S I X U E N E T

To Learn Without Thinking Is To Be Useless, To Think Without Learning Is To Lose
(http://www.mysixue.com/)

ALSA & ASOC

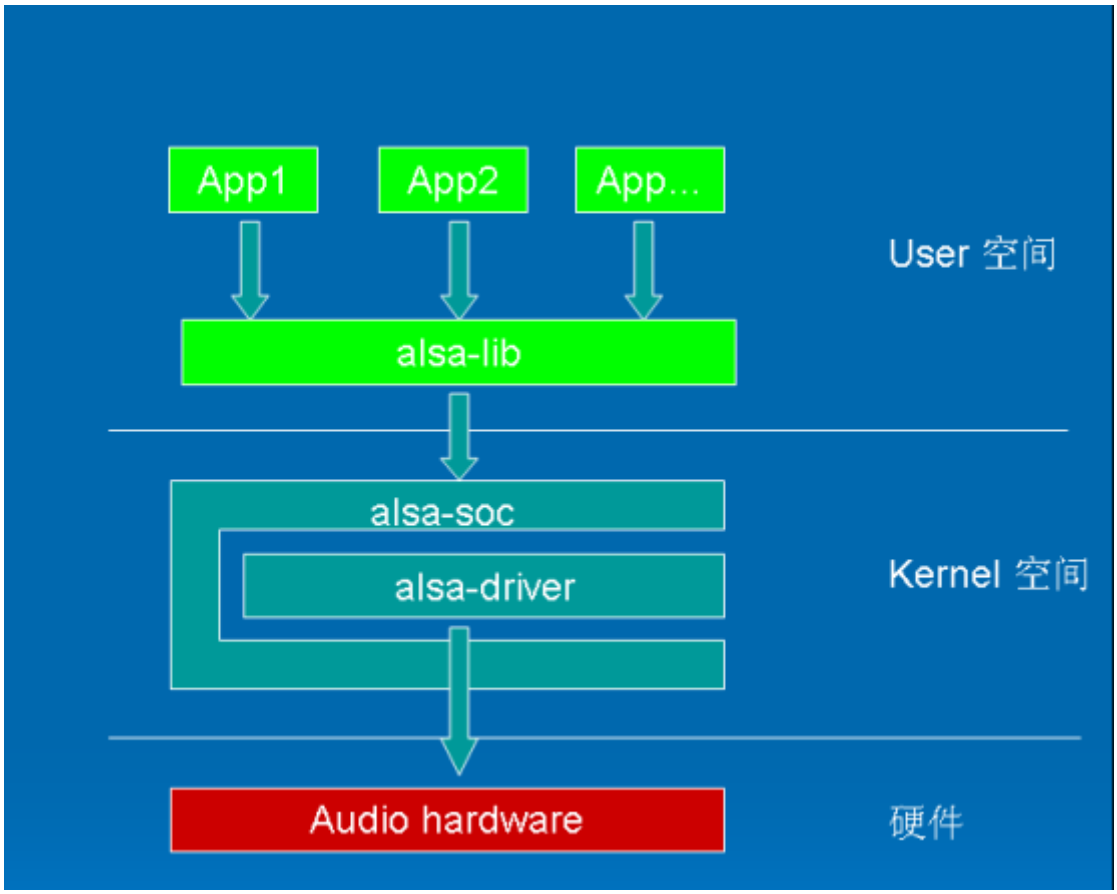
April 24, 2019 (Http://Www.Mysixue.Com/?P=134) JL
(Http://Www.Mysixue.Com/?Author=1)

1 ALSA

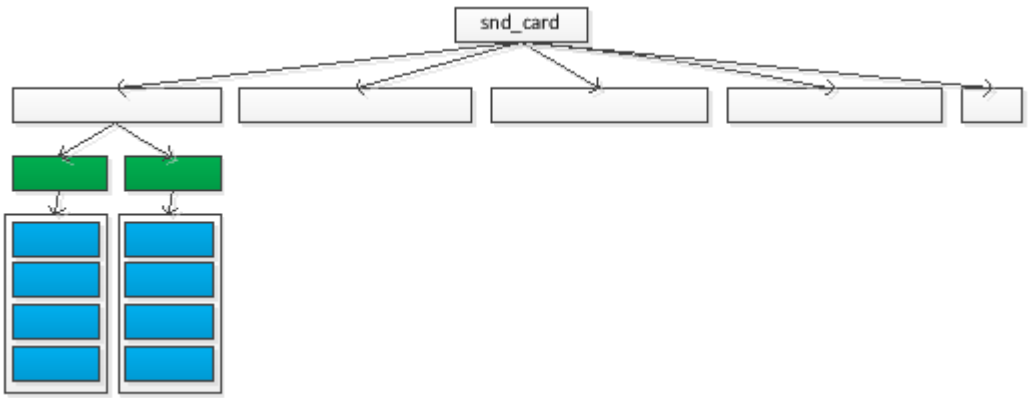
1.1 Brief description of architecture

ALSA is the abbreviation of Advanced Linux Sound Architecture , official website : http://www.alsa-project.org (http://www.alsa-project.org) . (http://www.alsa-project.org)

At the kernel device driver layer , ALSA provides alsa-driver . At the same time, at the application layer , ALSA provides us with alsa-lib . As long as the application calls the API provided by alsa-lib , it can complete the control of the underlying audio hardware .



From the point of view of the data structure organization level , ALSA with Card represents a pond , Card below may have one or more logical devices Device; each Device below may be subdivided other devices (e.g., the PCM Device) , may no longer continue to fine Points (for example, Control Device) .



From a hardware perspective , simply , it can be considered a C ARD corresponding to a board , a D evice one channel on the board corresponds .

Suppose, for example, a CPU has . 3 th I2S, and then add the one WM8920 audio chip , audio chip and three-way interfaces AIFl , AIF2 , AIF3. And that such a circuit board have a C ARD, the following three D evice (I2S1 - AIFl, I2S2 – AIF2, I2S3 – AIF3).

Of course , there can also be multiple Cards on a board. In this case, one type of interface is generally abstracted as a Card. For example, all I2S interfaces are abstracted as a Card, and all PCM interfaces are abstracted as a Card.

A Device may also be a pure software-level abstraction , and does not correspond to specific hardware . Therefore, we usually call Device a logical device .

1.2 Introduction to device nodes

/dev/snd

```
crw-rw—-+ 1 root audio 116, 8 2011-02-23 21:38 controlC0
crw-rw—-+ 1 root audio 116, 4 2011-02-23 21:38 midiC0D0
crw-rw—-+ 1 root audio 116, 7 2011-02-23 21:39 pcmC0D0c
crw-rw—-+ 1 root audio 116, 6 2011-02-23 21:56 pcmC0D0p
crw-rw—-+ 1 root audio 116, 5 2011-02-23 21:38 pcmC0D1p
crw-rw—-+ 1 root audio 116, 3 2011-02-23 21:38 seq
crw-rw—-+ 1 root audio 116, 2 2011-02-23 21:38 timer
```

- ✓ controlC0 : used for sound card control, such as channel selection, mixing, microphone control, etc.
- ✓ midiC0D0 : used to play midi audio
- ✓ pcmC0D0c : pcm device for recording
- ✓ pcmC0D0p : pcm device for playback
- ✓ seq : sequencer
- ✓ timer: timer

Wherein , c0d0 represents the sound 0 devices 0 , pcmC0D0c last c representative of Capture , pcmC0D0p last p Representative Playback , these are alsa-driver naming rules .

/proc/asound

```
dr-xr-xr-x 6 root root 0 Oct 8 00:04 card0
-r-r-r- 1 root root 0 Oct 8 00:04 cards
-r-r-r- 1 root root 0 Oct 8 00:04 devices
-r-r-r- 1 root root 0 Oct 8 00:04 modules
dr-xr-xr-x 2 root root 0 Oct 8 00:04 oss
-r-r-r- 1 root root 0 Oct 8 00:04 pcm
dr-xr-xr-x 2 root root 0 Oct 8 00:04 seq
-r-r-r- 1 root root 0 Oct 8 00:04 timers
-r-r-r- 1 root root 0 Oct 8 00:04 version
```

- ✓ cards: can show how many sound cards exist in the system
- ✓ card0: represents a sound card
- ✓ devices: can display how many logical devices exist in the system

/proc/asound/card0

The node provides some info of the card .

```
-r-r-r- 1 root root 0 Oct 7 19:57 audiopci
dr-xr-xr-x 2 root root 0 Oct 7 19:57 codec97#0
-r-r-r- 1 root root 0 Oct 7 19:57 id
-r-r-r- 1 root root 0 Oct 7 19:57 midi0
dr-xr-xr-x 3 root root 0 Oct 7 19:57 pcm0c
dr-xr-xr-x 3 root root 0 Oct 7 19:57 pcm0p
dr-xr-xr-x 3 root root 0 Oct 7 19:57 pcm1p
```

/sys/class/sound/

```
lrwxrwxrwx 1 root root 0 Oct 7 20:27 card0 ->
../../devices/pci0000:00/0000:00:11.0/0000:02:02.0/sound/card0
...
```

1.3 Card creation and registration

1.3.1 What is a card

Struct snd_card can be said to be the topmost structure of the entire ALSA audio driver . The software logical structure of the entire sound card starts from this structure . Almost all sound-related logical devices are under the management of snd_card , the first action of the sound card driver. Usually it is to create a snd_card structure .

1.3.2 Data Structure

struct snd_card

A struct snd_card is used to describe a sound card .

Header file : include/sound/ cor e .h (https://elixir.bootlin.com/linux/v4.18.5/source/include/sound/core.h#L94)

struct snd_card	Comment
int number	Sound card number , the kernel supports the creation of multiple sound cards
...	
struct list_head devices	Record the linked list of all logical devices under the sound card
struct list_head controls	Record the linked list of all control units under the sound card
s truct snd_info_entry *proc_root	A sound card can have multiple snd_info_entry, and each entry represe nts a node under /proc/asound/cardx/ . These entries are organized in a tree structure , and proc_root is the ro ot of this number .
...	
void *private_data;	The private data of the sound card, the size of the data can be specified by parameters when the sound card is created
...	

1.3.3 API analysis

snd_card_new

snd_card_new (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/init.c#L201) is defined in sound/core/init.c .

There are explanations of parameters and return values in the code :

```
/**
 *  snd_card_new - create and initialize a soundcard structure
 *  @parent: the parent device object
 *  @idx: card index (address) [0 ... (SNDRV_CARDS-1)]
 *  @xid: card identification (ASCII string)
 *  @module: top level module for locking
 *  @extra_size: allocate this extra size after the main soundcard structure
 *  @card_ret: the pointer to store the created card instance
 *
 *  Creates and initializes a soundcard structure.
 *
 *  The function allocates snd_card instance via kzalloc with the given
 *  space for the driver to use freely.  The allocated struct is stored
 *  in the given card_ret pointer.
 *
 *  Return: Zero if successful or a negative error code.
 */
int snd_card_new(struct device *parent, int idx, const char *xid,
                 struct module *module, int extra_size,
                 struct snd_card **card_ret)
```

The whole thing the code does is assign a snd_card structure , and initialize the related fields of the structure :

- First call kzalloc to allocate a piece of storage space . If extra_size> 0, it means that an extra space needs to be created for card->private_data.

```
if (extra_size < 0)
        extra_size = 0;
card = kzalloc(sizeof(*card) + extra_size, GFP_KERNEL);
if (!card)
        return -ENOMEM;
if (extra_size > 0)
        card->private_data = (char *)card + sizeof(struct snd_card);
```

- Copy the ID string of the sound card .

```
if (xid)
        strlcpy(card->id, xid, sizeof(card->id));
```

➢✋➚ ♦≈♏ incoming sound card number is -1 , an index number will be assigned automatically .

```
if (idx < 0) /* first check the matching module-name slot */
        idx = get_slot_from_bitmask(idx, module_slot_match, module);
if (idx < 0) /* if not matched, assign an empty slot */
        idx = get_slot_from_bitmask(idx, check_empty_slot, module);
if (idx < 0)
        err = -ENODEV;
else if (idx < snd_ecards_limit) {
        if (test_bit(idx, snd_cards_lock))
                err = -EBUSY;   /* invalid */
} else if (idx >= SNDRV_CARDS)
        err = -ENODEV;
if (err < 0) {
        mutex_unlock(&snd_card_mutex);
        dev_err(parent, "cannot find the slot for index %d (range 0-%i), error: %d\n",
                idx, snd_ecards_limit - 1, err);
        kfree(card);
        return err;
}
```

- Initialize the necessary fields in the snd_card structure .

```
	card->dev = parent;
	card->number = idx;
	card->module = module;
	INIT_LIST_HEAD(&card->devices);
	init_rwsem(&card->controls_rwsem);
	rwlock_init(&card->ctl_files_rwlock);
	INIT_LIST_HEAD(&card->controls);
	INIT_LIST_HEAD(&card->ctl_files);
	spin_lock_init(&card->files_lock);
	INIT_LIST_HEAD(&card->files_list);
#ifdef CONFIG_PM
	init_waitqueue_head(&card->power_sleep);
#endif
	init_waitqueue_head(&card->remove_sleep);

	device_initialize(&card->card_dev);
	card->card_dev.parent = parent;
	card->card_dev.class = sound_class;
	card->card_dev.release = release_card_device;
	card->card_dev.groups = card->dev_groups;
	card->dev_groups[0] = &card_dev_attr_group;
	err = kobject_set_name(&card->card_dev.kobj, "card%d", idx);
	if (err < 0)
		goto __error;

	snprintf(card->irq_descr, sizeof(card->irq_descr), "%s:%s",
		 dev_driver_string(card->dev), dev_name(&card->card_dev));
```

Note that there is a sentence card->card_dev.class = sound_class , which means that when card_dev is added to the sysfs framework , a symbolic link to the device will be created under /sys/class/sound / .

The sound_class (https://elixir.bootlin.com/linux/v4.18.5/source/sound/sound_core.c#L39) creation code also initializes sound_class->devnode to "snd/" , which means that the corresponding character device node will exist in the /dev/snd/ directory .

