 Main API description

Uevent

Header file : include/linux/kobject.h, line 214

Implementation file : lib/kobject_uevent.c

Uevent API	Comment
int kobject_uevent(struct kobject *kobj, enum kobject_action action)	Send an event of type kobject_action for a certain kobject . Remember , the parameter of this API is a kobject, which means that uevent events can only be sent for a certain kobject
int kobject_uevent_env(struct kobject *kobj, enum kobject_action action, char *envp[])	On a package API, the difference is , the API will attach some event information , the event information stored in envp the
int add_uevent_var(struct kobj_uevent_env *env, const char *format, ...);	Note that its first parameter , kobj_uevent_env, is the memory block that stores event information that we introduced in the data structure . This API can be understood as an interface for operating memory blocks , which is used to store event information (that is, the following parameter : string) in the memory block .
int kobject_action_type(const char *buf, size_t count, enum kobject_action *type)	Convert the action of enum kobject_action type to a string

6.4 Key code analysis

kobject_uevent_env : In envp environment variables , reports a specific action of a uevent , specific operation is as follows :

- Find whether kobj itself or its parent belongs to a certain kset , if not , then report an error and return (this can explain that if a kobject does not join the kset , it is not allowed to report uevent)
- Check kobj-> uevent_suppress is set , if set , then ignore all the uevent report and return (can be seen , by Kobject of uevent_suppress signs , control Kobject of uevent reporting)
- If you belong kset have uevent_ops-> filter function , then call this function , filtering the report (refer back to kset_uevent_ops data structure , which evidence about the filter interface description , kset can filter interface to filter not want to report the Event , thereby Achieve the overall management effect)
- Determine whether the kset to which it belongs has a legal name (called subsystem , which is different from the previous kernel version) , otherwise it is not allowed to report to uevent
- Allocate a buffer for storing environment variables for this report (that is, allocate a kobj_uevent_env data structure) , and obtain the path information of the Kobject in sysfs (user space programs need to access it in sysfs based on the path information)
- call add_uevent_var interface (will be described below) , to the Action , route information, Subsystem information , added to the kobj_uevent_env memory block
- If the incoming envp is not empty , the incoming environment variables will be parsed, and the add_uevent_var interface will also be called and added to the kobj_uevent_env memory block
- If you belong kset presence uevent_ops-> uevent interfaces , invoking the interface , add kset unified environment variable to kobj_uevent_env memory block
- The ACTION type , provided kobj-> state_add_uevent_sent and kobj-> state_remove_uevent_sent variables , to record the correct state
- call add_uevent_var interfaces , add the format of "=% SEQNUM LLU " sequence number
- If the definition of "CONFIG_NET " , using netlink send the uevent

- If the definition of " CONFIG_UEVENT_HELPER " , then call init_uevent_argv initialization kobj_uevent_env the argv variables (look back kobj_uevent_env data structure), which specifies a uevent Helper path , the parameter ; then to kobj_uevent_env add environment variable information in the memory block (HOME = /, PATH = /sbin:/bin:/usr/sbin:/usr/bin).
  Finally , with kmod module provides call_usermodehelper functions , reported uevent
  Wherein a uevent Helper executable path by the kernel configuration item CONFIG_UEVENT_HELPER_PATH (located ./drivers/base/Kconfig) determined (refer to lib / kobject_uevent.c, Line 32) , the configuration item specifies a user-space program (or Script) , used to parse the reported uevent , such as "/sbin/hotplug "
  call_usermodehelper role , is the fork a process , in order to uevent parameters , perform uevent_helper

kobject_uevent : and kobject_uevent_env functions, like , just not specify any additional event information

add_uevent_var : kobj_uevent_env memory block operation interface , in the form of formatted characters (similar to printf , printk, etc.) , copy environment variables to the memory block

6.5 About how to specify the uevent helper executable program

The above described kobject_uevent_env internal operation , there are mentioned , uevent module Kmod reported uevent time , will pass call_usermodehelper function , call user space executable file (or script , referred uevent helper) handle the event. Which uevent helper path Saved in the uevent_helper array .

You can compile the kernel , by CONFIG_UEVENT_HELPER_PATH configuration items , statically assign uevent helper. But in this way would be for each event fork a process , as the number of devices supported by the kernel , in this way at system startup would be fatal (Can lead to memory overflow, etc.) . Therefore, this method is only used in the early kernel version, and now the kernel is no longer recommended to use this method . **Therefore, when the kernel is compiled , you need to leave this configuration item blank**

After the system starts , most of the equipment has been READY , as needed , to reassign a a uevent helper , so that the thermal detection system during operation of the pulling plug event . This may be accomplished by helper path is written to the "/ sys / kernel / uevent_helper " file .
In fact , the kernel through sysfs form of a file system , the uevent_helper array of open space to the user , for the user to modify the program space access , specific reference "./kernel/ksysfs.c " corresponding code , not described in detail herein

7. The top-level directory under /sys/

We have introduced Kobjet and Kset earlier. We know that Kobject represents a directory under /sys/ , which is used to organize hierarchical relationships ; we also know that kset is a special Kobject, so it is also a directory under /sys/ , which has similar functions for collections. the Kobject (mainly Uevent handling events).

Here we want to introduce the / sys / some of the top-level directory , you temporarily do not have to care about the role of each directory , only requires a combination of knowledge learned earlier , to understand how each directory can be created .

/sys/ devices

/sys/ dev

/sys/ dev/ block

/sys/ dev/ char

File: drivers/base/core.c

Function:

```
int __init devices_init (void)
{
devices_kset = kset_create_and_ add( " devices ", &device_uevent_ops, NULL);
if (!devices _kset)
return -ENOMEM;
dev_kobj = kobject_create_and_ add( “ dev “, NULL);
if (!dev _kobj)
goto dev_kobj_err;
sysfs_dev_block_kobj = kobject_create_and_ add( “ block “, dev_kobj);
if (!sysfs _dev_block_kobj)
goto block_kobj_err;
sysfs_dev_char_kobj = kobject_create_and_ add( " char ", dev_kobj);
...
}
```

Note that this function with __init (kernel module article Introducing loading functions , introduced the mark) mark , on its behalf is called the system initialization phase .

/ sys/bus

/sys/devices/system

File: drivers/base/bus.c

Function:

```
int __init buses_init(void)
{
bus_kset = kset_create_and_ add( “ bus “, &bus_uevent_ops, NULL);
if (!bus _kset)
return -ENOMEM;

system_kset = kset_create_and_ add( “ system “, NULL, &devices_kset->kobj);
if (!system _kset)
return -ENOMEM;

return 0;
}
```

Similarly , __init indicates buses_init.

system_kset specified parent when it was created , so it is located in the /sys/devices directory .

/ sys/class

File: drivers/base/class.c

Function:

```
int __init classes_init(void)
{
class_kset = kset_create_and_ add( " class ", NULL, NULL);
if (!class _kset)
return -ENOMEM;
return 0;
}
```

/ sys/kernel

File: kernel/ksysfs.c

Function:

```
static int __init ksysfs_init(void)
{
int error;

kernel_kobj = kobject_create_and_ add( “ kernel “, NULL);
if (!kernel _kobj) {
error = -ENOMEM;
goto exit;
}
...
}
```

Similarly , __init indicates ksysfs_init .

/ sys/module

File: kernel/params.c

Function:

```
static int __init param_sysfs_init(void)
{
module_kset = kset_create_and_ add( “ module “, &module_uevent_ops, NULL);
if (!module _kset) {
printk( KERN_WARNING “%s (%d): error creating kset\n”,
__FILE__, __LINE__);
return -ENOMEM;
}
...
}
```

subsys_initcall(param_sysfs_init);

Similarly , __init indicates param_sysfs_init .

/ sys/block

/ sys / block is actually a class, the kernel defines a system called " Block " of the class, when the the class registration into the kernel , will produce the catalog (reading Class after a chapter you will understand this catalog is How it was generated).

8. BUS

8.1 Introduction

Bus is a processor with a plurality of devices or channels between , in order to facilitate the realization of the device model , the kernel specified , the system must be connected to each device in a Bus on . This Bus can be an internal Bus , Virtual Bus or Platform Bus .

The role of Bus in the kernel is mainly reflected in

- Connect devices and drivers under the Bus
- Bus defines the matching rules of device and driver
- Bus can define the default attributes of subordinate devices and drivers

There are various systems in a bus , these buses intuitive feel is / sys / bus / under a directory . For example / sys / bus / spi represents the SPI Bus, / sys / bus / Platform / Representative Platfrom Bus.

In the previous chapter, we introduced how /sys/bus is generated , then how are these buses in the /sys/bus directory generated ? What is the role of Bus in the kernel system ? With these questions , start the following Chapters .

8.2 The main data structure

bus_type

The kernel abstracts Bus through the struct bus_type structure , and each Bus corresponds to a bus_type structure .

/* inlcude/linux/device.h, line 93 */

struct bus_type	Comment
const char *name	Bus 's name Corresponds to the name of the directory under /sys/bus/ .
const char *dev_name	The default prefix name of the device (struct device) under the bus When a device is defined , when added to the bus , does not specify the name of the device , th e bus it will help to define a name , in the form of : BUS-> Device ID dev_name + < the bus can be mounted a plurality Device, each of Each device has an ID under the bus , and this ID is gene rally automatically assigned by the bus >

struct device *dev_root	Just like every kobject can have a parent, and every device can have a parent device. dev _root is the default parent device of all devices under the Bus
struct d evice_attribute *dev_a ttrs;	/* use dev_groups instead */
const struct attribute_group ** bus_groups	Defining a bus_type time , you can specify a set of " attribute group " , concept of attribute gro ups , reference may be 4.3 Section Attribute of the API: **sysfs_create_group** When bus is added to the kernel , the attribute file defined in the attribute group will be created in the /sys/bus/xxx directory
const struct attribute_group ** dev_groups	Another attribute group When there device is added to the Bus the next , it will be for each device creates these properti es files
const struct attribute_group ** drv_groups	Another attribute group When there device_driver added to the Bus the next , it will be for each driver to create these p roperties files
int (* **match**)(struct device *de v, struct device_driver *drv)	When we intend to create a bus , we need to define the function . This function is used to define the matching rules of device and device_driver under the Bus When any part of the Bus of the device or device_driver added to the Bus time , the system will call the function , the function will tell the system device and driver matches , if the match is s uccessful , the function should return a non-zero value , then the system will perform Subseque nt processing
int (* **uevent**)(struct device *de v, struct kobj_uevent_env *env)	Remember the main role of kset ? Remember the role of kset-> uevent_ops -> uevent functio n? If you don't remember, look back When we intend to create a bus , we need to define this function . Function and effect of the kset in uevent action function similarly , when any part of the Bus 's Device , occurrence added, removed, or when an operation other , Bus core logic module will c all this interface , to add uevent event information of the event .
int (* **probe**)(struct device *de v)	When a device and driver after the match on , the system calls the driver-> probe to initialize t he driver a . However , if you need the Probe (in fact initialization) specified device , then , ne ed to ensure that the device where the bus is being initialized ensure be able to work properly . it is necessary in the implementation of device_driver the probe ago , before the implementatio n of its bus of probe Note : Not all bus need to probe and remove the interface , because some bus for (eg Platform bus) , which is itself a virtual bus , it does not matter initialization , straightforward to use , so this function can be NULL
int (* **remove**)(struct device *d ev)	The logic is similar to the above .
void (* **shutdown**)(struct devic e *dev)	Similar to probe logic , related to power management , not explained yet
int (* **suspend**)(struct device * dev, pm_message_t state)	Similar to probe logic , related to power management , not explained yet
int (* **resume**)(struct device *d ev)	Similar to probe logic , related to power management , not explained yet
int (*online)(struct device *dev)	Hot-swappable relevant , if the Bus of device want to support hot-swap , then this Bus must de fine the online function

int (*offline)(struct device *dev)	Hot-swappable relevant , if the Bus of device want to support hot-swap , then this Bus must de fine the offline function
const struct dev_pm_ops *pm	Power management related logic , not explained yet
struct iommu_ops *iommu_ops	Not for the time being
struct subsys_private *p	A very important structure , we specifically explain
struct lock_class_key lock_key	

subsys_private

This is a very interesting structure , let's take a look at its data structure

/* drivers/base/base.h, line 28 */

struct subsys_private	Comment
struct kset subsys	kset exists in the form of a directory in /sys/ , which represents this Bus, for example, /sys/bu s/spi, which represents the directory of spi , and the name of the directory is specified in bus_t ype
struct kset *devices_kset	kset, representing the directory /sys/bus/ xxx /devices All belong to the Bus of the device, creates a link in the directory (note the link Oh , device rea l directory is not here , here is a real link directory , just as we can in XP for a folder Create a s hortcut)
struct klist klist_devices	Linked list head , all devices under the Bus will be mounted under this linked list
struct kset *drivers_kset	kset, representing the directory /sys/bus/xxx/ drivers All belong to the Bus 's driver, will be stored in the directory , pay attention to is not a link , is t he real head record . Means that all Bus under driver-> kobject the parent is the kset-> kobject
struct klist klist_drivers	Linked list head , all drivers under the Bus will be mounted under this linked list
struct list_head interfaces	Linked list header , used to mount all interfaces under the Bus , the concept of interfaces will n ot be detailed for the time being
struct mutex mutex	And interfaces list head fitted , used in mutex
struct bus_type *bus	Save the upper Bus pointer
struct class *class	Or save the upper Class pointer
struct blocking_notifier_head b us_notifier	The bus_notifer notification chain , when others need to monitor the notification chain , they c an register to the chain . A brief notice kernel chain mechanism , will be behind the article in detail . This practice is to define a mechanism to inform the chain , and then if others want to listen to this motion on noti ce if a new chain , will be registered with the notification chain , when When there is an action on this notification chain , the system will notify all registered people .
unsigned int drivers_autoprobe: 1	For controlling when the driver is added to the Bus when , whether to start matching driver and device
struct kset glue_dirs	

We can see from the data structure , the structure of the body is a collection of some of the bus private data module requires the use of , for example kset friends, klist friends and so on . So why this structure name shall subsys_private, rather than call bus_private it ?