- Establish a logical device : Control .

```
	/* the control interface cannot be accessed from the user space until */
	/* snd_cards_bitmask and snd_cards are set with snd_card_register */
	err = snd_ctl_create(card);
	if (err < 0) {
		dev_err(parent, "unable to register control minors\n");
		goto __error;
	}
```

- establish proc file info node : usually the / proc / asound / card0 . Note , this only creates card0 this directory , files in the directory is not created here , create a file, refer to " info nodes" .

```
	err = snd_info_card_create(card);
	if (err < 0) {
		dev_err(parent, "unable to create card info\n");
		goto __error_ctl;
	}
```

- Finally , return the prepared structure to the caller .

```
	*card_ret = card;
	return 0;
```

snd_card_register

snd_card_register (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/init.c#L762) is defined in sound/core/init.c .

There are explanations of parameters and return values in the code :

```
/**
 *  snd_card_register - register the soundcard
 *  @card: soundcard structure
 *
 *  This function registers all the devices assigned to the soundcard.
 *  Until calling this, the ALSA control interface is blocked from the
 *  external accesses.  Thus, you should call this function at the end
 *  of the initialization of the card.
 *
 *  Return: Zero otherwise a negative error code if the registration failed.
 */
int snd_card_register(struct snd_card *card)
```

Note the code comments , the main purpose of this function is a registered card below the mount all the device; another only after the function returns successfully , the user space to pass the control interface to access the underlying .

- First call device_add to add yourself to the sysfs framework :

```
if (!card->registered) {
        err = device_add(&card->card_dev);
        if (err < 0)
                return err;
        card->registered = true;
}
```

- Call snd_device_register_all() to register all logical devices connected to the sound card .

  snd_device_register_all () is actually through snd_card the devices list , through all snd_device , and calls snd_device of ops-> dev_register () to achieve their registered device .

```
if ((err = snd_device_register_all(card)) < 0)
        return err;
```

- in / proc / asound / card0 add some fields , the exposure information related to the user space . Code is not posted .

snd_card_disconnect

snd_card_disconnect (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/init.c#L395) is defined in sound/core/init.c .

```
/**
 *  snd_card_disconnect - disconnect all APIs from the file-operations (user space)
 *  @card: soundcard structure
 *
 *  Disconnects all APIs from the file-operations (user space).
 *
 *  Return: Zero, otherwise a negative error code.
 *
 *  Note: The current implementation replaces all active file->f_op with special
 *        dummy file operations (they do nothing except release).
 */
int snd_card_disconnect(struct snd_card *card)
```

1.4 Device creation and registration

1.4.1 What is a logical device (device)

Logical device belongs Card, is typically used to achieve a function , a logical device may correspond to a general or more device nodes of the user space (there are only certain device node does not create the logical devices used by the kernel) .

1.4.2 Data Structure

struct snd_device

A struct snd_device is used to describe a logical device .

Header file : include/sound/ core.h (https://elixir.bootlin.com/linux/v4.18.5/source/include/sound/core.h#L81)

struct snd_ device	Comment
struct list_head list	Used to mount yourself under the snd_card-> devices list .
struct snd_card *card	card which holds this device

enum snd_device_state (https://elixir.bootlin.com/linux/v4.18.5/source/include/sound/core.h#L67) state	The status of the device . There are three types: BUILD , REGISTERED , and DISCONNECTED .
enum snd_device_type (https://elixir.bootlin.com/linux/v4.18.5/source/include/sound/core.h#L52) type	Type device . The system defines various device types , e.g. SNDRV_DEV_PCM , SNDRV_DEV_CONTROL the like .
void *device_data	Private data of the device .
struct snd_device_ops *ops	ops, detailed below .

struct snd_device_ops

Each logical device has a corresponding snd_device_ops, we see new growth when a logical device , need to prepare well for this device the data structure .

Header file : include/sound/ core.h (https://elixir.bootlin.com/linux/v4.18.5/source/include/sound/core.h#L75)

struct snd_ device_ops	Comment
int (*dev_free)(struct snd_device *dev)	When someone calls the API snd_device_free time , the core layer code callback here dev_free
int (*dev_register)(struct snd_device *dev)	When someone calls the API snd_device_register time , the core layer code callback here dev_register
int (*dev_disconnect)(struct snd_device *dev)	When someone calls the API snd_device_disconnect time , the core layer code callback here dev_disconnect

1.4.3 API analysis

snd_device_new

snd_device_new (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/device.c#L44) the source / sound / core / device.c definition .

```
/**
 * snd_device_new - create an ALSA device component
 * @card: the card instance
 * @type: the device type, SNDRV_DEV_XXX
 * @device_data: the data pointer of this device
 * @ops: the operator table
 *
 * Creates a new device component for the given data pointer.
 * The device will be assigned to the card and managed together
 * by the card.
 *
 * The data pointer plays a role as the identifier, too, so the
 * pointer address must be unique and unchanged.
 *
 * Return: Zero if successful, or a negative error code on failure.
 */
int snd_device_new(struct snd_card *card, enum snd_device_type type,
                   void *device_data, struct snd_device_ops *ops)
```

This API is used to create a logical device . The main content is to allocate a struct snd_device space , initialize the relevant fields , and add the logical device to the card->devices linked list .

Note that when adding new devices to the card->devices linked list , an orderly insertion method is used : the smaller type is first, and the larger type is last . For details, you can read the code .

snd_device_register

snd_device_register (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/device.c#L186) the source / sound / core / device.c definition .

```
/**
 * snd_device_register - register the device
 * @card: the card instance
 * @device_data: the data pointer to register
 *
 * Registers the device which was already created via
 * snd_device_new().  Usually this is called from snd_card_register(),
 * but it can be called later if any new devices are created after
 * invocation of snd_card_register().
 *
 * Return: Zero if successful, or a negative error code on failure or if the
 * device not found.
 */
int snd_device_register(struct snd_card *card, void *device_data)
```

Notes which says more clear . Generally during registration card (snd_card_register) will automatically call here the API; but can also be card after registration is completed , the manual calls this API to register a new logical device .

The implementation of this API is very simple : first call back snd_device_ ops->dev_register , and then mark the state of the core layer as SNDRV_DEV_REGISTERED .

snd_device_register_all

snd_device_register_all (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/device.c#L204) the source / sound / core / device.c definition .

```
/*
 * register all the devices on the card.
 * called from init.c
 */
int snd_device_register_all(struct snd_card *card)
```

It is equivalent to the encapsulation of snd_device_register : for each logical device under the card , call __snd_device_register to register this logical device .

snd_device_disconnect

snd_device_disconnec (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/device.c#L121) t is defined in source/sound/core/ device.c (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/device.c#L121) .

```
/**
 * snd_device_disconnect - disconnect the device
 * @card: the card instance
 * @device_data: the data pointer to disconnect
 *
 * Turns the device into the disconnection state, invoking
 * dev_disconnect callback, if the device was already registered.
 *
 * Usually called from snd_card_disconnect().
 *
 * Return: Zero if successful, or a negative error code on failure or if the
 * device not found.
 */
void snd_device_disconnect(struct snd_card *card, void *device_data)
```

This API is generally called when snd_card_disconnect is called .

The implementation is also very simple , call back snd_device_ ops->dev_ disconnect, and then mark the state of the core layer as SNDRV_DEV_DISCONNECTED .

snd_device_disconnect_all

snd_device_disconnect_all (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/device.c#L223) the source / sound / core / device.c definition .
For each logical device , call snd_device_disconnect .

snd_device_free

snd_device_free (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/device.c#L145) the source / sound / core / device.c definition .

```
/**
 * snd_device_free - release the device from the card
 * @card: the card instance
 * @device_data: the data pointer to release
 *
 * Removes the device from the list on the card and invokes the
 * callbacks, dev_disconnect and dev_free, corresponding to the state.
 * Then release the device.
 */
void snd_device_free(struct snd_card *card, void *device_data)
```

It is relatively simple , first of all himself from card-> devices to remove the list , then call __snd_device_disconnect , then the callback snd_device_ OPS-> dev_ as Free, the last call kfree release snd_device structure .

snd_device_free_all

snd_device_free_all (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/device.c#L237) the source / sound / core / device.c definition .

For each logical device , call snd_device_ free.

1.4.4 Creation of character device driver

The logical device and user space interact through character device drivers .

A logical device can create one or more character device nodes . The timing of node creation is as follows :

When the sound card is registered (snd_card_register) , snd_device_register will be called for each logical device under it. The latter will call back the callback function of the logical device (snd_device_ ops->dev_register) .

Logic devices to achieve dev_register time , usually call snd_register_device , which will pass device_add create a device node . So call several times snd_register_device , it will exist a few device nodes .

The setup of character devices in the ALSA subsystem is similar to the misc subsystem .

All ALSA major number of characters equipment is CONFIG_SND_MAJOR (https://elixir.bootlin.com/linux/v4.18.5/source/include/sound/core.h#L40) (116), ALSA system initialization , will alsa_sound_init (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/sound.c#L398) call register_chrdev , define a unified processing functions such characters equipment snd_fops .

snd_fops (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/sound.c#L179) only achieved snd_ open function , when the user opens the space of any one of ALSA when the device node , will enter into the open function

In snd_ Open (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/sound.c#L150) in , based on the minor number (each device has its own logical device number of times) , selecting a corresponding logical device ops function , and then replace File-> the f_op , any user space operation is thereafter directly The ops function of the logical device is interactive .

There is a global array snd_minors (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/sound.c#L57) [] in the ALSA system. The subscript of the array is the minor device number . Each element corresponds to a logical device . snd_minor [i]-> f_ops stores the ops function of the logical device itself .

With it , snd_open is relatively easy to achieve , simply by minor number as a subscript , from snd_minors [] finds a corresponding element , then the element f_ops Alternatively file-> f_op to .

That snd_minors [] The array is to be filled when it ? The answer is in snd_register_device inside . Here we look at the details of the function .

snd_register_device

snd_register_device (https://elixir.bootlin.com/linux/v4.18.5/source/sound/core/sound.c#L258) main role is to fill snd_minors [] array , and create a character device node .

```
258  int snd_register_device(int type, struct snd_card *card, int dev,
259                          const struct file_operations *f_ops,
260                          void *private_data, struct device *device)
261  {
262          int minor;
263          int err = 0;
264          struct snd_minor *preg;
265
266          if (snd_BUG_ON(!device))
267                  return -EINVAL;
268
269          preg = kmalloc(sizeof *preg, GFP_KERNEL);
270          if (preg == NULL)
271                  return -ENOMEM;
272          preg->type = type;
273          preg->card = card ? card->number : -1;
274          preg->device = dev;
275          preg->f_ops = f_ops;
276          preg->private_data = private_data;
277          preg->card_ptr = card;
278          mutex_lock(&sound_mutex);
279          minor = snd_find_free_minor(type, card, dev);
280          if (minor < 0) {
281                  err = minor;
282                  goto error;
283          }
284
285          preg->dev = device;
286          device->devt = MKDEV(major, minor);
287          err = device_add(device);
288          if (err < 0)
289                  goto error;
290
291          snd_minors[minor] = preg;
292   error:
293          mutex_unlock(&sound_mutex);
294          if (err < 0)
295                  kfree(preg);
296          return err;
297  }
298  EXPORT_SYMBOL(snd_register_device);
```

- ✓ parameters f_ops: logic devices to achieve their own ops set of functions . User space is the final here ops interaction .
- ✓ Line 287 : Create a character device node , the node name is determined by the parameter device

1.5 Logical device middle layer

In the current system , there are many logical device intermediate layers , which are equivalent to encapsulating the steps in `` Device Creation and Registration" , and then provide a more concise API to create the corresponding logical device .

These intermediate layers include :

middle layer	API
C ontrol equipment	snd_ctl_create (https://elixir.bootlin.com/linux/latest/source/sound/core/control.c#L1893) : call snd_device_new in C ARD create a lower SNDRV_DEV_CONTROL type of logic device .

PCM equipment	snd_pcm_new (https://elixir.bootlin.com/linux/latest/source/sound/core/pcm.c#L835) : on the class
RAWMIDI device	snd_rawmidi_new (https://elixir.bootlin.com/linux/latest/source/sound/core/rawmidi.c#L1535) : Class on
TIMER device	snd_timer_new (https://elixir.bootlin.com/linux/latest/source/sound/core/timer.c#L873)
JACK equipment	snd_jack_new (https://elixir.bootlin.com/linux/latest/source/sound/core/jack.c#L212)
I nfo equipment	snd_card_proc_new (https://elixir.bootlin.com/linux/latest/source/include/sound/info.h#L148) -> snd_info_create_card_entry (https://elixir.bootlin.com/linux/latest/source/sound/core/info.c#L761) , the latter will eventually create a leaf node under the card->proc_root tree .

In addition to the I NFO devices outside , several other intermediate layers are of snd_device_new conducted a package . When invoked intermediate layer provides API time , eventually will C ARD add a new logical device under .

In the card registration phase , scans it below each logical device , then call snd_ device_register to register the logical device . When the logical device registration is complete , the core layer will callback snd_device_ops .dev_register function , if dev_register which call snd_register_device, the A character device node will appear in the user space .

Below we focus on the analysis of the common types of Control, PCM, and info .

1.5.1 info logical device

info device is not like any other logic devices , it should be called more info node . It does not create a character device node , but will be in / proc / create some files , in order to get some information about the user space through these files .

A info node with a snd_info_entry expressed , a entry corresponding to the / proc / asound / cardx / a file or a directory .

We can use snd_card_proc_new (https://elixir.bootlin.com/linux/latest/source/include/sound/info.h#L148) > - snd_info_create_card_entry (https://elixir.bootlin.com/linux/latest/source/sound/core/info.c#L761) create these entry, all entry will be mounted in a tree structure card-> proc_root down , pay attention to these two API just creates a entry, but users do not see this space entry, need Call snd_info_register (https://elixir.bootlin.com/linux/latest/source/sound/core/info.c#L824) , and then the user space will see it .

In the card registration phase , will traverse card-> proc_root each leaf node under , and then call snd_info_register the leaf nodes displayed in / proc / asound / cardx / under . Its call flow is :
snd_card_register -> init_info_for_card (https://elixir.bootlin.com/linux/latest/source/sound/core/init.c#L110) -> snd_info_card_register (https://elixir.bootlin.com/linux/latest/source/sound/core/info.c#L552) -> snd_info_register_recursive -> snd_info_register (https://elixir.bootlin.com/linux/latest/source/sound/core/info.c#L824) .

1.5.2 Control logical device

In the ALSA system , we can configure the various parameters of the audio through the amixer tool . Amixer operates the Control device here .

The common usage of amixer is as follows:

- amixer contents : List all available controls
- amixer cget numid=5, iface=MIXER,name='PCM Volume': get the value of a control
- amixer cset numid=5, iface=MIXER,name='PCM Volume' 25: set the value of a certain control

Note here numid, iface, name these fields , later will be analyzed in detail .

The Control device is presented to the user space as a character device , and its device node is /dev/snd/ controlC x. This section will briefly describe the creation process of this device node .

K ernel build a C ontrol device of the intermediate layer , the corresponding code is Sound / Core / CONTROL.C (https://elixir.bootlin.com/linux/v4.19.11/source/sound/core/control.c) , the intermediate layer on the counter ALSA register the core layer , will register itself as a logical device , and finally creating a character device node , it is responsible for user space interaction (snd_ctl_f_ops (https://elixir.bootlin.com/linux/v4.19.11/source/sound/core/control.c#L1829)); next is provided API to the respective specific codec driver , for added specific Control; internal data structures are created inside the intermediate layer to manage all the controls.

Next, let's take a look at the data structures inside the C ontrol middle layer and the APIs it provides to the lower layer , and then look at how a specific driver can use these APIs to add a control, and finally look at the process of creating a C ontrol character device .

1.5.2.1 Data structure

Before introducing the control- related data structure , let's recall the familiar " sysfs A ttribute " attribute file , because the two have strong similarities .

When the driver wants to create a property file, it will prepare the name , permissions, read and write functions , and then register with the'property file middle layer' . Then we can see the property file /sysfs/xxx/ name , and can o/ r/w/c this file .

control similarly , when a particular codec driver would like to add a control time , it should follow the ' C ontrol intermediate layer is required' ready name , authority, check read and write functions, etc. , and then register with the intermediate layer . After registration after , We can see this control with amixer contents , and modify it with amixer cget/cset .

In this process , the middle layer has to assume the following roles :

a) Manager role : There will be multiple registrations to the middle layer , because all registered elements must be managed in a linked list or other ways in the middle layer .

b) Positioning function : When the user space accesses an element , the middle layer must be able to locate the corresponding element from the linked list . Therefore, the middle layer will give each element a token to distinguish different elements . The user space needs to access a certain element . when

elements , need to specify this element of the token. for the properties file for , token is the absolute path to the file , but to control it , this token is more complicated .

c) a bridge : the user space of an element of the read and write operations , is converted into the element custom read and write operations . read operation means that a storage element to have value, the reader is to read the value value or modified Its value . In this process , the middle layer does not touch the value , so in general , the middle layer does not create a v alue storage space for the element , this storage space is created and maintained in the registrant . the intermediate layer will create a data structure that represents the elements , this data structure will function as a parameter passed back to the reading and writing of the registrant , the registrant by Contain ER _of , you can find the corresponding element of storage space , and then modify the The contents of the storage space .

d) the storage space on the element , in the attribute file , is relatively simple , generally int or string type ; but ALSA to be complicated , and therefore ' C ontrol intermediate layer' defines the data structures used to describe the storage space , after The text is detailed .

Understood that these points of action of the intermediate layer , and then look at C ontrol intermediate layer will be made easier . In ' C ontrol intermediate layer' interior , with snd_kcontrol representative of a control element ; with snd_ctl_elem_id as control of the token, in order to distinguish the different control; with snd_ctl_elem_info / snd_ctl_elem_value to describe the value storage space .

struct snd_kcontrol

struct snd_kcontrol (https://elixir.bootlin.com/linux/v4.19.11/source/include/sound/control.h#L69) : represents a control element

```
struct snd_kcontrol {
        struct list_head list;          /* list of controls */
        struct snd_ctl_elem_id id;
        unsigned int count;             /* count of same elements */
        snd_kcontrol_info_t *info;
        snd_kcontrol_get_t *get;
        snd_kcontrol_put_t *put;
        union {
                snd_kcontrol_tlv_rw_t *c;
                const unsigned int *p;
        } tlv;
        unsigned long private_value;
        void *private_data;
        void (*private_free)(struct snd_kcontrol *kcontrol);
        struct snd_kcontrol_volatile vd[0];     /* volatile data */
};
```

- list: All control elements will be mounted under the card-> controls header .
- id: the token of the element
- count: An element can have only one value or multiple value[] , where count represents the number . In the control system , the technical term for value is element, and one value corresponds to one element. Therefore, the comments after the code are written It is /* count of same elements */ .
- i nfo/get/put: search / read / write function .
- tlv: I do n't know the details yet .
- private_value: A long type of data , which can be accessed through the callback functions info , get , and put . The underlying codec driver customizes its role (for example, it can be split into multiple bit fields , or a pointer , pointing to a data structure) , ' Control intermediate layer' do not touch it .

  In general the underlying codec driver address field stores will register itself , so that when the user space control time , the bottom thereof can put the content of the register function , so as to achieve control hardware .

- private_date / private_free: The underlying private data , the Control middle layer will not touch it . Perhaps the kernel developers found that unsigned long private_value is not used as a pointer , so such a void * pointer was specially added .
- VD [0]: zero-length array , its final size and count related . There may be multiple Elements, each element read and write privileges may be different , so each VD [I] element describes an element privileges .

Note that snd_kcontrol only describes the relevant characteristics of the control element , and does not contain storage space for data .

struct snd_ctl_elem_id

struct snd_ctl_elem_id (https://elixir.bootlin.com/linux/v4.19.11/source/include/uapi/sound/asound.h#L884) : token, used to distinguish each kcontrol.

```
struct snd_ctl_elem_id {
        unsigned int numid;             /* numeric identifier, zero = invalid */
        snd_ctl_elem_iface_t iface;     /* interface identifier */
        unsigned int device;            /* device/client number */
        unsigned int subdevice;         /* subdevice (substream) number */
        unsigned char name[SNDRV_CTL_ELEM_ID_NAME_MAXLEN];              /* ASCII name of item */
        unsigned int index;             /* index of item */
};
```

The token rules in the Control system are a bit complicated :

- numid: In one card at , it is so kcontrols has a different numid . By amixer controls can see all kcontrols and each kcontrol corresponding numid .
  Because numid is unique under a card , if numid is specified in amixer cget/cset , the kernel layer can directly find the corresponding snd_kcontrol through it .
  numid a major role for spatial separation of different user elem , its value in a card within from 0 increments .
- If you do not specify a space numid , that go to the core layer judgment iface + Device + subdev ICE + name + index these fields , only when they are all matching to uniquely identify a snd_kcontrol .
  The kernel layer is responsible for finding the corresponding snd_kcontrol function through snd_ctl_elem_id is snd_ctl_find_id (https://elixir.bootlin.com/linux/v4.19.11/source/sound/core/control.c#L702) , you can look at the code , it is easier to understand the logic here . (https://elixir.bootlin.com/linux/v4.19.11/source/sound/core/control.c#L702)
- index: Because a kcontrol below may have multiple elem , index for kcontrol labeling different internal . Elem its value in a kcontrol the range of from 0 increments .

snd_ctl_elem_id is equivalent to a link , so that all structures containing the token can be indirectly related through it .

struct snd_ctl_elem_info

struct snd_ctl_elem_info (https://elixir.bootlin.com/linux/v4.19.11/source/include/uapi/sound/asound.h#L902) : used to describe the info information of an elem . When the user space queries the info information of an elem (for example, amixer contents) , the underlying driver needs to fill in the content of the structure and feed it back to the user space .

```
struct snd_ctl_elem_info {
        struct snd_ctl_elem_id id;      /* W: element ID */
        snd_ctl_elem_type_t type;       /* R: value type - SNDRV_CTL_ELEM_TYPE_* */
        unsigned int access;            /* R: value access (bitmask) - SNDRV_CTL_ELEM_ACCESS_* */
        unsigned int count;             /* count of values */
        __kernel_pid_t owner;           /* owner's PID of this control */
        union {
                struct {
                        long min;               /* R: minimum value */
                        long max;               /* R: maximum value */
                        long step;              /* R: step (0 variable) */
                } integer;
                struct {
                        long long min;          /* R: minimum value */
                        long long max;          /* R: maximum value */
                        long long step;         /* R: step (0 variable) */
                } integer64;
                struct {
                        unsigned int items;     /* R: number of items */
                        unsigned int item;      /* W: item number */
                        char name[64];          /* R: value name */
                        __u64 names_ptr;        /* W: names list (ELEM_ADD only) */
                        unsigned int names_length;
                } enumerated;
                unsigned char reserved[128];
        } value;
        union {
                unsigned short d[4];            /* dimensions */
                unsigned short *d_ptr;          /* indirect - obsoleted */
        } dimen;
        unsigned char reserved[64-4*sizeof(unsigned short)];
};
```

- id: When the user space wants to obtain the specific information of an elem , the user space will prepare the snd_ctl_elem_info data structure , and fill the elem 's token id into the structure , and then pass the structure to the kernel . ' C ontrol middle layer 'Will find the corresponding snd_kcontrol through the id , and then call back its info function . The underlying driver fills the content of the structure in the i nfo function , and then returns it to the user space .

  That is how you know that a user space elem the token id it ? ' C ontrol middle layer' implemented a list method snd_ctl_elem_list (https://elixir.bootlin.com/linux/v4.19.11/source/sound/core/control.c#L755) , when the method is called when the user space , ' C ontrol middle layer' will all elem the token id to Feedback to the user space in the form of a list .
- type: Different elems may have different types . The defined types are bool/int/ enumerate/int64 /byte. Refer to snd_ctl_elem_type_t for details (https://elixir.bootlin.com/linux/v4.19.11/source/include/uapi/sound/asound.h#L838) (https://elixir.bootlin.com/linux/v4.19.11/source/include/uapi/sound/asound.h#L838)
- access: the access authority of the elem , refer to SNDRV_CTL_ELEM_ACCESS_ (https://elixir.bootlin.com/linux/v4.19.11/source/include/uapi/sound/asound.h#L858) * for optional values . (https://elixir.bootlin.com/linux/v4.19.11/source/include/uapi/sound/asound.h#L858)
- count: a snd_kcontrol below may have more elem , a elem case may also have more than one value . In snd_ctl_elem_value can more clearly the count sense
- value: This is a big consortium , which is used to show the user space elem available range of values . In particular consortium is valid and which one element type related : If the type is bool / int type , the user reads space value. Integer to obtain the EL EM desirable minimum / maximum / step ; if the type is int64 type , the user space reads value.integer64 ; If the type is the enumerate , the user reads the space value. enumerated to get the elem What is desirable value . Specific examples are byte type , the data in this case will be as bin to handle , so there can not be feedback bin details of the content .

struct snd_ctl_elem_value

struct snd_ctl_elem_value (https://elixir.bootlin.com/linux/v4.19.11/source/include/uapi/sound/asound.h#L935) : Describes a storage space . As mentioned earlier, snd_kcontrol only contains methods for reading and writing data, and does not include the space for storing data . The data storage space is defined by this structure .

```
struct snd_ctl_elem_value {
        struct snd_ctl_elem_id id;      /* W: element ID */
        unsigned int indirect: 1;       /* W: indirect access - obsoleted */
        union {
                union {
                        long value[128];
                        long *value_ptr;        /* obsoleted */
                } integer;
                union {
                        long long value[64];
                        long long *value_ptr;   /* obsoleted */
                } integer64;
                union {
                        unsigned int item[128];
                        unsigned int *item_ptr; /* obsoleted */
                } enumerated;
                union {
                        unsigned char data[512];
                        unsigned char *data_ptr;        /* obsoleted */
                } bytes;
                struct snd_aes_iec958 iec958;
        } value;                /* RO */
        struct timespec tstamp;
        unsigned char reserved[128-sizeof(struct timespec)];
};
```

- id: token , when the user space wants to read and write the value of an elem , the user space will prepare the snd_ctl_elem_value data structure , and fill the elem 's token id into the structure , and then pass the structure to the kernel . ' C ontrol The middle layer will find the corresponding snd_kcontrol through the id , and then call back its get/put function . The bottom driver fills the content of the structure in the get/put function , and then returns it to the user space .
- value: This is a major consortium , in particular complexes in which the active element snd_ctl_elem_info specified type information : If the type is bool / int type , the . Value Integer valid ; if the type is int64 type , the value.integer64 valid ; if the type is the enumerate , then the value. enumerated valid ; if the type is byte type , the value.bytes . effective

  Also , note that each of the active storage is an array type , e.g. value.integer.value [128]. Mentioned before a elem can have multiple values , the specific number depending snd_ctl_elem_info .count. Here value. integer.value [128] mean count no more than 128 , if the count is 1, then value.integer.value [0] represents valid data , if the count is 3, the value.integer.value [0, 1, 2] representatives Valid data .

struct snd_kcontrol_new

struct snd_kcontrol_new (https://elixir.bootlin.com/linux/v4.19.11/source/include/sound/control.h#L46) : When the driver wants to register a bottom kcontrol time , must be filled of the structure , and the ' C ontrol intermediate layer' register .

```
struct snd_kcontrol_new {
        snd_ctl_elem_iface_t iface;     /* interface identifier */
        unsigned int device;            /* device/client number */
        unsigned int subdevice;         /* subdevice (substream) number */
        const unsigned char *name;      /* ASCII name of item */
        unsigned int index;             /* index of item */
        unsigned int access;            /* access rights */
        unsigned int count;             /* count of same elements */
        snd_kcontrol_info_t *info;
        snd_kcontrol_get_t *get;
        snd_kcontrol_put_t *put;
        union {
                snd_kcontrol_tlv_rw_t *c;
                const unsigned int *p;
        } tlv;
        unsigned long private_value;
};
```

- iface: snd_ctl_elem_iface_t (https://elixir.bootlin.com/linux/v4.19.11/source/include/uapi/sound/asound.h#L848) type , generally used SNDRV_CTL_ELEM_IFACE_MIXER . Different types do not have any special treatment at the kernel code level , and it seems that this type definition does not have much effect . The only known effect is " iface= " when the amixer controls command is used . The fields will display different strings . (https://elixir.bootlin.com/linux/v4.19.11/source/include/uapi/sound/asound.h#L848)
- Device / subdevice / name / index: when a bottom register kcontrol time , the core layer creates a corresponding snd_kcontrol , and creates a corresponding token S nd_ctl_elem_id be uniquely identifies the kcontrol . These fields will be assigned to the S nd_ctl_elem_id respective Field .
  Regarding the name field , the official naming rule (https://www.kernel.org/doc/html/latest/sound/designs/control-names.html) : source - direction - function . However, the code level does not seem to do any mandatory check on the name , it seems that it doesn't matter if you take it casually . Maybe this rule is just a soft constraint .
- access: access authority , refer to SNDRV_CTL_ELEM_ACCESS_ (https://elixir.bootlin.com/linux/v4.19.11/source/include/uapi/sound/asound.h#L858) * for optional values . (https://elixir.bootlin.com/linux/v4.19.11/source/include/uapi/sound/asound.h#L858)
- count: the same as the meaning of count in snd_kcontrol .
- info / get / put: The bottom layer needs to implement these three functions .
- TLV: the so-called TLV , is the Type-Lenght-Value means , is a meta-data . In particular significance here temporarily I do not know .
- private_value: Same as the meaning of private_value in snd_kcontrol . The underlying driver usually stores the codec register address corresponding to the kcontrol in this field .