Look at include / linux / device.h in struct class structure (in the next we will introduce chapter class) will know , because the class structure also contains an identical struct subsys_private pointer , seems class and the bus is very similar ah

Thought here , like to understand , whether it is Bus , or class , or any other contains subsys_private data structures , they can be understood as a sub-system, sub-system as an independent kingdom , it will contain the following kinds of devices , drivers and so on .
Therefore , the kernel abstracts the public content in various sub-systems with a data structure such as subsys_private .

8.3 Main API description

Bus

APIs related to Bus mainly include :

- Registration and cancellation of bus
- this bus have the device or device_driver processing of registration to the kernel
- Handling when there is a device or device_driver under this bus that logs out of the kernel
- device_drivers the probe process
- management bus all under the device and device_driver

Header file : inc lude/linux/device.h
Implementation file : drivers/base/bus.c

bus API	Comment
extern int __must_check bus_register(struct bus_type *bus)	Add a new bus to the system
extern void bus_unregister(struct bus_type *bus)	Remove a bus from the system
extern int bus_register_notifier(struct bus_type *bus, struct notifier_block *nb)	Register to the notification chain of Bus When Bus has on devices of Addition / Removal , or when the device and driver pairing / understanding of success when , everyone will receive a notification on the notification chain
extern int bus_unregister_notifier(struct bus_type *bus, struct notifier_block *nb)	Remove from the notification chain of Bus You will no longer receive notifications after removal

Header file : drivers/base/base.h
Implementation file : drivers/base/bus.c

bus API	Comment
extern int bus_add_device(struct device *dev)	When calling device_add to a device added to the Bus when on , device_add internal calls this function , the processing device is added to the Bus when on , Bus here related things

extern void bus_remove_device(struct device *dev	On the contrary , device from the Bus is removed , call the function
extern void bus_probe_device(struct device *dev)	When you call device_add to a device added to the Bus when , if bus_set-> subsys _private-> drivers_autoprobe is 1 , then the (creating Bus time , the default is 1), will be at the Bus of drivers searching for all under the list driver, to see if there The driver that can be paired with the device , if there is , call driver->probe to start initialization
extern int bus_add_driver(struct device_driver *drv)	When you call driver_register to a driver added to the Bus when , driver_register internal calls this function , processing driver added to the Bus when on , Bus here related things
extern void bus_remove_driver(struct device_driver *drv)	Contrary to the above , call this function when driver_unregister

Driver & Device binding related

Header file : include/linux/device.h
Implementation file : drivers/base/dd.c
The content of this section **can be skipped for** now (it may not **be easy to** understand right now) . In the following chapters , the related functions here will be quoted . When you need to understand the details , you can refer to this chapter .

device_bind_driver

Bind a driver to a device. When calling this function , the device->driver pointer already points to a driver.
The specific execution logic is as follows :

- **driver_sysfs_ add** (struct device *dev)
  - If the device is part of a Bus, then Bus to send a message on the notification chain BUS_NOTIFY_BIND_DRIVER
  - In the device pointed to by the driver under the directory , create a link to the device link , the link name is device-> kobj 's name, for example :

```
[redacted]:~$ ls /sys/bus/platform/drivers/i8042/ -l
total 0
lrwxrwxrwx 1 root root    0 Aug 13 22:51 i8042 -> ../../../../devices/platform/i8042
```

  - In the directory of the device , create a link to the driver , the name of the link is " driver "

```
[redacted]$ ls /sys/devices/platform/i8042/ -l
total 0
lrwxrwxrwx 1 root root    0 Aug 13 20:13 driver -> ../../../bus/platform/drivers/i8042
```

- **driver_ bound** (struct device *dev)
  - Determine whether the device (dev->p->knode_driver) has been added to the head of a driver 's list
  - If not , then it is added as directed driver of the head of the list (the dev-> driver-> p-> klist_devices), can be seen from this , a device can only be bound to a driver, and a driver the can mount multiple a device
  - calling driver_deferred_probe_trigger processing delay probe mechanism , here temporarily concerns
  - If the device is part of a Bus, then Bus to send a message on the notification chain BUS_NOTIFY_BOUND_DRIVER

device_release_driver

detach device from driver .

The specific execution logic is as follows :

__device_release_ driver (struct device *dev)

- Get the driver pointed to by device-> driver
- Call driver_sysfs_remove to delete the 2 directory links created in device_bind_driver
- If the device belongs to a Bus , then Bus to send a message on the notification chain BUS_NOTIFY_ UN BIND_DRIVER
- If the device is part of a Bus and the Bus defines remove function , which is called Bus of the remove function ; otherwise, if the device-> driver pointed to by the driver defines remove function , then call driver-> remove
- Call devres_release_all to handle related things , and don't know the details for the time being
- Set device->driver and device->driver_data to NULL
- The Device (dev-> p-> knode_driver) from driver to delete the list of
- If the device is part of a Bus, then Bus to send a message on the notification chain BUS_NOTIFY_ UN BOUND_DRIVER

device_attach

It means that a certain device is added to the kernel , try to bind the device to a certain driver, the specific execution logic is as follows :

- If device->driver has pointed to a driver, and the device is not added to the head of any driver 's linked list , call device_bind_driver
- Otherwise , if device->driver is NULL, the following logic is executed for each driver under the Bus :
  - First call Bus defined match function , determining driver with the device match . If a match is not directly returned
  - If a match , then call driver_probe_device (in this section analysis below)

driver_attach

It means that a certain driver is added to the kernel , and try to bind the driver to one or more devices under the Bus . The specific execution logic is as follows :

driver_ attach(struct device_driver *drv)

- for Bus each under the device, perform the following logic :
  - first call Bus defined match function , determining driver with the device match . If a match is not directly returned
  - ✋➚ it matches , and the device->driver is NULL, call driver_probe_device . (Analyze below in this section)

driver_probe_device

driver_probe_ device(struct device_driver *drv, struct device *dev)

Try to bind drv and dev, the specific pointing logic is as follows :

- First , determine device is already registered , namely dev-> kobj.state_in_sysfs whether 1, if it is not 1, an error is returned
- If it is 1, then call really_probe(dev, drv) to handle subsequent things (analyzed below in this section)

really_probe

really_ probe(struct device *dev, struct device_driver *drv)

The specific execution logic is as follows :

- Assign dev->driver to drv
- calls pinctrl_bind_pins (dev), this function is mainly to do is to initialize the device associated GPIO, for example, we define the LCD device when , specifies the device uses what GPIO, what multiplexing mode , this code will Automatically configure these GPIOs to the correct state . This function depends on the GPIO subsystem . We will have a special article to detail the GPIO subsystem .
- calls driver_sysfs_add (dev) create some / sys / directory link under , in front of the function description
- If the **bus defines the probe function , call the probe function of the Bus** , otherwise if the driver defines the probe function , **call the probe function of the driver**
- Call driver_bound(dev) for processing , this function was introduced earlier

8.4 Key code analysis

bus_register

bus added to the kernel by bus_register interface implementation , the interface is the main function is to **create some folders** and a **few properties file** , is relatively simple , execution logic is as follows :

- is bus_type in struct subsys_private type pointer allocated space , And update the two pointers priv->bus and bus-> p to the correct values
- initialization notification chain priv-> bus_notifier
- Next manually created kset manner (can be created manually look back kset the DEMO), creating priv-> subsys , this kset behalf of Bus, i.e., / sys / bus / xxx in xxx directory
  - The priv-> subsys.kobj 's name is initialized to bus_type-> name, the name is xxx directory name directory
  - The priv-> subsys.kobj of kset initialized to bus_kset , bus_kset is created by the system , represents the / sys / bus directory (if you do not remember , back look " / SYS / top-level directory under " chapter), the kobject feature shows , if you do not specify priv-> subsys.kobj the parent, the kset will as its parent. in fact, too , so all the Bus will be present in the / sys / bus directory
  - The priv-> subsys.kobj of ktype initialized to bus_ktype, the ktype is created by the system . We also define themselves through ktype (created manually kobject of DEMO in), and therefore know ktype responsible for freeing resources and read / write priv-> subsys.kobj property files under (i.e., / sys / bus / xxx / properties file under).
  - calling kset_register the priv-> subsys registered with the kernel , after the call is completed , will be in the / sys / bus / generate the next xxx directory
- call bus_create_file in / sys / bus / xxx created under a uevent properties file .
  - The properties file is a system-defined , with BUS_ATTR this macro definition , we have 4.6 as described in section about this macro , back at 4.5, 4.6 Liang section , you should understand the user space through sysfs write the property file process .
  - But this system created uevent properties file , only write functions , read function is NULL, which means that users can only write to the properties file space
  - When the user space to write a file to the property kobject_action when the type of string , such as KOBJ_ADD, write function attributes file calls kobject_uevent , for priv-> subsys.kobj this kobject, reported a kobject_action type of Uevent events , such as reporting KOBJ_ADD event .

- As for what is the use of reporting this incident , it is unclear for the time being

- Call kset_create_and_add to create the kset priv->devices_kset , devices_kset corresponds to the directory named " devices " in the /sys/bus/xxx directory . All devices under Bus will create a link in this directory
- call kset_create_and_add create priv-> the Drivers _kset this kset, the Drivers _kset correspondence / sys / bus / xxx directory named " the Drivers ' directory . Bus all under the drivers, creates its own directory in the directory
- initialization priv pointer of the mutex , klist_devices and klist_drivers variables
- Call add_probe_files to create 2 attribute files in the /sys/bus/xxx directory . These two attribute files are also defined by the system with the BUS_ATTR macro .
  - The first attribute file is drivers_probe , it only write function , when the user space to write a file attribute that belongs to Bus 's device when the name , would end up calling *device_attach* , the trigger system to find the device corresponding to the driver, match After uploading , call driver->probe to initialize
  - second attribute is file drivers_autoprobe , provides read and write functions , a read / write priv-> drivers_autoprobe
- call bus_add_groups in / sys / bus / xxx create a directory bus_type-> bus_groups properties file defined in the group

bus_add_device

The kernel provides the device_register interface to add a device to a Bus, device_register will call device_add, and device_add will call bus_add_device to handle things related to the Bus side .

The processing logic of bus_add_device :

- calls internal device_add_attrs interfaces , by bus-> dev_attrs pointer defined default property group added to the kernel , they will be out now device directory is located at
- calls internal device_add_groups interfaces , will BUS-> dev_groups add a pointer-defined default attribute group into the kernel , they will appear in the device under the directory where
- call sysfs_create_link interfaces , the device in sysfs directory , linked to the bus of the devices directory , e.g.

```
[redacted]:~$ ls /sys/bus/platform/devices/i8042 -l
lrwxrwxrwx 1 root root 0 Aug 13 20:13 /sys/bus/platform/devices/i8042 -> ../../.
./devices/platform/i8042
```

  / sys / devices / platform / i8042 as the device truly directory where , for the convenience of management , the kernel device belongs Bus of devices creates a symbolic link directory , in front of the introduction bus_type data structures already mentioned this time .
- Invoke the sysfs_create_link interface , in the sysfs directory of the device (such as /sys/devices/platform/ i8042 /) , create a link to the bus directory where the device is located , and name it subsystem , for example

```
[redacted]$ ls /sys/devices/platform/i8042/subsystem -l
lrwxrwxrwx 1 root root 0 Aug 13 20:13 /sys/devices/platform/i8042/subsystem -> .
./../../bus/platform
```

- Finally , without a doubt , make this device a pointer attached to bus-> priv-> klist_devices list head in , so , you can by the head of the list to find Bus all under devices.

bus_add_driver

The kernel provides the driver _register interface to add a driver to a Bus, and the driver _register will call bus_add_ driver to handle things related to the Bus side .