When the underlying data structure is ready , it can be called ' C ontrol intermediate layer' provided API register a kcontrol up . Let's see the API.

1.5.2.2 API

' C ontrol intermediate layer' to use the underlying API. Including two : snd_ctl_new1 and snd_ctl_add .

snd_ctl_new1

snd_ctl_new1 (https://elixir.bootlin.com/linux/v4.19.11/source/sound/core/control.c#L247) is defined in sound/core/control.c .

```
/**
 * snd_ctl_new1 - create a control instance from the template
 * @ncontrol: the initialization record
 * @private_data: the private data to set
 *
 * Allocates a new struct snd_kcontrol instance and initialize from the given
 * template.  When the access field of ncontrol is 0, it's assumed as
 * READWRITE access. When the count field is 0, it's assumes as one.
 *
 * Return: The pointer of the newly generated instance, or %NULL on failure.
 */
struct snd_kcontrol *snd_ctl_new1(const struct snd_kcontrol_new *ncontrol,
                                  void *private_data)
{
```

When the bottom is ready snd_kcontrol_new the data structure , you can call this API to build a snd_kcontrol . Then invoked snd_ctl_add to return this snd_kcontrol registered to ' C ontrol intermediate layer' .

Relatively simple logic function , first make the necessary checks on the input parameters , and then assign a snd_kcontrol storage space , and then initializes the appropriate field of the storage space with the input parameters , and returns that snd_kcontrol.

snd_ctl_add

snd_ctl_add (https://elixir.bootlin.com/linux/v4.19.11/source/sound/core/control.c#L399) in / core / control.c source / sound defined .

```
/**
 * snd_ctl_add - add the control instance to the card
 * @card: the card instance
 * @kcontrol: the control instance to add
 *
 * Adds the control instance created via snd_ctl_new() or
 * snd_ctl_new1() to the given card. Assigns also an unique
 * numid used for fast search.
 *
 * It frees automatically the control which cannot be added.
 *
 * Return: Zero if successful, or a negative error code on failure.
 *
 */
int snd_ctl_add(struct snd_card *card, struct snd_kcontrol *kcontrol)
```

The main logic of this function is divided into 3 sections :

First , mount this kcontrol structure under the card-> controls linked list header . So that the middle layer can manage all kcontrols through this header .

```
list_add_tail(&kcontrol->list, &card->controls);
```

Secondly , kcontrol-> the above mentioned id this token , in snd_ctl_new1 stage has initiated most of the elements , but there is an uninitialized , it is kcontrol -> the above mentioned id. Numid . Here to complete its initialization :

```
card->controls_count += kcontrol->count;
kcontrol->id.numid = card->last_numid + 1;
card->last_numid += kcontrol->count;
```

This numid is unique under a certain card . Through numid , we can quickly locate a certain snd_kcontrol under the card . Code reference snd_ctl_find_numid (https://elixir.bootlin.com/linux/v4.19.11/source/sound/core/control.c#L675) .
(https://elixir.bootlin.com/linux/v4.19.11/source/sound/core/control.c#L675)

Finally , notify the user space , there is count one elem registered into the kernel , and inform the user space each elem of the token.

```
id = kcontrol->id;
count = kcontrol->count;
for (idx = 0; idx < count; idx++, id.index++, id.numid++)
        snd_ctl_notify(card, SNDRV_CTL_EVENT_MASK_ADD, &id);
```

This event notification mechanism is completed by the read/poll function of the character device driver , which will be described in detail later .

1.5.2.3 add a kcontrol

Through the previous introduction , we have mastered the method of adding a kcontrol , the following is a simple demonstration :

Step1 : Define the snd_kcontrol_new structure .

```
static struct snd_kcontrol_new my_control __devinitdata = {
    .iface = SNDRV_CTL_ELEM_IFACE_MIXER,
    .name = "PCM Playback Switch",
    .index = 0,
    .access = SNDRV_CTL_ELEM_ACCESS_READWRITE,
    .private_value = 0xffff,
    .info = my_control_info,
    .get = my_control_get,
    .put = my_control_put
};
```

Step2 : Implement the info/get/put method .

```
static int my_control_info (struct snd_kcontrol *kcontrol,
    struct snd_ctl_elem_info *uinfo)
{
    uinfo->type = SNDRV_CTL_ELEM_TYPE_BOOLEAN;
    uinfo->count = 1;
    uinfo->value.integer.min = 0;
    uinfo->value.integer.max = 1;
    return 0;
}
```

Alsa has implemented some general info callback functions for us , such as : snd_ctl_boolean_mono_info() , snd_ctl_boolean_stereo_info(), etc. , you can use them directly if necessary .

```
static int my_control_get(struct snd_kcontrol *kcontrol,
    struct snd_ctl_elem_value *ucontrol)
{
    struct mychip *chip = snd_kcontrol_chip(kcontrol);
    ucontrol->value.integer.value[0] = get_some_value(chip);
    return 0;
}
```

If the count field of snd_ctl_elem_info is greater than 1 , it means that elem has multiple element units , and the get callback function should also fill multiple elements for value [] .

```
static int my_control_put(struct snd_kcontrol *kcontrol,
    struct snd_ctl_elem_value *ucontrol)
{
    struct mychip *chip = snd_kcontrol_chip(kcontrol);
    int changed = 0;
    if (chip->current_value !=
        ucontrol->value.integer.value[0]) {
        change_current_value(chip,
        ucontrol->value.integer.value[0]);
        changed = 1;
    }
    return changed;
}
```

As shown in the above example , when the value of control is changed , the put callback must return 1, and if the value is not changed , it returns 0. If an error occurs , it returns a negative error number .

And get callbacks as , when the count is greater than . 1 when , PUT callback should Processing value [] a plurality of element values .

Step3 : Register kcontrol .

```
err = snd_ctl_add(card, snd_ctl_new1(&my_control, chip));
if (err <0)
    return err;
```

1.5.2.4 Control character device

Each card has a lower C ontrol device , it is with the creation of a sound card automatically generated , as long as we have added a sound card in the system , then this card under Control device was created at the same time .

Let's look at how the Control device is automatically created from the code level :

- First , snd_card_new -> snd_ctl_create (https://elixir.bootlin.com/linux/v4.19.11/source/sound/core/control.c#L1893) -> snd_device_new , that is to say whenever new when a sound card , it will automatically add a sound card Under Control logic devices . (https://elixir.bootlin.com/linux/v4.19.11/source/sound/core/control.c#L1893)
- Then , in snd_card_register stage , will for the card to call each logical device under snd_device_register . This action will eventually be registered callback logic devices snd_device_ops . Dev_register . For C ontrol equipment is , ultimately is snd_ctl_dev_register (https://elixir.bootlin.com/linux/latest/source/sound/core/control.c#L1845) be recalled . Snd_ctl_dev_register then call snd_register_device , Created a C ontrol character device node . (https://elixir.bootlin.com/linux/latest/source/sound/core/control.c#L1845)

User space operations on Control character device nodes will eventually be transferred to snd_ctl_f_ops (https://elixir.bootlin.com/linux/latest/source/sound/core/control.c#L1829) :

```
static const struct file_operations snd_ctl_f_ops =
{
        .owner =        THIS_MODULE,
        .read =         snd_ctl_read,
        .open =         snd_ctl_open,
        .release =      snd_ctl_release,
        .llseek =       no_llseek,
        .poll =         snd_ctl_poll,
        .unlocked_ioctl =       snd_ctl_ioctl,
        .compat_ioctl = snd_ctl_ioctl_compat,
        .fasync =       snd_ctl_fasync,
};
```

The function of this ops can be divided into two major blocks : read / poll is responsible for reporting snd_ctl_event events to user space ; ioctl is responsible for responding to user space to check / read / write kcontrol .

snd_ctl_event event

Event events include element add/remove/change these categories , see SNDRV_CTL_EVENT_XXX (https://elixir.bootlin.com/linux/latest/source/include/uapi/sound/asound.h#L1002) for details .
event significance of the event is when a new kernel space kcontrol time , the kernel needs to inform the user space kcontrol.count one element is added into the kernel , and user space can check / read / write these elements. Similarly , when kcontrol is When deleting , or when the content of the element itself changes , the user space needs to be notified .

Kernel with struct snd_ctl_event (https://elixir.bootlin.com/linux/latest/source/include/uapi/sound/asound.h#L1008) describe an event event :

```
struct snd_ctl_event {
        int type;       /* event type - SNDRV_CTL_EVENT_* */
        union {
                struct {
                        unsigned int mask;
                        struct snd_ctl_elem_id id;
                } elem;
                unsigned char data8[60];
        } data;
};
```

- type: represents the event type .
- data.elem.id is the token of this elem .

User space acquired event event mode is : user space open device nodes after , can be read or poll the node . In the core layer read / poll function , will wait at change_sleep this wait_queue_head on . When the specific event occurs , e.g. call snd_ctl_add add a kcontrol time , calls snd_ctl_notify (https://elixir.bootlin.com/linux/latest/source/sound/core/control.c#L156) . in the notify function calls wake_up (& ctl-> change_sleep) wake-waiting queue , so that the read / poll from the waitThe state wakes up , which in turn causes the user space to receive the event event .

ioctl

When user space checks / reads / writes elem information through amixer or other methods , it actually interacts with the kernel through ioctl .

The kernel's snd_ctl_ioctl (https://elixir.bootlin.com/linux/latest/source/sound/core/control.c#L1547) is responsible for processing ioctl commands . For a list of all available commands , see SNDRV_CTL_IOCTL_XXX (https://elixir.bootlin.com/linux/latest/source/include/uapi/sound/asound.h#L967) . Several commonly used commands are as follows :

```
static long snd_ctl_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
        struct snd_ctl_file *ctl;
        struct snd_card *card;
        struct snd_kctl_ioctl *p;
        void __user *argp = (void __user *)arg;
        int __user *ip = argp;
        int err;

        ctl = file->private_data;
        card = ctl->card;
        if (snd_BUG_ON(!card))
                return -ENXIO;
        switch (cmd) {
        case SNDRV_CTL_IOCTL_PVERSION:
                return put_user(SNDRV_CTL_VERSION, ip) ? -EFAULT : 0;
        case SNDRV_CTL_IOCTL_CARD_INFO:
                return snd_ctl_card_info(card, ctl, cmd, argp);
        case SNDRV_CTL_IOCTL_ELEM_LIST:
                return snd_ctl_elem_list(card, argp);
        case SNDRV_CTL_IOCTL_ELEM_INFO:
                return snd_ctl_elem_info_user(ctl, argp);
        case SNDRV_CTL_IOCTL_ELEM_READ:
                return snd_ctl_elem_read_user(card, argp);
        case SNDRV_CTL_IOCTL_ELEM_WRITE:
                return snd_ctl_elem_write_user(ctl, argp);
        case SNDRV_CTL_IOCTL_ELEM_LOCK:
                return snd_ctl_elem_lock(ctl, argp);
        case SNDRV_CTL_IOCTL_ELEM_UNLOCK:
                return snd_ctl_elem_unlock(ctl, argp);
        case SNDRV_CTL_IOCTL_ELEM_ADD:
                return snd_ctl_elem_add_user(ctl, argp, 0);
        case SNDRV_CTL_IOCTL_ELEM_REPLACE:
                return snd_ctl_elem_add_user(ctl, argp, 1);
        case SNDRV_CTL_IOCTL_ELEM_REMOVE:
                return snd_ctl_elem_remove(ctl, argp);
        case SNDRV_CTL_IOCTL_SUBSCRIBE_EVENTS:
                return snd_ctl_subscribe_events(ctl, ip);
```

These commands are basically self-explanatory , for example :

- LIST is to query all the elems under the card ;
- INFO is to obtain the specific information of an elem , which will eventually cause the underlying info function to be called ;
- READ/WRITE is to read and write the value of elem , which will eventually cause the underlying get/put function to be called ;
- ADD/REMOVE is to add / delete an elem in the user space , which will also cause the kernel space to create a snd_kcontrol to hold the elem .
- REPLACE represents the user space to modify the information of an existing elem .

1.5.3 PCM logic device

The PCM logic device is similar to other logic devices in the system :

- PCM logic devices are also presented to the user space in the form of character devices ;
- The kernel also implements a ' PCM middle layer' , the intermediate layer on the ALSA registered core layer , will register itself as a logical device , and ultimately create a character device node , is responsible for user-space interaction with (snd_pcm_f_ops (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L3647)); the next provides the API to the underlying driver , for driving the bottom register a PCM device . (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L3647)

From a functional perspective , PCM effect logic device is space for a user playback / recording PCM audio . PCM audio essence is a Buffer, " PCM effect of the intermediate layer 'is received from a user this space B uffer, then Buffer data is transferred to the underlying driving play . underlying driver generally corresponds I2S controller ,

after it receives the data , will pass I2S always the data to the line codec, then the codec through DA converted to the playback speakers .

In addition to B uffer transmission , we have to solve the problem of configuration , which is configured I2S Controller and Codec chip , to tell them what should be in accordance Clock , sampling depth, number of channels to play B uffer data . Therefore ' PCM the intermediate layer 'also provides some ioctl to the user space , for their setting parameters .

From the perspective of division of labor, what should the PCM middle layer and bottom driver focus on ?
The PCM intermediate layer mainly completes some common things , which are generally not related to specific hardware (for example, when storing audio data , a Buffer is definitely needed . This Buffer is obviously better described by a ring buffer . How to manage this ring buffer read and write pointers ? in addition , audio parameters set by the user space to check its legitimacy . there , the user space may start and stop audio playback / recording , which means we'd better realize a state machine , to manage the switching operation of the user space . etc.) . for operations related to specific hardware , PCM middle layer will pass to the underlying driver to handle (this means PCM middle layer to define the interface, so that the underlying drive to achieve these interface) .
The underlying driver handles things related to specific hardware , such as the configuration of the I2S controller, the configuration of the Codec chip , the initialization of the clock, and so on .

Before the introduction of the ASOC architecture , the implementation of the underlying driver was placed in a file, and no further architectural design was carried out on it , which resulted in poor code reusability . For example, different I2S controllers are connected to the same Codec. chip , it would need two bottom-driven , in that the two driving , needs to contain C ODEC control logic , this logic is repeated . later introduced core ASOC architecture , divided M achine , Platform , C ODEC these modules , increasing the reusability of code . in " ASOC chapter," we will specifically discuss this architecture ,Therefore, this section will not describe the details of the underlying driver too much , but briefly mention it when necessary .
This section will focus on the implementation of the'PCM middle layer' , that is, the ring buffer management, state machine management, and definition of inter fac e mentioned above .

Next , let's start the journey!

1.5.3.1 Basic knowledge

Before we dive into the code , let's introduce some necessary basic knowledge , which is very helpful for us to understand the code behind .

Glossary

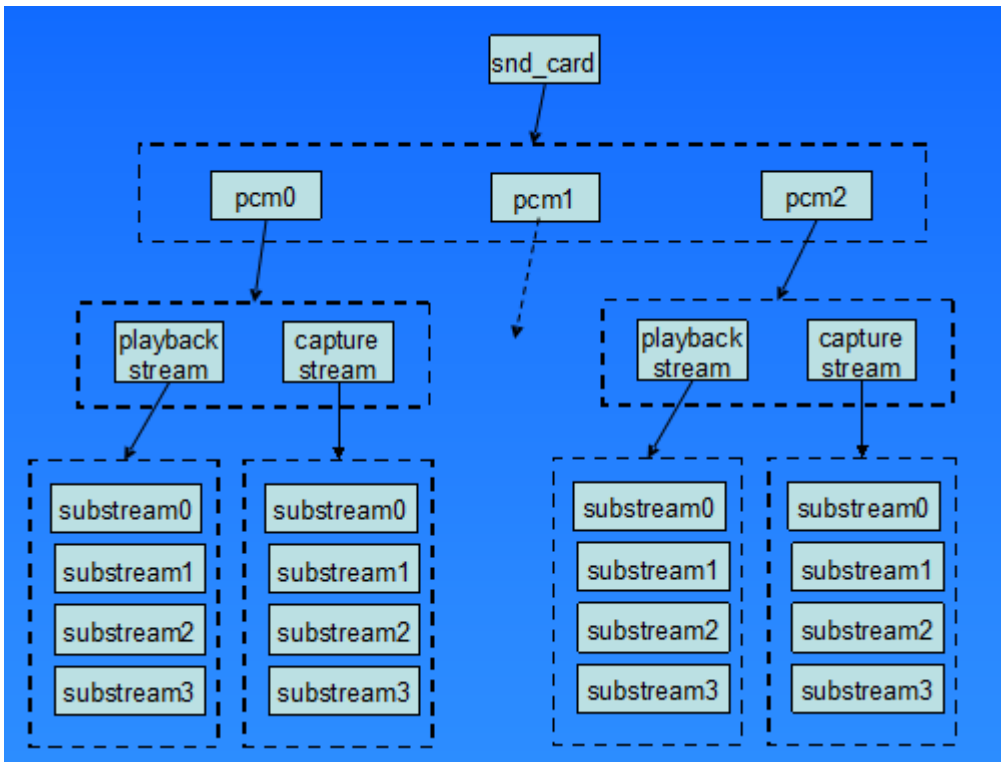
Here are a few audio samples associated with the term , follow-up in understanding the audio B uffer time management , these terms are important .

- **Sample** : sample length . It represents the audio data of one channel , the size depends on the sampling depth , and the common ones are 8 bits and 16 bits .
- **Channel** : number of channels . Common are mono, two-channel (stereo), . 5 .1 channels . 7 .1 channel and the like .
- **Frame** : Frame , which constitutes a complete sound unit . Frame = C hannel * Sample, for example, for mono , its size is 1* Sample, for 5.1 channels , its size is 6* Sample.

The concept of frame here is similar to the frame in LCD , which is the basic unit of hardware transmission . For example, I2S transmits a complete frame every time , and when a frame is transmitted , a hardware interrupt will be generated .

- **Rate** : Sampling rate , refers to the time required to sample **one frame of** data . For example, 44.1KH z means that it takes 1s/44100 = 0.022ms to sample one frame of data .
- **Period** : cycle . When the audio data is written to the user space of the RAM after , the bottom layer is DMA data from RAM move to I2S of the FIFO. DMA Each section will move Ends generate an interrupt after the data . We DMA move this process data is called a cycle . the size of this data is configurable , the user can set the space P eriod size , if the period is set to be large , more data is a single move , which means that the unit time With fewer internal hardware interrupts , the CPU has more time to process other tasks , and the power consumption is lower , but this also brings a significant drawback-the delay of data processing will increase .
- **period_size** : The number of frames in a period . It indirectly determines the size of the period . When the user space sets the period size , the given parameter is also the number of frames .
- **period_bytes** : For DMA hardware , it only cares about how many bytes the data has.period_bytes = period_size * C hannel s * Sample / 8.
- **Buffer** : representatives of a RAM, a user space and the kernel through this RAM to exchange data . The DMA also from this RAM move data . A B uffer comprises a plurality of inner Period.
- **buffer_size** : a Buffer inner frame number , this Buffer contained within a plurality of period.
- **buffer_bytes** : The size of the Buffer in bytes , which is usually needed when allocating RAM . buffer_bytes = buffer_size * C hannel s * Sample / 8.

The basic structure of PCM



- a pcm instance (e.g. PCM0) is card a logical device under , this logic device creates two device nodes in user space .
- a pcm instance contains two stream: Playback & Capture. Each stream corresponding to a node device .
- Each stream under may comprise a plurality of substream.

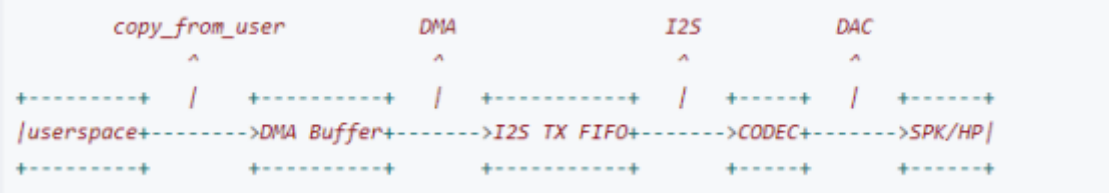
At the kernel layer , each substream has its own Buffer to exchange audio data with user space . From this perspective , the meaning of substream seems to be to time- share the underlying audio hardware .

In the user space , each device node may be open multiple times , each open core layer will find a free substream corresponding thereto , if the kernel layer S ubstream used up , then the open operation would fail . So , user space to read, write, and control operations are directed substream carried out , which further illustrate substream can be used to time division multiplexing underlying audio hardware .

PCM ring buffer management

The previous article mentioned many times that user space and kernel space exchange audio data through a Buffer . What kind of Buffer is this?

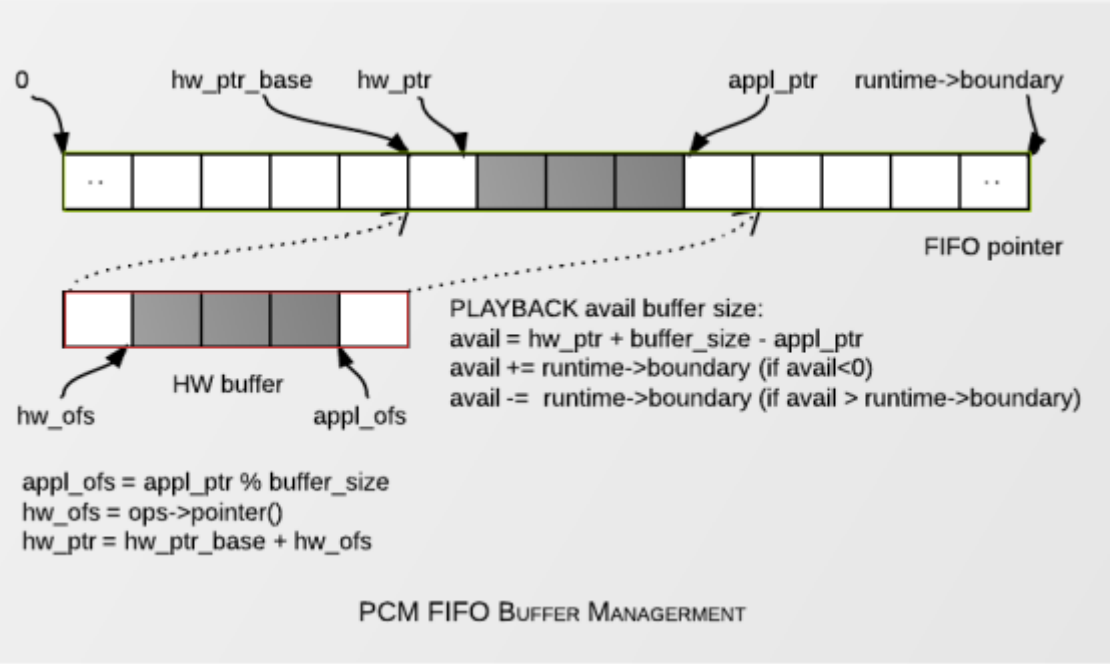
In general , this Buffer is continuous over a period of physical memory (more precisely , a piece can be DMA memory access), this memory space is created by the kernel .
For Playback, user space to this Buffer write data , and DMA data from B uffer move to I2S of the FIFO .
For capture, DMA moves the data from the I2S FIFO to this Buffer, and then the user space reads the data from the Buffer .



Obviously , this Buffer involves the production of one party and the consumption of the other . Therefore, it is most reasonable to use a ring buffer to manage this Buffer .
In the ring buffer mechanism , when the read / write pointer grow to Buffer when the border , require from 0 to re-count start . Should also be considered equal read and write pointers , Buffer is full or empty question . Ring buffer can have a variety Implementation , different implementations have different ways to solve these problems . Some implementations are very clever , such as the kernel's kfifo , which uses the power of 2 feature to make it very simple to solve the above problems . If you are interested Get to know it .

In ALSA in , it has its own set of ways to implement ring buffer , and kfifo quite similar , but personally feel no kfifo easy to use . Maybe ALSA is done to solve kfifo waste problem of space , kfifo space requirements will be allocated It is now a power of 2 regardless of the size you actually use , but the logic of ALSA does not have this limit .

The details of the ring buffer implemented by ALSA are as follows :



The red box represents the size of the actual Buffer , and the green box represents the size of the logical Buffer . Let's take P layback as an example to introduce the meaning of several main parameters :

- hw_ptr : Read pointer , which represents the location where DMA moves data .
- appl_ptr : write pointer , which represents the location of the user space to write data .
- buffer_size : the actual size of the buffer .
- runtime->boundary : The logical size of the Buffer , which is generally an integer multiple of the actual size . When the read and write pointer exceeds the boundary , it will restart counting from 0 .

Let's first look at the write pointer a ppl_ptr. The write pointer is incremented on the logical Buffer , but we can't say that writing data to the logical position , we need to find the actual writing position first . Then how to get the actual writing position through the logical position ?

- appl_ofs = appl_ptr% buffer_size : The modulo operation , ingenious obtained the actual write pointer Buffer Offset in .
- real_appl_ptr = real_buffer_base + appl_ofs : with an offset + actual Buffer start address , can be obtained the actual write address . Allocation Buffer would the real time Buffer start position stored .

Look at the read pointer hw_ptr. Think about when it will update hw_ptr? Usually when DMA finished moving a period after the data , generates an interrupt , update interrupt handling function hw_ptr more appropriate . That how to update blanket ?

- hw_of s = ops->pointer() : First , obtain the offset of the read pointer in the actual Buffer through the underlying driver .
- hw_ptr = hw_ptr_base + hw_ofs : hw_ptr_base is the base address of the Buffer in the logical position , base address + offset , you can get the position of the read pointer on the logical address . The code will update the value of hw_ptr_base in a timely manner during the execution of the code .