The processing logic of bus_add_driver :

- Deal with driver-> driver_private related things , struct driver_private will be explained in detail in the chapter introducing the driver . This structure contains the kobject corresponding to the driver and some linked list headers.
  - for the driver of the struct driver_private pointer (PRIV) allocate space
  - initialize priv-> klist_devices head of the list , the list used to store driver all of the corresponding device, that is to say a driver can communicate with multiple device matching . But later we will know , a device can only match a driver
  - initialize priv-> driver points to the driver
  - initialize the driver-> p pointing to the priv
  - The priv-> kobj.kset assigned BUS-> p-> drivers_kset , then call kobject_init_and_add provided kobj-> ktype is driver_ktype, and to the kobj added to the kernel . After this step is finished , will be in / sys / bus A directory corresponding to the driver is generated under the /xxx/drivers directory . The name of the directory is driver->name . For example, " /sys/bus/platform/drivers/i8042 "
  - The priv-> knode_bus added to a bus-> p-> klist_drivers list head , that is, the driver attached to the Bus the drivers at the list head
- If the driver of Bus of drivers_autoprobe is 1, then call *driver_attach* , to see if the driver can and Bus in the device on the match , if the match , calling driver-> probe initialization
- call driver_create_file interfaces , in sysfs that the driver under the directory , create uevent properties files , in this property file bus.c with DRIVER_ATTR_WO (uevent) defined . This property file with the / sys / bus / xxx under uevent similar properties file , when the user writes a file to the property kobject_action time , will trigger the system for the driver 's kobject reported a kobject_action type of uevent event .
- call driver_add_attrs interfaces , in sysfs that the driver directory , created by the bus-> drv_attrs attributes defined set of pointers
- If driver-> suppress_bind_attrs is 0, two attribute files will be created in the driver directory
  - the bind attribute file , in bus.c with DRIVER_ATTR_WO (bind) is defined , when the user space to write that belong to the Bus one of the device 's name when , will call *driver_probe_device* , trigger the system to match the driver and the device, if the match is successful , Will call driver->probe to initialize .
    This function is the same asThe driver_probe attribute file under /sys/bus/xxx is very similar . The difference is that when the name of the device is written to the drivers_probe , the system will try to match the device and all the drivers under the Bus ; and when the name of the device is written to the bind , The system will only try to match the device with the driver
  - unbind properties files , in bus.c with DRIVER_ATTR_WO (unbind) is defined , and bind attribute file Instead , write device 's name, will call *device_release_driver* , lifting device and driver bindings

bus_probe_device

The kernel provides the device_register interface to add a device to a Bus, device_register will call device_add, and device_add will call the bus_probe_device function .

The processing logic of bus_probe_device :

- If the Bus of drivers_autoprobe to 1, will call *device_attach* to try to match the device and Bus certain under the driver.

8.5 Demo

We intend to define such a Demo, first create a bus (Bus), under which bus , create 1 attribute file ; then to the Bus add a device and a driver, observation device and driver matching conditions .

This demo will be completed several times . In this chapter , we will only create a bus and add the property file under the bus . In the subsequent chapters , we will add device and driver to the bus . You can learn about each bus through git log The details of a submission .
https://gitlab.com/study-linux/linux_driver_model/blob/master/bus_device_driver.c (https://gitlab.com/study-linux/linux_driver_model/blob/master/bus_device_driver.c)

9. Class

9.1 Introduction

A class is a high-level view of a device , which abstracts the commonality of the device .

The concept of class is not easy to understand . For example : some people of similar age and need to acquire similar knowledge , gather together to study , constitute a class (Class) . This class can have its own name (such as 295) , But if you leave the students (device) that make up it, it has no meaning . In addition , what is the greatest significance of the existence of the class ? It is every course taught by the teacher! Because the teacher just say again , a class of students can hear . Otherwise (for example, every student learning at home) , it is necessary for each person please a teacher , teaching again . And speaking of content , mostly the same , This is a great waste .

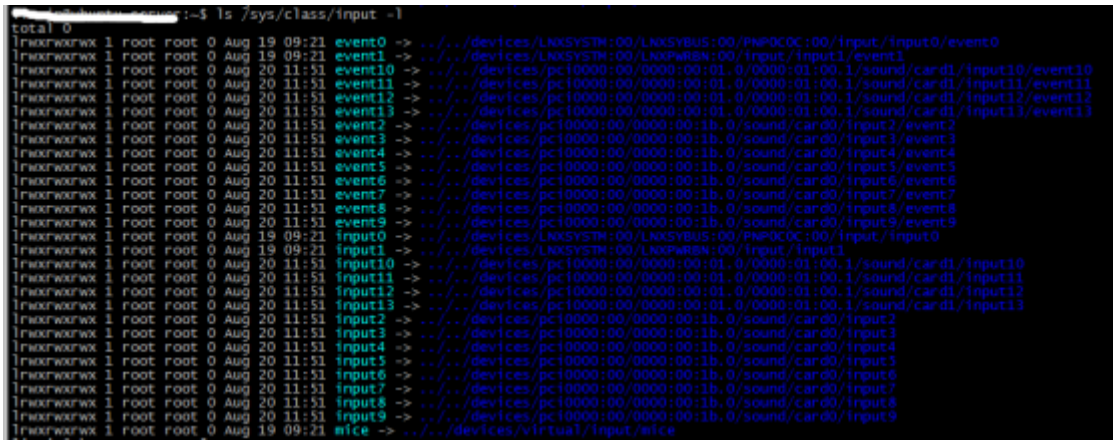
Device model Class provided the same function , such as some similar Device (student) , the need to provide similar interfaces (courses) to the user space , if the driver for each device to achieve it again, then , it would have led to the kernel A lot of redundant code , this is a huge waste . So , Class said , I will help you implement it , and you can use it.

Almost all classes are displayed in the /sys/class directory . For example, input device /sys/class/input; serial device /sys/class/tty.

An exception is block device , for historical reasons , Block class is / SYS / Block . The so-called block class is the name for the " Block " of the class.

Members of the class codes are typically controlled by the upper layer , without explicit support from the driver . In rare cases , a driver needs to handle class

Quick experience class



- /sys/class: Except block class, all classes are in this directory
- /sys/class/input: input is a class
  - In this class directory , create each belonging to the class of device symbolic links .
    The advantage of this link is that user space can more easily access device- related features (that is, property files) through sysfs , because the path name is short .
  - If the device belongs to the class, in the device directory is located , create a link to this class meets links , link name is " the Subsystem " , the figures do not show , are interested can look at themselves

9.2 Main data structure

class

The kernel abstracts a class with struct class , and its definition is as follows
/* include/linux/device.h, line 332 */

struct class	Comment
const char *name	Class name Corresponds to the name of the directory under /sys/class/ .
struct module *owner	The role is unclear
struct class_attribute *class_attrs	The class default properties , when creating class time , will be in the class directory (/ sys / class / xxx) create these properties files
const struct attribute_group **dev_groups	Belonging to the class of device default attributes , when calling device_register add a device when , if the device belongs to the class time , will device creates these properties files directory
struct kobject *dev_kobj	Indicates that the device under this class is in the directory under /sys/dev/ . Now there are generally two char and block . If dev_kobj is NULL , char is selected by default
int (*dev_uevent)(struct device *dev, struct kobj_uevent_env *env)	When the device is added to the kernel , if the device belongs to a certain class, it will call class->dev_uevent to add some event information . We will explain the details of event reporting when we talk about the device , where the function will be involved
char *(*devnode)(struct device *dev, umode_t *mode)	Callback to provide the devtmpfs Obtain the path of device node file, and then report the result to "DEVNAME=%s" in the format of uevent event information

void (*class_release)(struct class *class)	Callback function used to release the class Why is there no similar release bus function in bus_type ? The reason is simple , nature will come , whether it is Bus or Class, is a kobject, look back kobject chapter , we said , kobject released by its ktype completed . Bus corresponding ktype is bus_ktype (bus.c); Class corresponding ktype is class_ktype (class.c), bus_ktype-> Release is invoked directly kfree release Bus, class_ktype-> release calling class-> class_release release Class, so here There is the class_release function
void (*dev_release)(struct device *dev)	For release class back into the device transfer function . In device_release interface , checks Device where the class , whether registered release the interface , if you call the appropriate release interfaces release device pointer
int (*suspend)(struct device *dev, pm_message_t state)	Power management related
int (*resume)(struct device *dev)	Power management related
const struct kobj_ns_type_operations *ns_type	It is related to the namespace in sysfs , not detailed here There is a similar item in the K type data structure
const void *(*namespace)(struct device *dev)	It is related to the namespace in sysfs , not detailed here There is a similar item in the K type data structure
const struct dev_pm_ops *pm	Power management related
struct subsys_private *p	Remember , this data structure was introduced in the Bus chapter . Below we will introduce which parts of this data structure are used by class

subsys_private

We introduced this data structure comprehensively in the Bus chapter , here only a few items related to class are briefly listed .

/* drivers/base/base.h, line 28 */

struct subsys_private	Comment
struct kset subsys	kset exists in the form of a directory in /sys/ , which represents this Class, for example, /sys/class/input, which represents the input directory , and the name of the directory is specified in the struct class
struct kset *devices_kset	Unused
struct klist klist_devices	Linked list head , all devices under this Class will be mounted under this linked list
struct kset *drivers_kset	Unused
struct klist klist_drivers	Unused
struct list_head interfaces	The head of the linked list , all the class_interfaces under the Class will be mounted under this linked list . The role of class _interface is described below
struct mutex mutex	And interfaces list head fitted , used in mutex

struct bus_type *bus	Unused
struct class *class	Save the upper Class pointer
struct blocking_notifier_head bus_notifier	Unused
unsigned int drivers_autoprobe: 1	Unused
struct kset glue_dirs	Class is useful . It seems that it should correspond to a directory . The official explanation is to avoid namespace conflicts . The details are not yet clear.

class_interface

The role of class_interface is : you can register a class_interface to a class (all registered class_interfaces will be mounted under the head of the class->interface list), so that when a device is added or removed under the class , it will call class_interface-> add_dev or class_interface->remove_dev.
As for the use of these two functions , it is up to the specific implementer to decide .
/* include/linux/device.h, line 4 68 */

struct class _interface	Comment
struct list_head node	Used to mount to the class->interface list header
struct class *class	Point to the class it belongs to
int (*add_dev) (struct device *, struct class_interface *)	When a device is added to the class time , will the device is passed to add_dev function
void (*remove_dev) (struct device *, struct class_interface *)	When a device from class is removed , will the device is passed to remove_dev function

9.3 Main API description

Class

APIs related to Class mainly include :

- Registration and cancellation of class
- Register class_interface with class , to monitor the addition / removal of device
- when a device under this class is registered to the kernel
- This Handling when a device in class logs out of the kernel

Header file : inc lude/linux/device.h
Implementation file : drivers/base/class.c

class API	Comment
class_register(class)	It is a macro , will directly call __class_register. Used to add a class to the kernel . Calling this API requires self-allocation of class memory space
class_unregister(struct class *class)	Remove a class from the kernel
class_create (owner, name)	Allocate class memory space and put the class to the kernel name is the class name , which is / sys / class / xxx name

class_destroy(struct class *cls)	Call class_unregister to remove a class from the kernel . This API is used to destroy the class created by class_create
class_interface_register(struct class_interface *)	Register a class_interface to the class
class_interface_unregister(struct class_interface *)	Remove a class_interface from the class

9.4 Key code analysis

__class_register

class registration , by __class_register interface implementation , its processing logic and bus similar registration , including :

- is a class structure struct subsys_private type pointer (CP) to allocate space , update CP -> class and class -> P two pointers to the correct value
- initialization CP-> klist_devices list head
- initialization CP-> in the interfaces list head
- Call kset_init to initialize the kset cp->glue_dirs , note that it is only initialized , but not added to the kernel , so you can't see the directory
- initialize the mutex cp-> mutex
- provided this class corresponding to the kobject (CP-> subsys.kobj) the name of class->name
- If class-> dev_kobj is NULL, set it to sysfs_dev_char_kobj (i.e. /sys/dev/char)
- it is not a block class, setcp-> subsys.kobj .kset is class_kset (i.e. / sys / class), so that , as long as the cp-> subsys.kobj this kobject when added to the core , set its parent is NULL, the system will put kset of kobject do For its parent.
  The result is : for block classes, the directory is /sys/block, for other types of classes, the directory is /sys/class/xxx
- Set cp->subsys.kobj.ktype to class_ktype , the ktype is created by the system (class.c) . So know that the ktype will be responsible for releasing resources and reading / writing the property file under cp->subsys.kobj (ie (Properties file under /sys/ class /xxx/)
- Call kset_register to initialize the kset cp->subsys and add it to the kernel .
- call add_class_attrs the class-> class_attrs add default attributes defined in the / sys / class / xxx / directory

__class_create

The prototype of this function is

struct class *__class_ create(struct module *owner, const char *name,
struct lock_class_key *key)

Several important points can be seen from this prototype :

- main parameters are the name and module of the class
- returned result is a class

One sentence summary : Given a class name and module parameters , it can help you create a class.