During the update process of the read and write pointers , the kernel will ensure that they meet certain constraints :

- ✓ First , because appl_ptr and hw_ptr is a pointer to a logical , therefore theoretically appl_ptr - hw_ptr may be greater than the actual Buffer size . Buffer_size but ensure that the kernel will appl_ptr and hw_ptr distance of not greater than buffer_size , because equal buffer_size mean impending overrun Event , that is, the speed of user space writing data is too fast , DMA ca n't move them , and finally there is no free buffer to write to user space . At this time, the kernel does not allow user space to continue writing , so it will not continue to update appl_ptr Up .

- ✓ Secondly , hw_ptr in the process of growth in , will not exceed appl_ptr. Because when hw_ptr grow to the appl_ptr equal , means imminent underrun events , that is, DMA move too fast , user space to write too slow , no The new data can be moved . At this time, the kernel will generally suspend the DMA movement , and then we may hear a short popping sound (of course, the kernel can also be configured to move a piece of old data to avoid the popping sound) . When this problem occurs, the kernel does not We will continue to increase hw_ptr up , so it does not cross the appl_ptr.

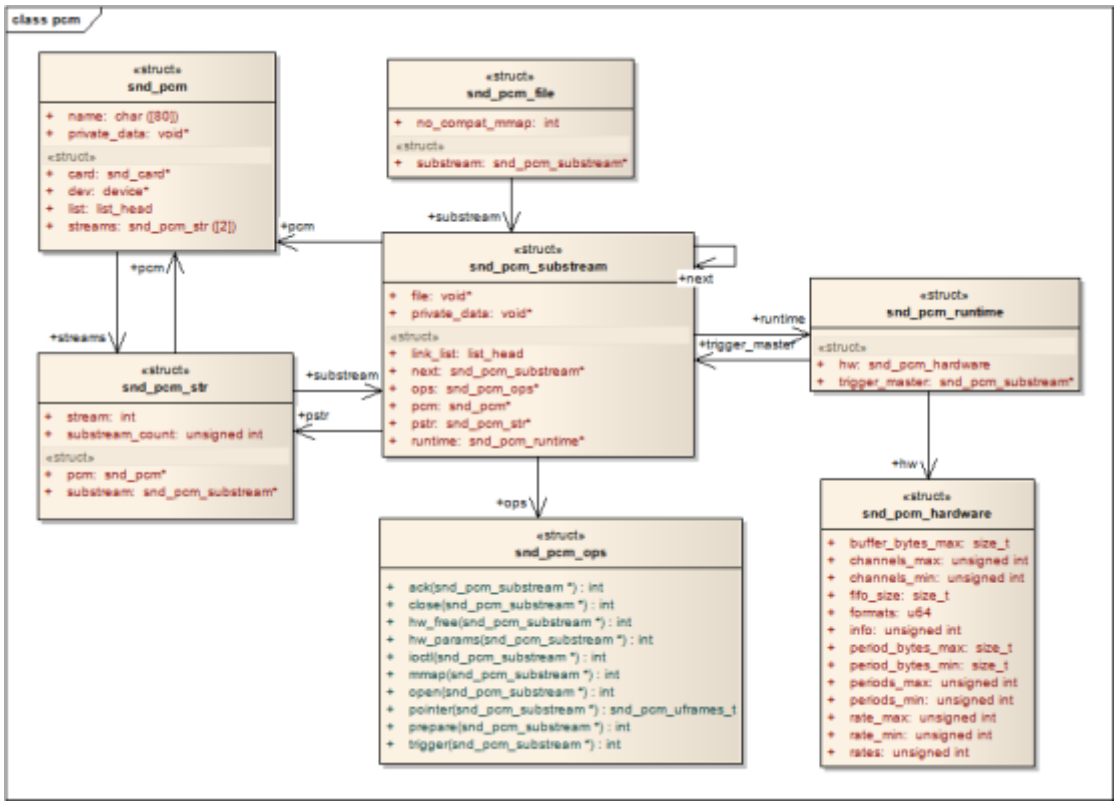
Why should incorporate logic Buffer this stuff ? Its purpose is to facilitate the calculation of the free buffer size :

- ➢ A Vail buffer_size = hw_ptr + - appl_ptr : (logical time position + actual buffer size) - Logical Write location is idle buffer size .
- ➢ Avail - = runtime-> boundary (IF Avail> runtime-> boundary) : When will avail> runtime-> boundary? When appl_ptr crossed the boundary be reassigned to 0, hw_ptr very close to the boundary but has not yet crossed the boundary , this The avail calculated at the time is larger than the boundary. But the actual free buffer is not so large , you need to subtract the boundary. If you don't understand, draw it on the paper .
- ➢ avail += runtime->boundary (if avail<0) : When will avail <0 appear ? It seems very rare . Only when the boundary is set very large , the hw_ptr+buffer_size has an integer data overflow problem . After the overflow Because the highest bit is discarded , it becomes a very small number . At this time, hw_ptr+buffer_size - appl_ptr will be less than 0. But even if this problem occurs , avail += runtime->boundary can also get the actual free space .

Regarding the details of the ring buffer code level , we will introduce them in the section `` struct snd_pcm_runtime " .

1.5.3.2 Data structure

Let's first look at the full picture to facilitate the viewing of the affiliation between various data structures .



struct snd_pcm

struct snd_pcm (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L525) : Represents a pcm instance and also represents a pcm logical device .

```
struct snd_pcm {
        struct snd_card *card;
        struct list_head list;
        int device; /* device number */
        unsigned int info_flags;
        unsigned short dev_class;
        unsigned short dev_subclass;
        char id[64];
        char name[80];
        struct snd_pcm_str streams[2];
        struct mutex open_mutex;
        wait_queue_head_t open_wait;
        void *private_data;
        void (*private_free) (struct snd_pcm *pcm);
        bool internal; /* pcm is for internal use only */
        bool nonatomic; /* whole PCM operations are in non-atomic context */
#if IS_ENABLED(CONFIG_SND_PCM_OSS)
        struct snd_pcm_oss oss;
#endif
};
```

- list: Mount yourself under the card-> devices header .
- streams: point to its subordinate playback stream and capture stream.

struct snd_pcm_str

struct snd_pcm_str (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L502) : represents a pcm stream.

```
struct snd_pcm_str {
        int stream;                             /* stream (direction) */
        struct snd_pcm *pcm;
        /* -- substreams -- */
        unsigned int substream_count;
        unsigned int substream_opened;
        struct snd_pcm_substream *substream;
#if IS_ENABLED(CONFIG_SND_PCM_OSS)
        /* -- OSS things -- */
        struct snd_pcm_oss_stream oss;
#endif
#ifdef CONFIG_SND_VERBOSE_PROCFS
        struct snd_info_entry *proc_root;
        struct snd_info_entry *proc_info_entry;
#ifdef CONFIG_SND_PCM_XRUN_DEBUG
        unsigned int xrun_debug;        /* 0 = disabled, 1 = verbose, 2 = stacktrace */
        struct snd_info_entry *proc_xrun_debug_entry;
#endif
#endif
        struct snd_kcontrol *chmap_kctl; /* channel-mapping controls */
        struct device dev;
};
```

- dev: A steam corresponds to a character device node . This dev is related to the creation of a character device node .
- substream_count: The number of substreams under the stream .
- substream_opened: How many substream has been user-space open up . User space for the same device node can open multiple times , each open INL will choose a free substream corresponding . If all substream are opened, the The new open will fail .
- substream: Concatenate multiple substreams in the form of a linked list .

struct snd_pcm_file

struct snd_pcm_file (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L226) : each open the substream corresponding to a snd_pcm_file.

```
struct snd_pcm_file {
        struct snd_pcm_substream *substream;
        int no_compat_mmap;
        unsigned int user_pversion;     /* supported protocol version */
};
```

struct snd_pcm_substream

struct snd_pcm_substream (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L447) : Represents a pcm substream.An important function of substream is to prepare a DMA buffer to exchange data with user space .

```
struct snd_pcm_substream {
        struct snd_pcm *pcm;
        struct snd_pcm_str *pstr;
        void *private_data;             /* copied from pcm->private_data */
        int number;
        char name[32];                  /* substream name */
        int stream;                     /* stream (direction) */
        struct pm_qos_request latency_pm_qos_req; /* pm_qos request */
        size_t buffer_bytes_max;        /* limit ring buffer size */
        struct snd_dma_buffer dma_buffer;
        size_t dma_max;
        /* -- hardware operations -- */
        const struct snd_pcm_ops *ops;
        /* -- runtime information -- */
        struct snd_pcm_runtime *runtime;
        /* -- timer section -- */
        struct snd_timer *timer;                /* timer */
        unsigned timer_running: 1;      /* time is running */
        long wait_time; /* time in ms for R/W to wait for avail */
        /* -- next substream -- */
        struct snd_pcm_substream *next;
        /* -- linked substreams -- */
        struct list_head link_list;     /* linked list member */
        struct snd_pcm_group self_group;        /* fake group for non linked substream (with substream lock inside) */
        struct snd_pcm_group *group;            /* pointer to current group */
        /* -- assigned files -- */
        void *file;
        int ref_count;
        atomic_t mmap_count;
        unsigned int f_flags;
        void (*pcm_release)(struct snd_pcm_substream *);
        struct pid *pid;
#if IS_ENABLED(CONFIG_SND_PCM_OSS)
        /* -- OSS things -- */
        struct snd_pcm_oss_substream oss;
#endif
#ifdef CONFIG_SND_VERBOSE_PROCFS
        struct snd_info_entry *proc_root;
        struct snd_info_entry *proc_info_entry;
        struct snd_info_entry *proc_hw_params_entry;
        struct snd_info_entry *proc_sw_params_entry;
        struct snd_info_entry *proc_status_entry;
        struct snd_info_entry *proc_prealloc_entry;
        struct snd_info_entry *proc_prealloc_max_entry;
#ifdef CONFIG_SND_PCM_XRUN_DEBUG
        struct snd_info_entry *proc_xrun_injection_entry;
#endif
#endif /* CONFIG_SND_VERBOSE_PROCFS */
        /* misc flags */
        unsigned int hw_opened: 1;
};
```

- buffer_bytes_max : the maximum value of buffer_bytes, buffer_bytes represents the actual size of the DMA buffer created by the substream .
- dma_buffer : used to describe the DMA buffer created by the substream . For details, see " struct snd_dma_buffer " .
- dma_max: It doesn't seem to have much effect , and it is generally equal to buffer_bytes_max .
- ops: Interface functions that need to be implemented by the underlying driver , see `` struct snd_pcm_ops " for details .
- runtime: Save some runtime information of the substream during its operation . The so-called operation process refers to the period of time after the substream is opened and before it is closed . For details, see " struct snd_pcm_runtime " .
- next: point to the next substream, so that all substreams can be stringed together in the form of a linked list .

struct snd_dma_buffer

struct snd_dma_buffer (https://elixir.bootlin.com/linux/v4.20/source/include/sound/memalloc.h#L67) : used to describe a DMA buffer.

```
struct snd_dma_buffer {
        struct snd_dma_device dev;      /* device type */
        unsigned char *area;    /* virtual pointer */
        dma_addr_t addr;        /* physical address */
        size_t bytes;           /* buffer size in bytes */
        void *private_data;     /* private for allocator; don't touch */
};
```

- Area : buffer virtual address , for CPU access buffer used .
- addr: buffer physical addresses , for DMA access buffer used .
- bytes: the size of the buffer .

struct snd_pcm_runtime

struct snd_pcm_runtime (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L351) : Saves various parameters of a substream during operation , mainly including these aspects : audio format, sampling rate and other related parameters . These parameters are closely related to the concepts we introduced in the ``Name Explanation'' ; ring Buffer management related parameters ; when an overrun/underrun event occurs , the parameters that indicate how to handle the event .

This section is very long and complex , and involves a lot of knowledge . But to understand them , you can basically understand the core logic of the'PCM middle layer' . The following are the parameters of these aspects :

```
struct snd_pcm_runtime {
        /* -- Status -- */
        struct snd_pcm_substream *trigger_master;
        struct timespec trigger_tstamp;/* trigger timestamp */
        bool trigger_tstamp_latched;   /* trigger timestamp latched in low-level driver/hardware */
        int overrange;
        snd_pcm_uframes_t avail_max;
        snd_pcm_uframes_t hw_ptr_base;      /* Position at buffer restart */
        snd_pcm_uframes_t hw_ptr_interrupt; /* Position at interrupt time */
        unsigned long hw_ptr_jiffies;    /* Time when hw_ptr is updated */
        unsigned long hw_ptr_buffer_jiffies;/* buffer time in jiffies */
        snd_pcm_sframes_t delay;      /* extra delay; typically FIFO size */
        u64 hw_ptr_wrap;            /* offset for hw_ptr due to boundary wrap-around */

        /* -- HW params -- */
        snd_pcm_access_t access;      /* access mode */
        snd_pcm_format_t format;      /* SNDRV_PCM_FORMAT_* */
        snd_pcm_subformat_t subformat;      /* subformat */
        unsigned int rate;            /* rate in Hz */
        unsigned int channels;        /* channels */
        snd_pcm_uframes_t period_size;      /* period size */
        unsigned int periods;         /* periods */
        snd_pcm_uframes_t buffer_size;      /* buffer size */
        snd_pcm_uframes_t min_align;/* Min alignment for the format */
        size_t byte_align;
        unsigned int frame_bits;
        unsigned int sample_bits;
        unsigned int info;
        unsigned int rate_num;
        unsigned int rate_den;
        unsigned int no_period_wakeup: 1;

        /* -- SW params -- */
        int tstamp_mode;              /* mmap timestamp is updated */
        unsigned int period_step;
        snd_pcm_uframes_t start_threshold;
        snd_pcm_uframes_t stop_threshold;
        snd_pcm_uframes_t silence_threshold; /* Silence filling happens when
                                                noise is nearest than this */
        snd_pcm_uframes_t silence_size;      /* Silence filling size */
        snd_pcm_uframes_t boundary;/* pointers wrap point */
        snd_pcm_uframes_t silence_start; /* starting pointer to silence area */
        snd_pcm_uframes_t silence_filled; /* size filled with silence */
        union snd_pcm_sync_id sync;   /* hardware synchronization ID */

        /* -- mmap -- */
        struct snd_pcm_mmap_status *status;
        struct snd_pcm_mmap_control *control;

        /* -- locking / scheduling -- */
        snd_pcm_uframes_t twake;      /* do transfer (!poll) wakeup if non-zero */
        wait_queue_head_t sleep;      /* poll sleep */
        wait_queue_head_t tsleep;     /* transfer sleep */
        struct fasync_struct *fasync;

        /* -- private section -- */
        void *private_data;
        void (*private_free)(struct snd_pcm_runtime *runtime);

        /* -- hardware description -- */
        struct snd_pcm_hardware hw;
        struct snd_pcm_hw_constraints hw_constraints;

        /* -- timer -- */
        unsigned int timer_resolution;  /* timer resolution */
        int tstamp_type;              /* timestamp type */

        /* -- DMA -- */
        unsigned char *dma_area;      /* DMA area */
        dma_addr_t dma_addr;          /* physical bus address (not accessible from main CPU) */
        size_t dma_bytes;             /* size of DMA area */
        struct snd_dma_buffer *dma_buffer_p; /* allocated buffer */

        /* -- audio timestamp config -- */
        struct snd_pcm_audio_tstamp_config audio_tstamp_config;
        struct snd_pcm_audio_tstamp_report audio_tstamp_report;
        struct timespec driver_tstamp;

#if IS_ENABLED(CONFIG_SND_PCM_OSS)
        /* -- OSS things -- */
        struct snd_pcm_oss_runtime oss;
#endif
};
```