The specific logic is as follows :

- First allocate the memory space of a struct class

- initialization class-> name and class-> module
- initialization class-> class_release as class_create_release , class_create_release is a system-defined (class.c), it is actually used kfree release allocated memory space first step
- call __class_register register the class

Here I would like to say a few words , __class_create equivalent to give an example , tell you how to manually create a most basic class. You can also be the basis of it , the design of more complex class, such as setting class default properties , design **dev_release** function , designed to suspend / resume power management related functions and the like .

device_add_class_symlinks

When device_register is called to add a device to the kernel , if device->class is not NULL, this function will be called to handle class- side related matters .

Bus there is also a similar function bus_add_device, except that bus_add_device in bus.c defined , and device_add_class_symlinks in drivers / base / core.c implemented in .

I believe you guessed it , this function is to create a symbolic link , let's take a look at the specific logic :

- First , the device directory is located , create a link to the class symbolic link , the link name is " the Subsystem "
- If device-> parent is not NULL, and the device is not a block device, then the device create a link to the directory where the parent link , link name is " device " .
  Very strange , this symbolic link has nothing to do with the class , why should it be created in this function ? I don't know the reason
- If the device belongs to class not block class, then the class directory , create a link to the device in line with the link , link name is the device 's name
- If the device belongs to class is a block class, is not in / sys / block create a link to the next device in line with the link . **Because for the block device, it's true that in the directory / sys / block down** .

device_add_to_class_ klist_devices

Uh , how come there is a function with such a long name ? Well , I lied to you , there is actually no such function . But this thing is still to be done , the device belongs to a certain class, and it must be added to the list head of class->klist_devices .

This thing is device_add function inside to do , direct look good code

```
if (dev->class) {
mutex_lock(&dev->class->p->mutex);
/* tie the class to the device */
klist_add_tail(&dev->knode_class,
     &dev->class->p->klist_devices);

/* notify any interfaces that the device is here */
list_for_each_ entry( class_intf,
   &dev->class->p->interfaces, node)
if (class_intf->add_dev)
class_intf->add_ dev( dev, class_intf);
mutex_unlock(&dev->class->p->mutex);
```

}

Very simple , two things

- The device is added class-> klist_devices list head
- call class-> interface all mounted on class_interface the add_dev function , tell them there is a device added to this class up .

9.5 Demo

We intend to design such a Demo : Create a new basic class, the so-called basic meaning , is counterfeit __class_create , only provide some of the necessary functionality , you can in this Demo based on , creating a more complex class.

Then add one to the class device .

https://gitlab.com/study-linux/linux_driver_model/blob/master/class_device.c (https://gitlab.com/study-linux/linux_driver_model/blob/master/class_device.c)

10. Device

Finally ..., we started to pay attention to things that are closely related to our daily work .

device and device_driver, when we introduced Bus and Class earlier , we have mentioned these two concepts countless times , and then we will introduce these two parts in detail.

10.1 Introduction

device and device _ Driver is a Linux basic concept-driven development , driver development is the idea is very simple : the development of specific software (device_driver) to drive the specified device (device).

So device is used to describe a device .

device_driver is used to describe how to make this device work .

10.2 The main data structure

device

/* include/linux/device.h, line 660 */

struct device	Comment
struct device *parent	The device's parent device . parent One of the roles is to organize the device hierarchy , in front of us had kobject one of th e main functions of the organization / sys / directory hierarchy under , this rule change . In fact , often device-> kobject-> parent = device-> parent-> kobject way , to organize the device hiera rchy
struct device_private *p	Private data structure pointer , from this structure we can see which linked list a device will be linked to . This structure will be introduced later
struct kobject kobj	The kobject corresponding to this device corresponds to a directory in sysfs

const char *init_name	Like bus, class , like , each device needs to have a name, this name reflects the directory name , that is, device-> kobject-> name. There are two ways to specify device->kobject->name : 1. It is the init_name here , which will be assigned to device->kobject->name at the end 2. If init_name is not specified , " bus->dev_name+devic e->id " will be assigned to device->ko bject->name
const struct device_type *type	And ktype&kset to kobject some similar This structure will be introduced later.
struct mutex mutex	Mutex
struct bus_type *bus	Which bus this device belongs to
struct device_driver *driver	Which device_driver does this device match?
void *platform_data	Private data , the device core code will not touch this data .
void *driver_data	A device_driver implementation code , may store some driver -related data to this place . The device core code layer will not touch this data . Set interface function : void dev_set_ drvdata(struct device *dev, void *data) Or the interface function : void *dev_get_drvdata(const struct device *dev)
struct dev_pm_info power	Power management related
struct dev_pm_domain *pm_do main	Power management related
struct dev_pin_info *pins	And GPIO (PINCTRL) data related subsystems , defines this device uses what GPIO pins , pin mu ltiplexing function is like . As we mentioned earlier , when the device and device_driver are matched , in the really_probe function , pinctrl_bind_pins (dev) will be called to automatically configure these GPIO pins.
int numa_node	"NUMA " function , not described yet
u64 *dma_mask ~ struct dev_archdata archdata	DMA related , not described yet
struct device_node *of_node	device tree related , for through the device tree to pass some device -related information , such as the register address, etc. . We have dedicated an article to describe the device tree It will also be used to match device and driver
struct acpi_dev_nodeacpi_node	Represents the Advanced Configuration and Power Management Interface , and ACPI device -r elated It will also be used to match device and driver
dev_t devt	Contains the main / minor number , when this number is not 0 Shi , will be in the / sys / dev / * / create the corresponding directory in the directory , the directory name is " major number : This device number " . At the same time , the corresponding /dev/xxx device node will also be generated
u32 id	The device 's id, may be > dev_name bus- binding generating device name This id can be specified manually , may be composed of a bus is automatically assigned
spinlock_t devres_lock struct list_head devres_head	Related to device resource management . Used to automatically release the resources occupied by the device , we will use a special chap ter to introduce device resource management
struct klist_node knode_class	Used to mount this device to the class->subsys_private-> klist_devices linked list

struct class *class	Which class does this device belong to
const struct attribute_group ** groups	Attribute group of this device
void (*release)(struct device *dev)	release the device
struct iommu_group *iommu_group	Do not describe
bool offline_disabled:1	Set this device supports hot-swappable , as 1 said they did not support hot swap
bool offline:1	Hot-swap-related , mark this device is in the online state or offline state

device_private

The official interpretation of this structure is : Structure to hold the private to the driver core portions of the device structure .

Obviously you can see , the structure is a struct device part of , but the structure inside the content will only be equipment usage model of the core code . We will not go touch the data structure in the preparation of the driver . This is perhaps the struct device peeling out of this part , defined separately as a reason for the structure .

/* drivers/base/base.h , line 71 */

struct device_private	**Comment**
struct klist klist_children	List head , for mounting the present device to children device. That is, children_device->parent = this device
struct klist_node knode_parent	Used to mount this device to the head of the klist_children list of the parent device
struct klist_node knode_driver	Used in the present device mount it matches driver 's klist_devices list head in , as here can be seen , a diver below can mount a plurality of device. In the 8.3 section *device_bind_driver* have introduced this mount process function .
struct klist_node knode_bus	Used in the present device mounted to belong Bus of bus_type-> subsys_private-> klist_devices
struct list_head deferred_probe	Used to add this device to the deferred_probe_list linked list . The official explanation is very clear , just post it directly : entry in deferred_probe_ list which is used to retry the binding of drivers which were unable to get all the resources needed by the device; typically because it de pends on another driver getting probed first
struct device *device	Point to this device

Summarize where a device will be mounted

- Device 's parent device linked list
- ✋■ the linked list of the driver corresponding to the device
- ✋■ the linked list of the Bus to which the device belongs

Forehead , seems wrong ah , is not there a class you , yes

- ✋■ the linked list of the class to which the device belongs , but the entry used for mounting is not placed in the device_private structure , but in device-> knode_class

device_type

And ktype & kset is to the kobject some similar . Recall ktype and kset role , remember ?
ktype effect is the release of kobject resources , defined kobject default properties file , define kobject read the next attribute files / write function .
The function of kset is to aggregate similar kobjects and control the uevent event reporting behavior of these kobjects , such as which events cannot be reported (filter), and add unified event information for events that can be reported .

device_type also did a similar thing , let's take a look at its data structure
/* include/linux/device.h , line 501 */

struct device_type	Comment
const char *name	Which type of device this device belongs to , such as "partitions" and "disks" ; "mouse" and "event" If name is not NULL, the event information of DEVTYPE will be included in the reported uevent event
const struct attribute_group **groups	Default attribute group
int (*uevent)(struct device *dev, struct kobj_uevent_env *env)	Add unified uevent event event information
char *(*devnode)(struct device *dev, umode_t *mode, kuid_t *uid, kgid_t *gid)	Get path of device node file class data structure also has a similar function , they have the same function , are acquiring device node file path . The system will give priority to calls device_type in devnode, if device_type is not defined , the call belongs to class of devnode
void (*release)(struct device *dev)	Used to release resources
const struct dev_pm_ops *pm	Power management related

10.3 Main API description

Header file : include/linux/device.h
Implementation file : drivers/base/core.c

device API	Comment
extern void device_initialize(struct device *dev)	After using kzalloc to allocate a struct device memory space , call this API to initialize
extern int __must_check device_add(struct device *dev)	Add the initialized device to the kernel
extern void device_del(struct device *dev)	And device_add action contrary
extern int __must_check device_register(struct device *dev)	First call device_initialize, then call device_add Generally, we own definition of a device after , will call the API
extern void device_unregister(struct device *dev)	First call device_del Then call put_device and use the kref mechanism to start the release process

device_create_groups_vargs	It is not an API, is core.c an internal function definition . The main functions are : The struct device memory space is automatically allocated , this memory space is i nitialized , and the device is added to the kernel . If the execution is successful , return a struct device * type data structure
extern struct device * device_ create_vargs (…);	This API directly calls device_create_groups_vargs
struct device *device_create(struct cla ss * cls, struct device *parent, dev_t devt, vo id *drvdata, const char *fmt, …)	This API converts the parameters (**const char *fmt, …)** to va_list , and then calls de vice_create_vargs The API can be specified device belongs to class, generally we want to automaticall y create a device when , calls this API
device_create_with_groups	The difference between this API and device_create is that this API can specify the d efault attribute group of the device , namely device-> groups
void device_destroy(struct class *class, dev _t devt)	Destroy the device created by device_create

10.4 Key code analysis

device_initialize

This API is very simple , that is , to initialize some linked list headers , mutex locks and so on . The general logic is as follows :

- device-> kobj.kset = devices_kset .
  device_kset is a kset created by the system in core.c , which corresponds to the /sys/devices directory .
- kobject_init(&dev->kobj, &device_ktype)
  device_ktype is a ktype defined by the system in core.c
- other is that some list head or something , but to say

device_add

Very important API.

Let me analyze the API in detail , and the specific logic is as follows :

- If device->device_private (dev->p) is empty , call kzalloc to allocate memory space for dev->p , and initialize dev->p->device , dev->p->klist_children , dev->p- >deferred_probe
- If the definition of dev-> init_name , put dev-> init_name assigned to dev-> kobj-> name. Then dev-> init_name set to NULL. Means device_add later , if you want to obtain the device 's name, only from Dev- >kobj->name is obtained (through the function dev_name(dev)) , but cannot be obtained from dev- >init_name .
- If dev_name (dev) still is NULL, that is the previous step init_name is NULL, then check dev-> bus if there is , if there is , put " dev-> BUS-> dev_name + dev-> the above mentioned id " assigned to dev->kobj->name
- Check again , if dev_name(dev) is still NULL , that is, init_name is NULL, and the device does not belong to a certain Bus, the kernel does not allow this situation to occur and returns an error
- Call get_device_parent to get the parent->kobject of the device , which is used to organize the directory hierarchy

- If the device belongs to a class
  - ◆✋➚ the class belongs to is a block class
    - If dev-> parent is not NULL, and dev-> parent-> class are block classs, returns parent-> kobject, i.e. / sys / block / xxx /
    - Otherwise , return block_class.p->subsys.kobj , that is, /sys/block/ directory

    As can be seen from this , block class of device is present in / sys / block directory or a subdirectory . And blcok class of this class of directory is / sys / block (see class description section).
  - ◆✋➚ the class belongs to is not a block class
    - If dev-> parent is not NULL
      - If dev-> parent-> class is not NULL, and dev-> class-> ns_type is NULL, then return dev-> parent directory is located
      - otherwise , in dev-> parent directory where the new one called " dev-> class-> name " directory , return to this directory as this device parent directory
    - If dev-> parent is NULL, then the / sys / devices / virtual / under a new name " dev-> class-> name " directory , return to this directory as this device parent directory
- Otherwise , Device does not belong to a certain class
  - ◆ If dev-> parent is NULL, then take a look at this device is part of a Bus, if part of a Bus, will dev-> bus-> dev_root directory as this device the parent directory . We Bus chapter When introducing the bus_type data structure , it was mentioned that dev_root is the default parent device of all devices under this Bus .
  - ◆ If dev-> parent is not NULL, put the parent directory is located as this device 's parent directory
  - ◆ Otherwise it returns NULL

➢ If get_device_parent acquired directory is not NULL, then set the dev-> kobj.parent , otherwise dev-> kobj.parent is NULL up ,

➢ call kobject_add the dev-> kobj added to the kernel , add after completion , will be in dev-> kobj.parent create a corresponding directory under " device-> name " directory , if dev-> kobj.parent is NULL, will the dev-> kobj -> kset directory where as a parent directory , ie, / sys / devices.

➢✋➚ platform_notify is defined in the system , call this function to notify that a device has been added to the kernel

➢ call device_create_file In this device the directory resides , create a call **uevent** properties file , the properties file by the DEVICE_ATTR_RW (uevent) defined

- ■♆♒♏■ writing a kobject_action type string to this property file , it will trigger the kernel to send a uevent event of this type for dev->kobj
- When reading the properties file , will be dev-> kobj belongs kset-> uevent_ops-> uevent default event which defines information

➢ call device_add_class_symlinks deal with class -related things , in class chapter has introduced the function

➢ call device_add_attrs in this device to create some files in the directory where the property

- If dev-> class is not NULL, then create a class-> dev-> dev_groups defined property groups
- If dev-> device_type is not NULL, then create device_type-> Groups defined property groups
- Create a dev-> Groups defined property groups
- If dev-> bus is not NULL, and bus-> offline and bus-> online not NULL, and dev-> offline_disabled is 0, create a named **online** property file , the file attribute by the DEVICE_ATTR_RW (online) Definition .
  - ◆ Writing true to the online attribute file will cause the kernel to execute the device_online function , which will call dev->bus->online , then send the KOBJ_ONLINE event for dev->kobj , and finally set dev->offline = false

- Writing false to the online attribute file will cause the kernel to execute the device_off line function . This function will call dev->bus->off line, then send the KOBJ_OFF LINE event for dev->kobj , and finally set dev->offline = true
- ◆☼♏♋♎♓■♑ the property file will return the value of dev->offline .

- Call the bus_add_device function to handle related things on the bus side . This function is introduced in the Bus chapter
- Call dpm_sysfs_add and device_pm_add to handle PM related matters
- If the dev-> DEVT not 0, description of the device present the primary / secondary device number , then
  - ■☝❒♏♋♦♏ an attribute file named dev in the directory where the device is located , the attribute file is defined by DEVICE_ATTR_RO (dev) , read the attribute file , you will get the value of dev->devt
  - get this device in / sys / dev directory under : If dev-> class is not NULL, from dev-> class-> dev_kobj get (/ sys / dev / char or / sys / dev / block); otherwise the default is / sys / dev / char. then in / sys / dev / xxx create a link to this next device directory of links , link name from the " dev-> DEVT " decision
  - calling devtmpfs_create_node create a device node , the system starts , will devtmpfs mount / dev directory , so that we can have access to / dev / xxx up . As for why to use devtmpfs, please refer to Section 3 describes the chapter .
- If the device belongs to a Bus, is to Bus sent a notification on link BUS_NOTIFY_ADD_DEVICE message
- calls kobject_uevent (& dev-> kobj, KOBJ_ADD) send uevent event , later analyzes uevent Detailed process events sent
- Call bus_probe_device(dev) to start the matching between device and driver . This function is analyzed in the Bus chapter
- If dev-> parent is present , the present device is added parent of klist_children list
- calls the imaginary function device_add_to_class_klist_devices , will this device to join class -related list , the virtual function in class analyzed a chapter

device_create_groups_vargs

The function prototype is as follows

```
static struct device *
device_create_groups_ vargs( struct class *class, struct device *parent,
                dev_t devt, void *drvdata,
                 const struct attribute_group **groups,
                const char *fmt, va_list args)
```

The specific code logic is also very simple :

- ❄♒♏ class parameter cannot be NULL, otherwise an error will be returned . It proves that the API must require specifying the class to which the device belongs , otherwise , add a device by manually defining a struct device
- ➢✌●●□♍♋♦♏ the memory space of a struct device
- Call device_initialize to initialize the memory space
- dev->devt = devt
- dev->class = class
- dev->parent = parent
- dev->groups = groups
- dev->release = device_create_release , device_create_release is a function defined in core.c , call kfree(dev) directly
- Assign drvdata to dev->driver_data

- calls kobject_set_name_vargs (& dev-> kobj, fmt , args) set dev-> kobj 's name
- Call device_add(dev)

device_create

The function prototype is as follows :

```
struct device *device_ create( struct class *class, struct device *parent,
                    dev_t devt, void *drvdata, const char *fmt, …)
```

The logic is simple :

Convert the parameters ... to va_list, and then call device_create_groups_vargs

device_create_with_groups

The function prototype is as follows :

```
struct device *device_create_with_ groups( struct class *class,
                        struct device *parent, dev_t devt,
                        void *drvdata,
                        const struct attribute_group **groups,
                        const char *fmt, …)
```

The logic is the same as device_create , the only difference is that this API specifies groups.

kobject_uevent: device uevent event reporting process

Remember uevent event reporting process it , the simple point is , is to call kobject_uevent for one kobject reported a uevent event , the kobject must belong to a kset allowed to report uevent, kset main role is to filter out some of the uevent; for you can For the reported uevent, add unified uevent event information . If you don't remember , look back at the Uevent chapter .

Back to the device on , device is packaged kobject, its escalation process to follow the above rules . So you need to find dev-> kobj belongs kset. In device_initialize Code knew this analysis kset is devices_kset.

OK, find kset, we focus on analyzing which uevent event information this kset adds , the uevent filtering function will not be mentioned here .

devices_kset calls dev_uevent to add event information . The logic of this function is as follows :

- If dev-> devt is not 0, then add the "MAJOR =% U" , "MINOR = U%" , "% S = DEVNAME" , " the DEVMODE # =% O" , "U% = DEVUID" , "DEVGID =%u" these event information
- if (dev->type && dev->type->name) , add ("DEVTYPE=%s", dev->type->name) event information
- if (dev->driver) , add ("DRIVER=%s", dev->driver->name)
- call of_device_uevent Add Device Tree -related event information
- if (dev->bus && dev->bus->uevent) , call dev->bus->uevent to add related event information
- if (dev->class && dev->class->dev_uevent) , call dev->class->dev_uevent to add related event information
- if (dev->type && dev->type->uevent) , call dev->type->uevent to add related event information

put_device: device resource release process

We know , ktype is responsible for whether kobject resources of the occupied , so for the device is the same , when you call put_device is dev-> kobj reference count is 0 Shi , dev-> kob belongs ktype will begin to release the resources .

So first find the ktype corresponding to dev->kobj , and know that this ktype is device_ktype in the device_initialize code analysis .

The logic of device_ktype 's release function device_release is as follows :

- devres_release_all(dev) : release device all occupied by the device Resource , such as clock, interrupt number, and so on . Remember device data structure devres_head list head it ? It is mounted below this device all device resource. We have A dedicated chapter to analyze the device resource management mechanism
- if (dev-> release) dev->release(dev);
  else if (dev->type && dev->type-> release) dev->type->release(dev);
  else if (dev->class && dev->class->dev_ release) dev ->class->dev_release(dev);
- kfree(device -> device_private)

10.5 Demo

For the device to do both a demo:
The first is with the Bus Demo together , put a device mounted to the Bus at , and then observe the device and driver matching process . This Demo in , we manually define a struct device, then call device_register .
https://gitlab.com/study-linux/linux_driver_model/blob/master/bus_device_driver.c (https://gitlab.com/study-linux/linux_driver_model/blob/master/bus_device_driver.c)

The second one is with Class Demo create a device, mount it to the Class under , this demo in , we can call device_create to create the device.
https://gitlab.com/study-linux/linux_driver_model/blob/master/class_device.c (https://gitlab.com/study-linux/linux_driver_model/blob/master/class_device.c)

11. Driver

11.1 Introduction

Device is used to describe a device .
device_driver is used to describe how to make this device work
Uh , there seems to be nothing else to say ...

11.2 The main data structure

device_driver

/* include/linux/device.h, line 2 29 */

struct device_driver	Comment
const char *name	This driver 's name
struct bus_type *bus	BookBus to which driver belongs
struct module *owner	class also has a similar data structure , temporarily unclear role
const char *mod_name	used for built-in modules
bool suppress_bind_attrs	disables bind/unbind via sysfs When the driver is added to the kernel , bus_add_driver will be called Bus analysis chapter over this function , it will be based driver-> suppress_bind_attrs to decide whether to create value bind / unbind two properties files . To these two properties files written device 's name when , will start the device and driver matching / Dematch
const struct of_device_id *of_match_table	It is often used when write driver code , for specifying the present driver can matching device. Remember device with the driver whether the match , by the match bus-> decision , so bus-> match function must be used in this table
const struct acpi_device_id *acpi_match_table	Similar to above
int (*probe) (struct device *dev)	This interface functions for implementing the driver start logic , as a segment of code main function . When deive the driver after the match , the first execution is driver-> probe. probe function is to initialize the hardware used for some , is a device to get some device Resource, if the step error which , on behalf of the driver can not drive the device
int (*remove) (struct device *dev)	This interface function is used to implement the end of the driver logic . When the device is removed , the remove function of the corresponding driver will be executed.
void (*shutdown) (struct device *dev)	Power management related
int (*suspend) (struct device *dev, pm_message_t state)	Power management related
int (*resume) (struct device *dev)	Power management related
const struct attribute_group **groups	Driver 's default attribute group
const struct dev_pm_ops *pm	Power management related
struct driver_private *p	device_driver should also be a kobject ah , why did not see struct kobject data structure it ? Yes , in driver_private defined .

driver_private

/* drivers/base/base.h , line 46 */

struct driver_private	Comment
struct kobject kobj	The kobject corresponding to this device_driver
struct klist klist_devices	Linked list header , used to mount all devices under this driver
struct klist_node knode_bus	Used in the present device_driver mount a Bus lower

struct module_kobject *mkobj	Temporarily unclear
struct device_driver *driver	Point to the upper device_driver

11.3 Main API description

Header file :include/linux/device.h

Implementation file : drivers/base/ driver.c

device_driver API	Comment
extern int __must_check driver_register(struct device_driver *drv)	Add this device_driver to a Bus . The amount , the argument list which does not seem to see Bus ah , how do you know which added to the Bus it ? Yes , you must call this API before initializing the driver-> bus
extern void driver_unregister (struct device_driver *drv)	This will device_driver from a Bus removed , the API will eventually call bus_remove_driver , we did not introduce this function , very simple , are interested can read the code yourself

11.4 Key code analysis

driver_register

The logic of the API is relatively simple , and most important things are done in the bus_add_driver function , which we introduced in the Bus chapter .

The specific logic of the API is as follows :

- First check if device_driver(drv)->bus exists
- then check whether the following functions have in the bus and the driver repeats the definition , in general , Bus defined , will give priority to calls bus-> probe, without calling driver-> probe
  drv->bus->probe && drv->probe
  drv->bus->remove && drv->remove
  drv->bus->shutdown && drv->shutdown
- Check whether a driver with the same name (drv->name) has been registered under Bus . If there is already , is duplicate registration is not allowed
- Call bus_add_driver(drv) , this function will allocate driver_private space , initialize drv->kobj and add it to the kernel , join the linked list , create a property file , start the match between the driver and the device , etc. .
- call driver_add_groups create drv-> groups the default attribute group defined in
- calls kobject_uevent (& drv-> p-> kobj , KOBJ_ADD) send uevent event

kobject_uevent: driver uevent event reporting process

And device reporting uevent the same logic of events , first find drv-> p-> kobj of kset of .

bus_add_driver initialized this kset. But it is very simple , no kset_uevent_ops function , so the driver in reporting uevent time events , does not add any additional event information

Summary of matching timing of device and device_driver

We have introduced the details of these matching timings before , here is just a summary :

- When you call device_register to a device is added to the kernel (device_register -> device_add-> bus_probe_device function analyzed)
- When you call driver_register to a driver added to the kernel is (driver_register-> bus_add_driver function analyzed)
- when to / SYS / Bus / XXX / drivers_probe attribute file writing a device name when . (Bus_register the analyzed)
- when to / SYS / Bus / XXX / Drivers / XXX / the bind attribute file writing a device name when . (Bus_add_driver function analyzed)

11.5 Demo

This Demo is with the Bus Demo together , the driver mounted to a Bus next , and then observe the device and driver matching process . This Demo in , we manually define a struct device _driver , then call driver_register to the driver mounted to a Under the Bus .

https://gitlab.com/study-linux/linux_driver_model/blob/master/bus_device_driver.c (https://gitlab.com/study-linux/linux_driver_model/blob/master/bus_device_driver.c)