Related parameters such as audio format and sampling rate (their meaning can be understood in conjunction with the "Name Explanation" section) :

- unsigned int **sample_bits** : Representative sampling depth , e.g. 8bit , 16 'bit and the like .
- snd_pcm_format_t **format** : audio format , snd_pcm_format_t (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L191) defines all the formats supported by the kernel . Generally, the user space does not directly set the sampling depth , but sets the audio format . For the conversion relationship between audio format and sampling depth, see pcm_formats (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_misc.c#L45) [] , pcm_formats[x]. phys It represents the sampling depth corresponding to a certain format . (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L191) (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_misc.c#L45)

- snd_pcm_subformat_t subformat : This parameter is temporarily useless .
- unsigned int **channels** : the number of channels , e.g. mono, stereo, 5.1 -channel and the like .
- unsigned int **frame_bits** : How many bits a frame of data occupies , the calculation formula is channels * sample_bits .
- snd_pcm_uframes_t **period_size** : represents the number of frames in a period .
- unsigned int **rate** : represents the sampling frequency .
- snd_pcm_uframes_t **buffer_size** : Representative a Buffer frames in . Piece Buffer by the kernel distribution , can be DMA access , for exchanging audio data with the user space .
- unsigned int **periods** : on behalf of Buffer within the periods number , this parameter buffer_size meaning somewhat repetitive , Talia interchangeable , buffer_size = periods * period_size.
- snd_pcm_sframes_t **delay** : from the point of view code comments , seemingly on behalf of I2S the FIFO size , in units of frames . Named it the delay reason is that if we know the I2S FIFO size , knowing that the sampling frequency , then we can calculate the The time required for I2S to transmit this piece of data .
- snd_pcm_uframes_t **min_align** ; size_t **byte_align** : These two parameters are related to alignment , one is in the unit of frame , and the other is in the unit of byte .

  This alignment means that the piece allocated by the kernel Buffer to be aligned to this border , seen from the foregoing , this kernel assigned Buffer must be the frame of an integer multiple of , not very 'neat' yet , what good alignment ? That's right , if sample_bits is an integer multiple of 8 , the allocated Buffer is not only an integer multiple of the frame , but also can be aligned to the byte boundary , there is nothing to adjust at this time . But if sample_bits is strange , it is not an integer multiple of 8 , For example , if it is 14 bit, the Buffer allocated by the kernel isIt is an integer multiple of the frame , but it may not be aligned to the byte boundary , so the irregular buffer is not easy to manage .

  Therefore, there is a core logic to calculate the min_align and byte_align, code snd_pcm_hw_params (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L669) are , summarized as follows :

```
bits = snd_pcm_format_physical_width(runtime->format);
runtime->sample_bits = bits;
bits *= runtime->channels;
runtime->frame_bits = bits;
frames = 1;
while (bits % 8 != 0) {
        bits *= 2;
        frames *= 2;
}
runtime->byte_align = bits / 8;
runtime->min_align = frames;
```

- Finally , the kernel provides several functions related to the above format , briefly list :

bytes_to_samples (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L686) : The number of bytes is converted to the number of samples .

bytes_to_frames (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L696) : The number of bytes is converted to the number of frames .

samples_to_bytes (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L706) : The number of bytes in sample s to transfer .

frames_to_bytes (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L716) : The number of frames to bytes .

snd_pcm_lib_buffer_bytes (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L735) : Get the number of bytes of the buffer allocated by a substream .

snd_pcm_lib_period_bytes (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L745) : Get the number of bytes in a period of a substream .

Parameters related to ring buffer management :

- unsigned char *dma_area
- dma_addr_t dma_addr
- size_t dma_bytes
- struct snd_dma_buffer *dma_buffer_p

Is used to describe these parameters allocated by the kernel for exchanging audio data piece Buffer. Piece Buffer possible substream has a good distribution initialization process , stored in snd_pcm_substream -> dma_buffer in , referred to as pre-allocated (e.g. atmel_pdc (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/atmel/atmel-pcm-pdc.c#L50) have pre allocated) , then just put this B uffer assigned to snd_pcm_rumtime of these parameters can be . If you do not pre-allocate, then substream assigned piece during operation Buffer, and filling these Field . For the code details of this logic, see snd_pcm_lib_malloc_pages (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_memory.c#L328) .

- snd_pcm_uframes_t **hw_ptr_base** : See the explanation of it in the section `` PCM Ring Buffer Management" .
- str uct snd_pcm_mmap_status (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L476) *status: status-> hw_ptr represents the read (playback) / write (capture) position of the hardware DMA . (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L476)
- struct snd_pcm_mmap_control (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L485) *control : control-> appl_ptr represents the write (playback) / read (capture) position of the user space . (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L485)
- snd_pcm_uframes_t **hw_ptr_interrupt** / unsigned Long **hw_ptr_jiffies** / unsigned Long **hw_ptr_buffer_jiffies** / u64 **hw_ptr_wrap** : these specific meaning temporarily I do not know , but should hw_ptr related .
- snd_pcm_uframes_t **boundary** : See the explanation of it in the section `` PCM Ring Buffer Management" .

Management ring buffer data structure has a few , that buffer read / write pointer is when to update it ? We have to Playback , for example , take a look at the write pointer appl_ptr and read pointer hw_ptr update timing :

- The first update timing of appl_ptr is when the user space calls the write function to write data to the buffer , the kernel layer snd_pcm_write -> snd_pcm_lib_write -> __snd_pcm_lib_xfer (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L2116) function will calculate the new position of appl_ptr , and finally call pcm_lib_apply_appl_ptr to (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L2091) update the parameter . (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L2116) (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L2091)
- The second update timing of appl_ptr is when user space writes data to the buffer through mmap . In mmap mode , the kernel does not know when the user space has finished writing , so when the user space has finished writing need some way inform the kernel a. LSA provides the ioctl SNDRV_PCM_IOCTL_SYNC_PTR , for the user to inform the kernel space updates appl_ptr, e.g. tinyalsa the pcm_sync_ptr (http://androidxref.com/9.0.0_r3/xref/external/tinyalsa/pcm.c#pcm_sync_ptr) is used in this way . in the core layer , snd_pcm_common_ioctl -> snd_pcm_sync_ptr -> pcm_lib_apply_appl_ptr (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L2091) will eventually update the Parameters . (http://androidxref.com/9.0.0_r3/xref/external/tinyalsa/pcm.c#pcm_sync_ptr) (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L2091)
- The update timing of hw_ptr is when the DMA moves the data of a period , a hardware interrupt will be generated at this time , and snd_pcm_period_elapsed -> snd_pcm_update_hw_ptr0 (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L262) will be called in the interrupt processing function to update hw_ptr. (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L262)

  In addition to external hardware interrupt , the user space via the ioctl SNDRV_PCM_IOCTL_HWSYNC / SNDRV_PCM_IOCTL_REWIND / SNDRV_PCM_IOCTL_FORWARD or in the ioctl SNDRV_PCM_IOCTL_SYNC_PTR designated flag is SNDRV_PCM_SYNC_PTR_HWSYNC to inform the kernel update hw_ptr (the ioctl -> ... -> do_pcm_hwsync

(https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L2578) -> ... -> Snd_pcm_update_hw_ptr0). INL also The hw_ptr can be updated by actively calling snd_pcm_update_hw_ptr (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L472) ->... -> snd_pcm_update_hw_ptr0 . In either case , hw_ptr is calculated and updated in snd_pcm_update_hw_ptr0 (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L262) in the end.

Finally , the kernel also provides several functions to obtain free space in the buffer :

snd_pcm_playback_avail (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L757) : How much space can be written in the user space in playback mode .

snd_pcm_capture_avail (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L773) : How much data can be read by user space in capture mode .

snd_pcm_playback_hw_avail (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L785) : How much data can be moved by DMA in playback mode .

snd_pcm_capture_hw_avail (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L794) : How much space can be written by DMA in capture mode .

Parameters related to overrun/underrun events :

Take the playback mode as an example to illustrate their meaning .

- snd_pcm_uframes_t **start_threshold** : When the data in the buffer exceeds this value , the hardware starts data transmission . If it is too large , the delay from the start of the playback to the sound is too long , and it may even cause too short sounds to be played out at all ; if it is too small , It may easily lead to X run .
- snd_pcm_uframes_t **stop_threshold** : When the free area of the buffer is greater than this value , the hardware stops transmitting . By default , this number is the size of the entire buffer , that is , when the entire buffer is empty , the transmission stops . But occasionally, the buffer is empty . such as CPU busy , increase the value , continue to play historical data buffer , without closing the restart transmission hardware (usually this time a clear voice Caton) , you can achieve a better experience .

  When stop_threshold when the whole is greater than the buffer size , when DMA when finished moving all valid data , read and write pointers overlap . If you do not stop the DMA, continues to move , it means that some of the old data will be moved after the write pointer , this time we You will hear some old sounds . If you don't want to hear the old sounds , we can clear the old data after the pointer is written . This is the so-called silence mode .
- snd_pcm_uframes_t **silence_threshold** : When the data that can be moved by the DMA is less than this value, it means that the move is about to be finished , and the kernel starts to clear the old data .
- snd_pcm_uframes_t **silence_size** : Every action is cleared up to how much space will be cleared , if it is set to 0, the kernel will not do it cleared the action , that is silence mode is disabled .
- snd_pcm_uframes_t **silence_start** : From which position to clear zero .
- snd_pcm_uframes_t **silence_filled** : How much space has been cleared .

  The snd_pcm_playback_silence (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L60) function in the kernel is responsible for the clearing action . Read the code for more details .

Some other more important parameters :

- unsigned int info : and SNDRV_PCM_INFO_XXX (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L278) related , seemingly is used to indicate some of the characteristics of the underlying hardware . Initialization parameters for this by the underlying hardware . (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L278)
- snd_pcm_access_t **Access** : snd_pcm_access_t (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L183) defines the user space access the kernel assigned piece for the exchange of audio data Buffer ways , including :

(https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L183)

- SNDRV_PCM_ACCESS_MMAP_INTERLEAVED / SNDRV_PCM_ACCESS_MMAP_NONINTERLEAVED / SNDRV_PCM_ACCESS_MMAP_COMPLEX :

  mmap means to map the Buffer to the user space and then access it .

  Interleaved and non interleaved refer to the way the channels in the buffer are discharged . Take dual channels as an example : interleaved refers to the alternate emission of left and right channels , and non interleaved refers to putting all the left channels together first, and then putting all the channels together. The right channels are arranged together .

- SNDRV_PCM_ACCESS_RW_INTERLEAVED / SNDRV_PCM_ACCESS_RW_NONINTERLEAVED : RW behalf of the user space with write / read access mode B uffer. Interleaved and non Interleaved same meaning as above .

- struct snd_pcm_hardware hw : See " struct snd_pcm_hardware " for details .
- struct snd_pcm_hw_constraints hw_constraints : See `` struct snd_pcm_hw_constraints " for details .

struct snd_pcm_ops

struct snd_pcm_ops (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L66) : The interface functions defined by the PCM middle layer that need to be implemented by the bottom driver , which is equivalent to interface. The middle layer will call back these interface functions when appropriate . From the perspective of the bottom driver , most of the work is to implement this ops definition function (usually only partially realized , other intermediate layer has a default implementation . unless the middle achieve layer does not take on their own hardware , it needs to realize ourselves .) , then the PCM middle layer 'registered' to .

```
struct snd_pcm_ops {
        int (*open)(struct snd_pcm_substream *substream);
        int (*close)(struct snd_pcm_substream *substream);
        int (*ioctl)(struct snd_pcm_substream * substream,
                     unsigned int cmd, void *arg);
        int (*hw_params)(struct snd_pcm_substream *substream,
                         struct snd_pcm_hw_params *params);
        int (*hw_free)(struct snd_pcm_substream *substream);
        int (*prepare)(struct snd_pcm_substream *substream);
        int (*trigger)(struct snd_pcm_substream *substream, int cmd);
        snd_pcm_uframes_t (*pointer)(struct snd_pcm_substream *substream);
        int (*get_time_info)(struct snd_pcm_substream *substream,
                        struct timespec *system_ts, struct timespec *audio_ts,
                        struct snd_pcm_audio_tstamp_config *audio_tstamp_config,
                        struct snd_pcm_audio_tstamp_report *audio_tstamp_report);
        int (*fill_silence)(struct snd_pcm_substream *substream, int channel,
                            unsigned long pos, unsigned long bytes);
        int (*copy_user)(struct snd_pcm_substream *substream, int channel,
                         unsigned long pos, void __user *buf,
                         unsigned long bytes);
        int (*copy_kernel)(struct snd_pcm_substream *substream, int channel,
                           unsigned long pos, void *buf, unsigned long bytes);
        struct page *(*page)(struct snd_pcm_substream *substream,
                             unsigned long offset);
        int (*mmap)(struct snd_pcm_substream *substream, struct vm_area_struct *vma);
        int (*ack)(struct snd_pcm_substream *substream);
};
```

- **open** : When a user space open a substream time , calls back here open function , and the substream as a parameter passed in .
- **close** : Same as above
- **ioctl** : Same as above
- **hw_params** : In the "name resolution" We introduced a number of parameters , these parameters are called kernel hw params. Users can configure the space of these parameters , such as setting the audio format, channel number, period size . When the user space When configuring these parameters , the hw_params function here will be called back . The parameter snd_pcm_hw_params of this function is equivalent to a parameter set , including the parameters that need to be set . For details, see the introduction of the structure in the section " hw params description " .