12. Platform system

12.1 Introduction

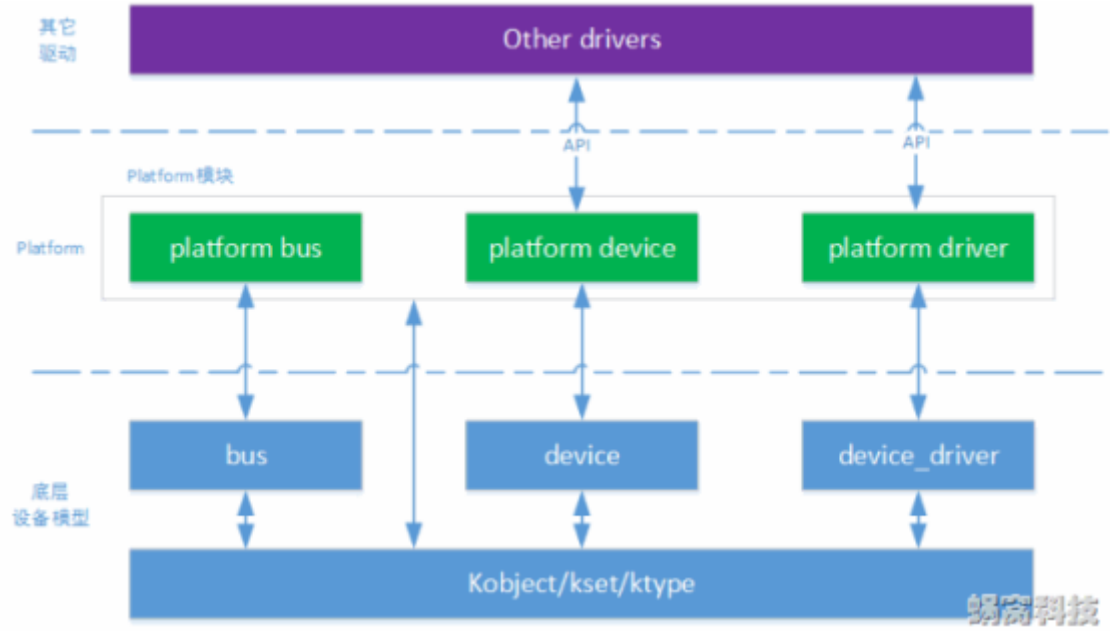
In the abstraction of the Linux device model , there is a type of device called " Platform Device " . These devices have a basic feature: they can be directly addressed through the CPU bus (for example, " registers " that are common in embedded systems) . It is the internal peripherals of the chip we often say , such as RTC, UART, LCD, SPI, I2C, etc. .

Accordingly , since the common , kernel model on the basis of the device (Device and the device_driver) , these devices were further packaging , abstract Paltform Bus , Platform Device and Platform Driver , in order to drive the developer can easily develop such Device driver .

It can be said , Paltform device for Linux driver engineer is very important , **because we write most device drivers , are designed to drive plaftom device** . In this article we take a look at Platform realize the device in the kernel .

Software architecture of the Platform module

The implementation of the Platform device in the kernel is located in the two files include/linux/platform_device.h and drivers/base/platform.c . Its software architecture is as follows



As can be seen from the picture , the implementation of the Platform device in the kernel mainly includes three parts :

Platform Bus, based on the underlying bus module , abstracts a virtual Platform bus for mounting Platform devices ;

Platform Device, based on the underlying device module , abstracts the Platform Device, which is used to represent the Platform device ;

Platform Driver, based on the underlying device_driver module , abstracts Platform Driver to drive Platform devices .

Among them, Platform Device and Platform Driver will provide packaged APs for other Drivers . For details, please refer to the following description .

12.2 Main data structure

platform_device

/* include/linux/platform_device.h , line 22 */

struct platform_device	Comment
struct device dev	platform _device is an encapsulation of struct device
const char *name	Device name , use your toes to know that you will eventually write the corresponding kobject->name
int id	Corresponding to device->id
bool id_auto	It represents id is not a platform automatic dispensing system of

struct resource *resource	resource , the resource description of the device , by struct resource (include/linux/ioport.h) Construction Abstract In Linux the , system resources, including the I / O , Memory , the Register , the IRQ , the DMA , Bus various types . These resources often has exclusive , do not allow a plurality of devices si multaneously , so Linux kernel provides a number of the API , for Allocate and manage these re sources . When a device requires the use of certain resources , simply use struct resource organize these resources (e.g., name, type, start and end address, etc.) , and stored in the device resource poin ter to . Then the device probe when , equipment needs will call the resource management inter face , distribution, use of these resources . the resource management logic cores , can determin e whether these resources have been used, whether can be used, and so on .
u32 num_resources	Represents how many device resources
const struct platform_device_id *id_entry	And device and driver -related match . Concrete can see platform_ bus the Bus of the match fu nction
char *driver_override	Driver name to force a match
struct mfd_cell *mfd_cell	And MFD content related equipment , temporarily Description
struct pdev_archdata archdata	A weird existence! ! Its purpose is to preserve some of the architecture related to data , to see arch / arm / include / asm / device.h in struct pdev_archdata defined structure , you know ho w this indulgent design garbage . Whether it up! !

platform_d river

/* include/linux/platform_device.h, line 173 */

struct platform_driver	**Comment**
struct device_driver driver;	platform_driver is a package of device_driver
int (*probe)(struct platform_de vice *)	device_driver defined function of the same name , it is clear , the function here is equivalent to covering device_driver function of
int (*remove)(struct platform_d evice *)	Same as above
void (*shutdown)(struct platfor m_device *)	Same as above
int (*suspend)(struct platform_ device *, pm_message_t state)	Same as above
int (*resume)(struct platform_d evice *)	Same as above
const struct platform_device_id *id_table	Noticed yet ? Platform_device also has the same type of a variable , it is clear , it is used for de vice and driver matching . You can view platform_bus the match function logic
bool prevent_deferred_probe	Delay probe mechanism , Detailed see struct device_private data structure **deferred_probe** th e described elements . From a literal description , it should refer to the prohibition of delayed probe function

platform_bus_type

The platform bus is an ordinary struct bus_type, and there is no new data structure

12.3 Main API description

P latform B us

Platform bus no new relevant API, all using Bus itself defined API, including the creation of a platform bus, destroyed the bus and so on .

Platform Device

Platform Device main distribution providing device, registration and other interfaces , for the other driver to use , comprises :

Header file : in clude/linux/platform_device.h

Implementation file : drivers / base /platform.c

platform device API	Comment
extern struct platform_device *platform_device _alloc(const char *name, int id);	Allocate a platform _device structure space and call device_initialize API initia lizeplatform_device.dev , and initialize the name, the above mentioned id, arc hdata and other elements .
extern void arch_setup_pdev_archdata(struct p latform_device *);	Initialize platform_device. archdata Previous API initialization archdata time , that this function call
extern int platform_device_add_resources(stru ct platform_device *pdev, const struct resource *res, unsigned int num);	Add some resources to a platform_device
extern int platform_device_add_data(struct pla tform_device *pdev, const void *data, size_t siz e);	The specified platfrom_device the platform data That is to assign a value to platfrom_device.dev. platform_data
extern int platform_device_add(struct platform _device *pdev);	Add a paltform_device to the kernel system , no doubt , the device_add functi on will eventually be called
extern void platform_device_del(struct platfor m_device *pdev);	To remove a platform_device from the kernel system , the main logic is to call device_del
extern void platform_device_put(struct platfor m_device *pdev)	Call put_device to reduce the reference count of platform_device.dev
extern struct platform_device *platform_device _register_full(const struct platform_device_info *pdevinfo);	Overall , the API means : Given a platform_device_info, the API will automatic ally create a platform_device and put the platfrom_device added to the kernel system . It will internally call alloc, add_resources, add_data, add, etc. mentioned abov e APIs It is noted that the first parameter is a structure platform_device_info , the stru cture is mainly described the platfrom_device number information , name, id, resources , etc.

static inline struct platform_device *platform_device_register_resndata(struct device *parent, const char *name, int id, const struct resource *res, unsigned int num, const void *data, size_t size)	A API variant , corresponding to the parameters provided in the form of a function name, id information . The API will encapsulate these parameters into a platform_device_info , and then call platform_device_register_full
static inline struct platform_device *platform_device_register_simple(const char *name, int id, const struct resource *res, unsigned int num)	Previous API variant , equivalent to provide some simple parameters , and then call platform_device_register_resndata
static inline struct platform_device *platform_device_register_data (struct device *parent, const char *name, int id, const void *data, size_t size)	The logic is the same as the previous API , except that the parameters are different , and platform_device_register_resndata will eventually be called
extern int **platform_device_register** (struct platform_device *)	After manually assigning a platform_device structure , you can call the API to add the device to the kernel . Note that the API will first call device_initialize to initialize the device, so this platform_device is generally allocated with kzlloc , rather thanallocated by platform_device_alloc , because device_initialize will also be called in platform_device_alloc
extern void **platform_device_unregister** (struct platform_device *)	The inverse function of the previous API
extern int **platform_add_devices** (struct platform_device **, int)	Note that the parameter is a double pointer , which means registering multiple at once platform_devices The API will call platform_device_register internally
extern struct resource *platform_get_resource (struct platform_device *, unsigned int, unsigned int)	Get the resource information in platform_device . First dicarboxylic parameter represents want to get resource type (DMA / IRQ , etc.) Di San This type of argument on behalf of want to get the resource number
extern int platform_get_irq(struct platform_device *, unsigned int)	And one API is similar , but here types have been identified , is the IRQ.
extern struct resource *platform_get_resource_byname(struct platform_device *, unsigned int, const char *)	By resource to get the name of the resource
extern int platform_get_irq_byname(struct platform_device *, const char *)	Get IRQ by name

Platform D river

Platform Driver provides the registration function of struct platform_driver . There is no allocation function . You need to manually allocate the structure space by yourself , as follows:

Header file : in clude/linux/platform_device.h

Implementation file : drivers / base /platform.c

platform_driver API	**Comment**

extern int platform_driver_register(struct platform_driver *)	Register a platform_driver into the kernel and eventually call driver_register
extern void platform_driver_unregister(struct platform_driver *)	Remove a platform_driver from the kernel , and eventually call driver_unregister
extern int platform_driver_probe(struct platform_driver *driver, int (*probe)(struct platform_device *))	For devices that are not hot-pluggable , the API should be used to register the driver corresponding to the device . Generally, the internal peripherals of the chip are not hot-swappable , so the driver code of the internal peripherals of the chip generally calls this API The API will eventually callplatform_driver_register , the difference from platform_driver_register is that it will do some processing for non-hot-swappable devices , see the code analysis below for details
static inline void *platform_get_drvdata(const struct platform_device *pdev)	Get the value of platform_device.dev. driver_data The API directly calls dev_get_drvdata You can look back at the description of driver_data in the struct device data structure
static inline void platform_set_drvdata(struct platform_device *pdev, void *data)	Set the value of platform_device.dev. driver_data This API directly calls dev_set_drvdata You can look back at the description of driver_data in the struct device data structure

Lazy API

It is registering the platform device again , and it is also registering the platform driver . It looks very verbose . But the kernel has thought of this , so it provides a lazy API , you can register the platform driver at the same time , and assign a platform device :

```
extern struct platform_device *platform_create_ bundle(
struct platform_driver *driver, int (* probe)( struct platform_device *),
struct resource *res, unsigned int n_res,
const void *data, size_t size);
```

As long as a platform_driver (put driver of probe interfaces explicitly passed) , and informs the resource occupied by the device information , Platform module will help allocate resources , and performs probe operation . For those who do not need the hot plug device That said , this method is the most trouble-free .

Early platform device/driver

I don’t know the role and working mechanism of the early platform for the time being . I will encounter it later to supplement this chapter . Related APIs include

```
extern int early_platform_driver_ register( struct early_platform_driver *epdrv,
  char *buf);
extern void early_platform_add_ devices( struct platform_device **devs, int num);

static inline int is_early_platform_ device( struct platform_device *pdev)
{
return !pdev ->dev.driver;
```

```
}

extern void early_platform_driver_register_ all( char *class_str);
extern int early_platform_driver_ probe( char *class_str,
    int nr_probe, int user_only);
extern void early_platform_cleanup(void);
```

12. 4 key code analysis

platform_bus_init

Obviously , the first step of the platform is to create a platform bus in the kernel system , and then you can add platform_device and platform_device to the bus. platform_driver

The code logic is very simple , as follows :

- early_platform_cleanup : make some preparations for the early platform , not detailed here
- device_register(&platform_bus)
  - calling device_register register a device, do not look at the name parameter is platform_bus, in fact, it is a struct device type .
  - ■❄♒♏ only effect of this sentence is to create a directory named platform under /sys/devices , all platform_devices will be stored in this directory, and the default parent device of all platform_device is platform_bus
  - Why can these words in / sys / devices to create a named under the platform directory it ? If you can understand quickly , explained previous knowledge has been absorbed . Platform_bus this device does not belong to any class, nor belong to any bus, There is also no parent device. According to the code analysis of the API device_add in the previous article , you can know that this device_register can create the platfrom directory under /sys/devices .
- bus_register(&platform_bus_type)
  - ■☝❒♏♋♦♏ a platform bus in the kernel system
  - About bus_type and bus_register more detail , in the Bus chapter has been introduced

platform_match

platform Bus of the match function , determines the Bus at platform_device (pdev) and platform_driver (pdrv) matching rule , it is quite important .

The main purpose of this section is given pdev and pdrv matching rule of the order , in this section can be read back to see more pdev and pdrv data structure , to facilitate understanding . DETAILED logic is as follows :

- If defined pdev-> driver_override (which is a string), the comparator pdev-> driver_override and pdrv-> driver-> name, identical to matching , different not match
- Attempt an OF style match
  - Determine whether pdrv->driver-> of_match_table and pdev->dev->of_node can match
- Then try ACPI style match
  - Determine whether pdrv->driver-> acpi_match_table and pdev->dev-> acpi_node can match
- If defined pdrv-> id_table , the comparator pdrv-> id_table .name and pdev-> name, the same is matching , different not match
- ❄♒♏ last item , return to the normal matching rules , and judge whether pdev->name and pdrv->driver->name are the same , the same will match , and the difference will not match

platform_device_register

Used to add a platform_device (pdev) to the kernel system .
The code logic is very simple :

- device_initialize(&pdev->dev)
- arch_setup_pdev_archdata(pdev)
- return platform_device_add(pdev) : This sentence is the key

platform_device_add

From the foregoing description we know , device which belongs to the bus is designated by the bus device-> decision . Obviously , all platform_device should mount platform bus under . Logically speaking , platform system should automatically to help us deal these things , we do not need to manually specify device-> bus up .

The platform system does this , the specific API is this platfrom_device_add, the logic is as follows :

- If pdev-> dev.parent == NULL, then pdev-> dev.parent assigned platform_bus . Remember platfrom_bus it , which confirms the mentioned earlier in this chapter , all platform_device default parent device is platform_bus
- pdev-> dev.bus = & platform_bus_type : the OK, all platform_device will mount platform Bus under
- Set the name of pdev-> dev.kobject , which is the name of the device .
  - If pdev-> ID > = 0, then the name of "pdev-> name.pdev-> id"
  - If pdev-> ID == PLATFORM_DEVID_NONE (-1), the name of "pdev-> name "
  - If pdev-> id == PLATFORM_DEVID_AUTO (-2), the system will automatically get an id assigned to pdev-> id , and the pdev-> id_auto set to true. Name is "pdev-> name.pdev-> id .auto "
- Process the resources defined in pdev , and specify the name and parent for each resource
- device_add(&pdev->dev) : add device to the kernel

platform_driver_register

Register a platform_driver (drv) into the kernel .
The logic is very simple , do some preparations , and then call driver_register . The specific logic is as follows :

- Drv-> driver.bus = & platform_bus_type : platform_driver are mounted in the platform bus at
- If you define Drv-> the Probe , will platform_drv_probe assigned to Drv-> driver.probe , platform_drv_probe is platform.c defined , it is clear that , when the device and the driver after the match , the kernel system calls driver-> probe, That is, platform_drv_probe here , then platform_drv_probe will definitely call drv->probe
- If defined Drv-> Remove , then platform_drv_remove assigned to Drv-> driver.remove , logic and the similar , are not repeated herein
- If defined Drv-> the shutdown , then platform_drv_shutdown assigned to Drv-> driver.shutdown , logic similar to the
- calls driver_register will platform_driver registered into the kernel

platform_driver_probe

It will eventually call platform_driver_register. The difference is that the API is for devices that are not hot-pluggable . The specific logic is as follows :

- dr v->prevent_deferred_probe = true: prohibit the delay probe mechanism of the platform_driver
- drv->driver.suppress_bind_attrs = true : do not generate bind/unbind two attribute files in the driver directory , so that it is not possible to write these two attribute files to start device/driver matching / unmatching , also in order not to deal with Hot swap
- Drv-> = probe probe (the probe is present API a parameter , a function pointer), and then call platform_driver_register . When the device with a driver after the match , will be called probe function
- ♑♒♏ next logic is to make this platfrom_driver no longer try to match other devices
  - drv->probe = NULL
  - Check the Drv-> driver.p-> klist_devices.k_list , if empty , prove this platfrom_driver no match to a device, an error is returned
  - Drv-> driver.probe = platform_drv_probe_fail , so that any other device attempting the driver match , returns failure

12.5 Demo

The idea of Demo is also very simple . The platform bus already exists and does not need to be rebuilt . We can create a new paltform_device and a platform_driver, add them to the kernel, and observe the matching process of the two .
https://gitlab.com/study-linux/linux_driver_model/blob/master/platform_device_driver.c
(https://gitlab.com/study-linux/linux_driver_model/blob/master/platform_device_driver.c)

13. device resource management

Finally , finally , all the equipment models are finally introduced ...
This chapter talks about a small thing , the idea is very simple , I believe it is easy to understand . However, as a driver engineer , the content of this section is very practical , can answer some confusions , and can make our code simple and concise .

13.1 Introduction

Let's look at an example first , don't look at it line by line , just glance at it

```
/* drivers/media/platform/soc_camera/mx1_camera.c, line 695 */
static int __init mx1_camera_ probe ( struct platform_device * pdev )
{
    …

    res = platform_get_ resource ( pdev , IORESOURCE_MEM , 0 );
    irq = platform_get_ irq ( pdev , 0 );
    if (! res || ( int ) irq <= 0 ) {
        err = - ENODEV ;
        goto exit ;
    }
```

```
clk = clk_get (& pdev -> dev , “csi_clk” );
if ( IS_ERR ( clk )) {
    err = PTR_ERR ( clk );
    goto exit ;
}

pcdev = kzalloc ( sizeof (* pcdev ), GFP_KERNEL );
if (! pcdev ) {
    dev_err (& pdev -> dev , “Could not allocate pcdev\n” );
    err = - ENOMEM ;
    goto exit_put_clk ;
}

…

/*
 * Request the regions.
 */
if (! request _mem_region ( res -> start , resource_size ( res ), DRIVER_NAME )) {
    err = - EBUSY ;
    goto exit_kfree ;
}

base = ioremap ( res -> start , resource_size ( res ));
if (! base ) {
    err = - ENOMEM ;
    goto exit_release ;
}
…

/* request dma */
pcdev -> dma_chan = imx_dma_request_by_ prio ( DRIVER_NAME , DMA_PRIO_HIGH );
if ( pcdev -> dma_chan < 0 ) {
    dev_err (& pdev -> dev , “Can't request DMA for MX1 CSI\n” );
    err = - EBUSY ;
    goto exit_iounmap ;
}
…

/* request irq */
err = claim_fiq (& fh );
if ( err ) {
    dev_err (& pdev -> dev , “Camera interrupt register failed\n” );
    goto exit_free_dma ;
}
```

```
    …
    err = soc_camera_host_register (& pcdev -> soc_host );
    if ( err )
        goto exit_free_irq ;

    dev_info (& pdev -> dev , "MX1 Camera driver loaded\n" );

    return 0 ;

exit_free_irq :
    disable_fiq ( irq );
    mxc_set_irq_ fiq ( irq , 0 );
    release_fiq (& fh );
exit_free_dma :
    imx_dma_free ( pcdev -> dma_chan );
exit_iounmap :
    iounmap ( base );
exit_release :
    release_mem_region ( res -> start , resource_size ( res ));
exit_kfree :
    kfree ( pcdev );
exit_put_clk :
    clk_put ( clk );
exit :
    return err ;
}
```

I believe that every engineer who has written a Linux driver has encountered the above confusion in the probe function : to apply for multiple resources in order (IRQ , Clock , memory , regions , ioremap , dma , etc.) , as long as any kind of resource If the application fails , it is necessary to roll back and release all the resources previously requested . So at the end of the function , there must be a lot of goto tags (such as exit_free_irq , exit_free_dma , etc.) , and when there is an error in the application of resources , carefully goto to the correct on the label , in order to release an application resource .

We mentioned this phenomenon in the article on kernel modules.In that article, we proposed two ways to deal with this problem : use goto or use status flags , and then unified processing in the module_exit function .

However , for the device resource, the kernel Linux device model by means of Device Resource Management (Device Explorer) , to help us solve this problem . Is : Driver You just apply on the line , regardless of release , my device model to help you release . Finally , we The driver can be written like this :

```
static int __init mx1_camera_ probe ( struct platform_device * pdev )
{
    …

    res = platform_get_ resource ( pdev , IORESOURCE_MEM , 0 );
```

```
    irq = platform_get_ irq ( pdev , 0 );
    if (! res || ( int ) irq <= 0 ) {
        return – ENODEV ;
    }

    clk = devm_clk_get (& pdev -> dev , “csi_clk” );
    if ( IS_ERR ( clk )) {
        return PTR_ERR ( clk );
    }

    pcdev = devm_kzalloc (& pdev -> dev , sizeof (* pcdev ), GFP_KERNEL );
    if (! pcdev ) {
        dev_err (& pdev -> dev , “Could not allocate pcdev\n” );
        return – ENOMEM ;
    }

    …

    /*
     * Request the regions.
     */
       if (! devm _request_mem_region (& pdev -> dev , res -> start ,
resource_size ( res ), DRIVER_NAME )) {
        return – EBUSY ;
    }

    base = devm_ioremap (& pdev -> dev , res -> start , resource_size ( res ));
    if (! base ) {
        return – ENOMEM ;
    }
    …

    /* request dma */
    pcdev -> dma_chan = imx_dma_request_by_ prio ( DRIVER_NAME , DMA_PRIO_HIGH
);
    if ( pcdev -> dma_chan < 0 ) {
        dev_err (& pdev -> dev , “Can't request DMA for MX1 CSI\n” );
        return – EBUSY ;
    }
    …

    /* request irq */
    err = claim_fiq (& fh );
    if ( err ) {
        dev_err (& pdev -> dev , “Camera interrupt register failed\n” );
```

```
        return err ;
    }


    …
    err = soc_camera_host_register (& pcdev -> soc_host );
    if ( err )
        return err ;

    dev_info (& pdev -> dev , "MX1 Camera driver loaded\n" );

    return 0 ;
}
```

How to do it ?

Pay attention to the interface at the beginning of " devm_ " above , the answer is there . Don't use those regular resource application interfaces anymore , use devm_xxx interfaces instead .

In order to maintain compatibility , the parameters of these new interfaces and the old interfaces are kept the same , but " devm_ " is added before the name , and a struct device pointer is added.

13.2 devm_xxx

Here are some of the common resources application interfaces , which by the respective Framework (such as Clock , Regulator , GPIO based, etc.) Device Resource Management implemented . When used , ignore " devm_ " prefix , the remaining rear portion , Driver engineers very familiar . just remember one thing , Driver can apply only , not released , the device will help model release . However, if strict order , the driver remove the time , you can take the initiative to release (also a corresponding interface , not listed here) .

```
extern void * devm_ kzalloc ( struct device * dev , size_t size , gfp_t gfp );

void __iomem * devm_ioremap_ resource ( struct device * dev ,
  struct resource * res );
void __iomem * devm_ ioremap ( struct device * dev , resource_size_t offset ,
  unsigned long size );

struct clk * devm_clk_ get ( struct device * dev , const char * id );

int devm_gpio_ request ( struct device * dev , unsigned gpio ,
  const char * label );

static inline struct pinctrl * devm_pinctrl_get_ select (
  struct device * dev , const char * name )

static inline struct pwm_device * devm_pwm_ get ( struct device * dev ,
  const char * consumer );

struct regulator * devm_regulator_ get ( struct device * dev , const char * id );
```

```
static inline int devm_request_ irq ( struct device * dev , unsigned int irq ,
  irq_handler_t handler , unsigned long irqflags ,
  const char * devname , void * dev_id );

struct reset_control * devm_reset_control_ get ( struct device * dev ,
  const char * id );
```

13.3 What is a device resource

A device can work , it needs to rely on many external conditions , such as power supply, clock, etc. , these external conditions are called device resources (device resouce) . For modern computer architecture , possible resources include :

a) power: power supply

b) clock: clock

c) memory , memory , generally allocated by kzalloc in the kernel

d) GPIO , user and CPU exchange simple control, status and other information

e) IRQ , trigger interrupt

f) DMA , data transfer without CPU participation

g) Virtual address space , generally allocated by ioremap , request_region, etc.

h) Wait

In the Linux kernel eyes , " resource " is defined more broadly , such as the PWM , the RTC , the Reset, can be abstracted as a resource , for the driver to use .

In earlier kernel in , the system is not particularly complex , and each framework is not working yet , so most of the resources by the driver to maintain their own . But with the increasing complexity of systems , driver situation more and more shared resources between , while demand for power management has become increasingly urgent . so kernel will each resource management rights to recover , based on " Device resource management " framework , by the respective framework unified management , including distribution and recovery

13. 4 Software Framework

After talking for so long , what exactly is the software framework of device resource management ? How does the management mechanism work, and which implementation code is it ? Next, let's look at these questions .

device resource management located " Drivers / Base / devres.c " in .

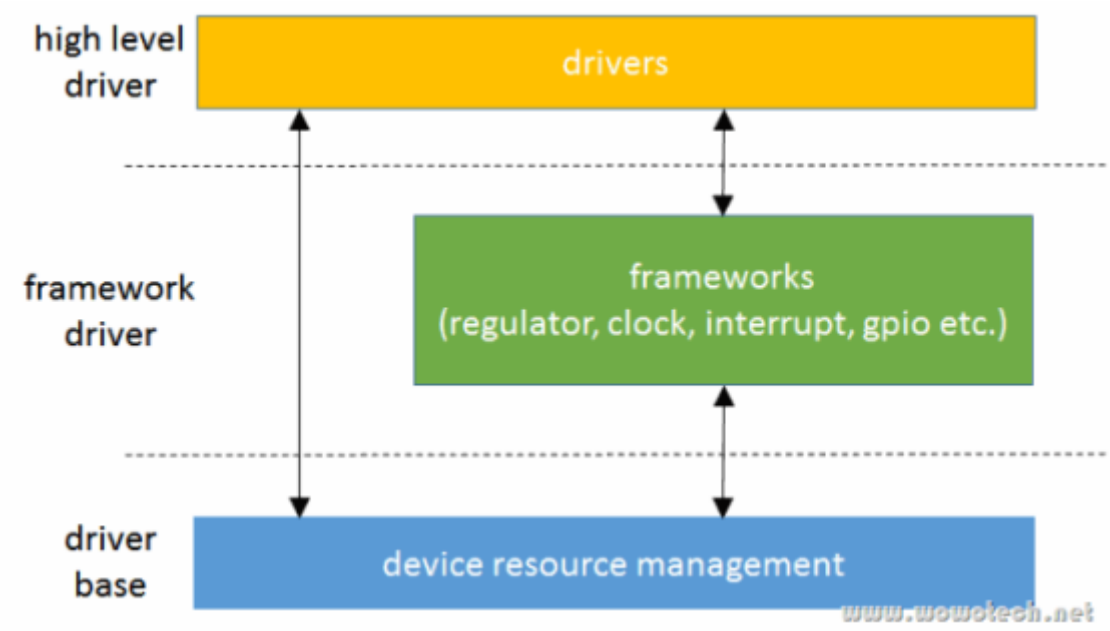
Its implementation is very simple , why ? Because there are many types of resources , forms are diverse , but devres impossible to informed , it can not carry out specific distribution and recovery .

Therefore , devres can do (is also its **only function**) , is :

Provides a mechanism , all the resources in the system of a device , in the form of a linked list , organized , in order to Driver the detach (*device_release_driver*) when , automatically released .

More specific things , such as how to abstract a certain device resource , are taken care of by the upper framework . These frameworks include : regulator framework (management of power resources) , clock framework (management of clock resources) , interrupt framework (management of interrupt resources) , gpio Framework (management gpio resources) , the PWM Framework (managing the PWM) , and so on .

Other Driver , located on the framework above , using the mechanisms and interfaces they offer (devm_xxx) , being developed is very convenient



13. 5 code analysis

Related data structure

Start struct device started ! Remember the introduction struct device mentioned data structure devres_head list head it ? Do not remember to look back , it is used to mount this device all use of resources.

What is the meaning of a single resource ?
In the " Drivers / Base / devres.c " in , the name struct devres , as follows :
Note : devres data structures associated , in devres.c defined in (a C file Oh!) . In other words , is transparent to the other modules . This is an elegant design (try to shield details)!

```
struct devres {
    struct devres_node node ;
    /* — 3 pointers */
    unsigned long long data [ ]; /* guarantee ull alignment */
};
```

This structure is very simple , the idea is very clear : struct devres_node describe different resources like parts ; Data [] This zero-length array for describing each resource different parts (Programming Basics- described article a zero-length of the array) .

Take a look at struct devres_node:

```
struct devres_node {
    struct list_head entry ;
    dr_release_t release ;
#ifdef CONFIG_DEBUG_DEVRES
    const char * name ;
    size_t size ;
#endif
```

```
};
```

Despite debug related matter , it is very clear :

A list_head is used to mount each resource to the head of the device->devres_head linked list .

What in the end is in the form of resources , Device Resource Management did not know , and therefore need the upper module provides a release back to the called function , for release resources

API for framework driver

Take a look at the picture in the software framework. The framework driver refers to the implementation part of a specific resource (such as IRQ) . It will provide its own API to a device driver (such as :devm_request_irq , devm_free_irq).

Framework driver is implemented based on device resource management . The API provided by device resource management to framework driver mainly includes :

devres_alloc/devres_free , devres_add/devres_remove

Combined with the description in the previous chapter , I guess you should be able to understand what is going on .

Let 's look at an example of using device resource management (IRQ module : devm_request_irq, devm_free_irq is the API provided by the IRQ module to the device driver, and it also uses the API provided by resource management) :

```
/* include/linux/interrupt.h */
static inline int __must_check
devm_request_ irq ( struct device * dev , unsigned int irq , irq_handler_t handler ,
                 unsigned long irqflags , const char * devname , void * dev_id )
{
        return devm_request_threaded_ irq ( dev , irq , handler , NULL , irqflags ,
                                        devname , dev_id );
}


/* kernel/irq/devres.c */
int devm_request_threaded_ irq ( struct device * dev , unsigned int irq ,
                              irq_handler_t handler , irq_handler_t thread_fn ,
                              unsigned long irqflags , const char * devname ,
                              void * dev_id )
{
        struct irq_devres * dr ;
        int rc ;

        dr = devres_ alloc ( devm_irq_release , sizeof ( struct irq_devres ),
                          GFP_KERNEL );
        if (! dr )
                return - ENOMEM ;

         / * Call we often see in the driver interrupt registration interface , register interrupt . The steps and device
resource management has nothing to do . * /
```

```
        rc = request_threaded_ irq ( irq , handler , thread_fn , irqflags , devname ,
                                    dev_id );
        if ( rc ) {
                devres_free ( dr );
                return rc ;
        }

        dr -> irq = irq ;
        dr -> dev_id = dev_id ;
        devres_ add ( dev , dr );

        return 0 ;
}
EXPORT_SYMBOL ( devm_request_threaded_irq );

void devm_free_ irq ( struct device * dev , unsigned int irq , void * dev_id )
{
        struct irq_devres match_data = { irq , dev_id };

        WARN_ ON ( devres_destroy ( dev , devm_irq_release , devm_irq_match ,
                                & match_data ));
        free_ irq ( irq , dev_id );
}
EXPORT_SYMBOL ( devm_free_irq );
```

devres_alloc

The first step for the framework driver to use device resource management is :

Described in their definition of a resource in the structure (e.g. IRQ in struct irq_devres), it is clear , that it will point to the zero-length array Inspiration address a structure :

```
/*
* Device resource management aware IRQ request/free implementation.
*/
struct irq_devres {
        unsigned int irq ;
        void * dev_id ;
};
```

Define the release function that releases your own resource . Note that this release is not used to release memory, but is related to the resource itself . For example, IRQ resources need to use request_irq when applying, and free_irq when releasing

```
static void devm_irq_ release( struct device *dev, void *res)
{
        struct irq_devres *this = res;
        free_irq(this->irq, this->dev_id);
}
```

The release function will be devres calling module , attention devres pass over the void * pointer (i.e. the length of the array at the address zero), each framework driver directly void * converted to describe their resource is a pointer to the structure . For devres In terms of it , it is impossible to know the specific form of the resource , so it can only use void*

Then callback function, resource structure corresponding to the size parameter , calls devres_alloc interfaces , to resource allocation space . Defined as the interface

```
void * devres_ alloc ( dr_release_t release , size_t size , gfp_t gfp )
{
        struct devres * dr ;

        dr = alloc_ dr ( release , size , gfp );
        if ( unlikely (! dr ))
                return NULL ;
        return dr -> data ;
}
```

Call alloc_dr , allocate a variable of struct devres type , and return the data pointer (data is a zero-length array) . The definition of alloc_dr is as follows :

```
static __always_inline struct devres * alloc_ dr ( dr_release_t release ,
                                                size_t size , gfp_t gfp )
{
        size_t tot_size = sizeof( struct devres ) + size ;
        struct devres * dr ;

        dr = kmalloc_track_ caller ( tot_size , gfp );
        if ( unlikely (! dr ))
                return NULL ;

        memset ( dr , 0 , tot_size );
        INIT_LIST_HEAD (& dr -> node . Entry );
        dr -> node . release = release ;
        return dr ;
}
```

A first look on it , the resource size before , plus a struct devres the size , that is, total space allocated . Removing struct devres in , that resources (by the data access pointer) . After initialization struct devres variable node

devres_add

alloc After successful , performed at initialization , and the pointer to the device (dev) and resource pointer (DR) parameter , calls devres_add, add resources to the resource list head (device devres_head) in , the interface is defined as follows :

```
void devres_ add ( struct device * dev , void * res )
{
        struct devres * dr = container_ of ( res , struct devres , data );
        unsigned long flags ;
```

```
        spin_lock_irqsave (& dev -> devres_lock , flags );
        add_ dr ( dev , & dr -> node );
        spin_unlock_irqrestore (& dev -> devres_lock , flags );
}

static void add_ dr ( struct device * dev , struct devres_node * node )
{
        devres_ log ( dev , node , "ADD" );
        BUG_ON (! list _empty (& node -> entry ));
        list_add_tail (& node -> entry , & dev -> devres_head );
}
```

devres_free

The space occupied by resources can be released through the devres_free interface , here is the memory:

```
void devres_ free ( void * res )
{
    if ( res ) {
        struct devres * dr = container_ of ( res , struct devres , data );

        BUG_ON (! list _empty (& dr -> node . Entry ));
        kfree ( dr );
    }
}
```

devres_remove

Remove a resource from the linked list in devres_head .

```
void * devres_ remove( struct device *dev, dr_release_t release,
     dr_match_t match, void *match_data)
{
struct devres *dr;
unsigned long flags;

spin_lock_irqsave(&dev->devres_lock, flags);
dr = find_ dr( dev, release, match, match_data);
if (dr) {
list_del_init (&dr-> node.entry );
devres_ log( dev, &dr->node, "REM");
}
spin_unlock_irqrestore(&dev->devres_lock, flags);

if (dr)
return dr->data;
```

```
return NULL;
}
```

devres_destroy

devm_free_irq interface , call the devres_destroy .
devres_destroy mainly does 2 actions

- calls devres_remove will devres from devres_head removed in
- Call devres_free to release memory resources

After devm_free_irq calls devres_destroy to process related matters , it will call free_irq to release the previously applied IRQ resources , and devm_free_irq is generally called actively by the driver .

API for driver model

Remember when we described in the introduction device resource management is designed to do ? That will solve the headache in the device initialization goto problems .
For example , the driver's probe function can request_irq to apply a IRQ resource , when an error occurs during the initialization of the subsequent , necessary to use free_irq releases the resources .
If you use it when applying for IRQ resourcesdevm_request_irq , in the event of subsequent initialization errors , you can manually use devm_free _irq to release resources (this method also uses goto, which is the same as the above method); you can also ignore everything and directly return - XXX. The core code of the driver model is received After this error , **resources** will be **automatically released** .

devres_release_all

devres_release_all is used to automatically release the resources in . to achieve the following :

```
int devres_release_ all ( struct device * dev )
{
        unsigned long flags ;

        /* Looks like an uninitialized device structure */
        if ( WARN_ON ( dev -> devres_ head . next == NULL ))
                return - ENODEV ;
        spin_lock_irqsave (& dev -> devres_lock , flags );
         return release_ nodes ( dev , dev -> devres_head . next , & dev -> devres_head
,
                               flags );
}
```

With the device pointer as a parameter , call release_nodes directly :

```
static int release_ nodes ( struct device * dev , struct list_head * first ,
                           struct list_head * end , unsigned long flags )
        __releases (& dev -> devres_lock )
{
        LIST_HEAD ( todo );
```

```
        int cnt ;
        struct devres * dr , * tmp ;

        cnt = remove_ nodes ( dev , first , end , & todo );

        spin_unlock_irqrestore (& dev -> devres_lock , flags );

        /* Release. Note that both devres and devres_group are
         * handled as devres in the following loop. This is safe.
         */
        list_for_each_entry_safe_ reverse ( dr , tmp , & todo , node . entry ) {
                devres_ log ( dev , & dr -> node , "REL" );
                dr -> node . release ( dev , dr -> data );
                kfree ( dr );
        }

        return cnt ;
}
```

remove_nodes is to remove this resource from the head of the device list .

node.release is to call the release function of a certain resource . For IRQ , it is to call devm_irq_release . This release function will call free_irq.

kfree (dr) is to release memroy resources .

This three-step action , in fact, with devm_free_irq the same effect .

When to call devres_release_all

We know devres_release_all is the device model core code to automatically release the device occupied by the resources of . That core code under what circumstances would call the function it ? There are two occasions :

- ♦♒♏■ ♦♒♏ probe fails

  The calling process is (the code will not be posted in detail) :

  __driver_attach / __ device_attach-> driver_probe_device - > really_probe , really_probe call driver or bus of probe interfaces , if (return fails nonzero , examples with reference to the beginning of this article) , is invoked devres_release_all

- ♦♒♏■ d river dettach (that is , when the driver removes)

  driver_detach / bus_remove_device–>__device_release_driver–>devres_release_all

---