- **hw_free** : As mentioned earlier, substream will prepare a buffer for users to exchange audio data . When hw_free is called back , we need to release this buffer in this function .

  Since there are released , there should be allocated . Substream is a piece of B uffer is allocated where it ? Low-level driver in open or hw_params callback function , will call snd_pcm_lib_malloc_pages (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_memory.c#L328) , this function checks for pre_allocated Buffer, if used directly It ; otherwise , the function snd_dma_alloc_pages (https://elixir.bootlin.com/linux/v4.20/source/sound/core/memalloc.c#L178) provided by the PCM middle layer will be called to allocate Buffer. (https://elixir.bootlin.com/linux/v4.20/source/sound/core/memalloc.c#L178)
- **Prepa Re** : When data is written to the user space Buffer later , informs the bottom do prepare operation , this time is generally DMA related configuration , such that it is in a standby state .
- **trigger** : used to start or suspend DMA transmission . The parameter cmd indicates whether to start or stop .
- **pointer** : Get the location of the Buffer where the DMA is moving data at the moment . It can be understood in conjunction with the section " PCM ring buffer management" .
- **get_time_info** : I don't know for now .
- **fill_silence** : When an X run event occurs , we can clear the old data , so that the user **hears'silence** ' . This function performs a specific clearing action .
- **copy_user** : when a user space write audio Buffer time , will call back function . In the function which usually use copy_from_user / copy_to_user.
- **copy_kernel** : When the kernel space to read and write audio Buffer when , will the callback function . Use this function inside memcpy can .
- **page :** Given a virtual mechanism , return the physical page corresponding to the address . The parameter offs et refers to the offset in the audio buffer , substream->runtime->dma_area + offset can get a virtual address . Generally not implemented this function , even though the real current with the intermediate layer is provided snd_pcm_lib_get_vmalloc_page (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_memory.c#L449) . (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_memory.c#L449)
- **mmap** : The user space can map the audio buffer to the user space first , and then access it . When the user space executes the map operation , the function will be called back . Because the Buffer has generally been allocated , the function only needs to modify the page table to establish the mapping That's it , and generally don't need to be implemented .
- **ACK** : When the PCM intermediate layer of the ring buffer pointer is updated appl_ptr time , will the callback function , only if it is greater than the range of values is equal to 0 when , updating operation was successful . E.g. pcm_lib_apply_appl_ptr (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L2091) inside to the callback function . (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L2091)

hw params description

We "explains the name" describes a lot of parameters , these parameters are called hw params. In the struct snd_pcm_runtime in , there are definitions of these parameters :

```
/* -- HW params -- */
snd_pcm_access_t access;        /* access mode */
snd_pcm_format_t format;        /* SNDRV_PCM_FORMAT_* */
snd_pcm_subformat_t subformat;  /* subformat */
unsigned int rate;              /* rate in Hz */
unsigned int channels;          /* channels */
snd_pcm_uframes_t period_size;  /* period size */
unsigned int periods;           /* periods */
snd_pcm_uframes_t buffer_size;  /* buffer size */
snd_pcm_uframes_t min_align;    /* Min alignment for the format */
size_t byte_align;
unsigned int frame_bits;
unsigned int sample_bits;
unsigned int info;
unsigned int rate_num;
unsigned int rate_den;
unsigned int no_period_wakeup: 1;
```

The PCM middle layer provides a mechanism that is not well understood to describe these parameters . Let's give a basic introduction to this mechanism .

First of all , the kernel defines snd_pcm_hw_param_t (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L345) to refer to these parameters . For example, SNDRV_PCM_HW_PARAM_ACCESS represents the access parameter , and SNDRV_PCM_HW_PARAM_SAMPLE_BITS represents the sample_bits parameter . It can be understood as the subscript of the array . Different subscripts represent different parameters , and the parameter values are stored in the array. .

It seems to be easy to understand , what's weird ? The trouble is in this'array' . If all the parameters are of int type , then we can use an int params[] array , which is simple . But here is the parameter type However , ordinary arrays cannot meet the requirements , and only one structure array can be designed . This structure array is struct snd_pcm_hw_params , which contains all the hw params.

To understand this structure array , we must first understand the types of parameters . The kernel divides the parameters into two types :

- The first is the mask type . The so-called mask type that is a bit represents one sense , more bit may or phase , for example Formats = SNDRV_PCM_FMTBIT_U8 | SNDRV_PCM_FMTBIT_S16_LE , see " struct snd_mask " . Kernel definition of access, format, subformat three This parameter is of type mask .

```
#define SNDRV_PCM_HW_PARAM_ACCESS       0       /* Access type */
#define SNDRV_PCM_HW_PARAM_FORMAT       1       /* Format */
#define SNDRV_PCM_HW_PARAM_SUBFORMAT    2       /* Subformat */
#define SNDRV_PCM_HW_PARAM_FIRST_MASK   SNDRV_PCM_HW_PARAM_ACCESS
#define SNDRV_PCM_HW_PARAM_LAST_MASK    SNDRV_PCM_HW_PARAM_SUBFORMAT
```

- The second type is of the interval type . This type is a bit strange , it may be an integer , or it may be a range [min, max] . For details, see " struct snd_interval " . The kernel defines that the following parameters are all of the interval type .

```
#define SNDRV_PCM_HW_PARAM_SAMPLE_BITS  8       /* Bits per sample */
#define SNDRV_PCM_HW_PARAM_FRAME_BITS   9       /* Bits per frame */
#define SNDRV_PCM_HW_PARAM_CHANNELS     10      /* Channels */
#define SNDRV_PCM_HW_PARAM_RATE         11      /* Approx rate */
#define SNDRV_PCM_HW_PARAM_PERIOD_TIME  12      /* Approx distance between
                                                 * interrupts in us
                                                 */
#define SNDRV_PCM_HW_PARAM_PERIOD_SIZE  13      /* Approx frames between
                                                 * interrupts
                                                 */
#define SNDRV_PCM_HW_PARAM_PERIOD_BYTES 14      /* Approx bytes between
                                                 * interrupts
                                                 */
#define SNDRV_PCM_HW_PARAM_PERIODS      15      /* Approx interrupts per
                                                 * buffer
                                                 */
#define SNDRV_PCM_HW_PARAM_BUFFER_TIME  16      /* Approx duration of buffer
                                                 * in us
                                                 */
#define SNDRV_PCM_HW_PARAM_BUFFER_SIZE  17      /* Size of buffer in frames */
#define SNDRV_PCM_HW_PARAM_BUFFER_BYTES 18      /* Size of buffer in bytes */
#define SNDRV_PCM_HW_PARAM_TICK_TIME    19      /* Approx tick duration in us */
#define SNDRV_PCM_HW_PARAM_FIRST_INTERVAL       SNDRV_PCM_HW_PARAM_SAMPLE_BITS
#define SNDRV_PCM_HW_PARAM_LAST_INTERVAL        SNDRV_PCM_HW_PARAM_TICK_TIME
```

Kernel with struct snd_mask and struct snd_ interval The describe the two parameters . Their details are as follows .

struct snd_mask

struct snd_mask (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L391) : Describes a parameter of type mask .

```
#define SNDRV_MASK_MAX  256

struct snd_mask {
        __u32 bits[(SNDRV_MASK_MAX+31)/32];
};
```

- ✓ SNDRV_MASK_MAX 256 represents a snd_mask can store up to 256 Ge bit .
- ✓ a u32 can store 32 a bit, so the array __u32 bits [(SNDRV_MASK_MAX + 31) / 32] represents can be stored SNDRV_MASK_MAX a bit.

In fact, does not take more than that , the biggest also only a 53 Ge bit, with u64 can be stored for , I wonder if the kernel is designed to take account of future extensibility . Kernel several mask parameters occupied bit number as follows :

- ➢ Access: #define SNDRV_PCM_ACCESS_LAST SNDRV_PCM_ACCESS_RW_NONINTERLEAVED (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L189) (5 Ge bit) (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L189)
- ➢ the format: #define SNDRV_PCM_FORMAT_LAST SNDRV_PCM_FORMAT_DSD_U32_BE (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L244) (. 5 . 3 th 'bit) (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L244)
- ➢ the SubFormat: #define SNDRV_PCM_SUBFORMAT_LAST SNDRV_PCM_SUBFORMAT_STD (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L276) (1 Ge bit) (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L276)

struct snd_interval

struct snd_interval (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L381) : Describes an interval type parameter .

```
struct snd_interval {
        unsigned int min, max;
        unsigned int openmin:1,
                     openmax:1,
                     integer:1,
                     empty:1;
};
```

- ✓ ♦≈ℳ■ openmin is set , it means that min is valid . At this time , it means what is the minimum value of the parameter .
- ✓ ♦≈ℳ■ openmax is set , it means that max is valid . At this time, it represents the maximum value of the parameter .
- ✓ openmin, OpenMAX are set , the range of the representative parameter is [min, max] .
- ✓ Integer is set , which means that the parameter is plastic , and min == max at this time .

struct snd_pcm_hw _ params

struct snd_pcm_hw_params (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L395) : for describing a hardware parameter set . When the user wants to modify a space hw params time , ready for the set of parameters , then it can be provided to the kernel . Conversely , when the user wants to obtain the spatial parameter set , it requests the kernel Just return the parameter set .

```
struct snd_pcm_hw_params {
        unsigned int flags;
        struct snd_mask masks[SNDRV_PCM_HW_PARAM_LAST_MASK -
                              SNDRV_PCM_HW_PARAM_FIRST_MASK + 1];
        struct snd_mask mres[5];        /* reserved masks */
        struct snd_interval intervals[SNDRV_PCM_HW_PARAM_LAST_INTERVAL -
                                      SNDRV_PCM_HW_PARAM_FIRST_INTERVAL + 1];
        struct snd_interval ires[9];    /* reserved intervals */
        unsigned int rmask;             /* W: requested masks */
        unsigned int cmask;             /* R: changed masks */
        unsigned int info;              /* R: Info flags for returned setup */
        unsigned int msbits;            /* R: used most significant bits */
        unsigned int rate_num;          /* R: rate numerator */
        unsigned int rate_den;          /* R: rate denominator */
        snd_pcm_uframes_t fifo_size;    /* R: chip FIFO size in frames */
        unsigned char reserved[64];     /* reserved for future */
};
```

- ✓ flags: You can set a number of condition flags , and snd_pcm_hw_constraints relevant , detailed later .
- ✓ **masks** []: struct snd_mask type array with a size of 3. It is used to describe the parameters of access , format , and subformat .
- ✓ **intervals[]** : struct snd _interval type array , used to describe all interval parameters defined by the kernel .
- ✓ **rmask** : Set by the user space . When the user space wants to modify the hw params , it does not necessarily modify all the parameters . This flag indicates which parameters the user space wants to modify , such as SNDRV_PCM_HW_PARAM_ACCESS | SNDRV_PCM_HW_PARAM_SAMPLE_BITS means that you want to modify the two parameters of access and sample_bits The kernel will detect the rmask, only care about the parameters that need to be modified, and ignore other parameters .
- ✓ **cmask** : Set by the kernel space . When the user space tells the kernel to modify some parameters through rmask , the kernel space will set the corresponding bit in the cmask after the modification is successful . If the corresponding bit is not set , it means that the parameter modification was not successful .
- ✓ **info** / msbits / rate_num / rate_den / fifo_size: These parameters are set by the kernel space . When the user space requests the kernel to return the snd_pcm_hw_params parameter set , the kernel will set these parameters and then return .

By the way , the user space uses ioctl SNDRV_PCM_IOCTL_HW_PARAMS (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L2874) to set / get this parameter set .

struct snd_pcm_hardware

struct snd_pcm_hardware (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L45) : its meaning and SND _pcm_hw_params very similar , are used to describe various hw params. except that it is used to describe the underlying hardware capabilities which have , for example, which supports the format, which uses frequency . It is generally Defined in the underlying driver , the engineer will fill in the structure after consulting the relevant hardware manual .

```
struct snd_pcm_hardware {
        unsigned int info;              /* SNDRV_PCM_INFO_* */
        u64 formats;                    /* SNDRV_PCM_FMTBIT_* */
        unsigned int rates;             /* SNDRV_PCM_RATE_* */
        unsigned int rate_min;          /* min rate */
        unsigned int rate_max;          /* max rate */
        unsigned int channels_min;      /* min channels */
        unsigned int channels_max;      /* max channels */
        size_t buffer_bytes_max;        /* max buffer size */
        size_t period_bytes_min;        /* min period size */
        size_t period_bytes_max;        /* max period size */
        unsigned int periods_min;       /* min # of periods */
        unsigned int periods_max;       /* max # of periods */
        size_t fifo_size;               /* fifo size in bytes */
};
```

The meaning of each field is very clear , so I wo n't repeat it .

s w params description

When an Xrun event occurs , we need some parameters to control the behavior of the kernel (whether to continue playing or stop playing , if it continues , play old sounds or mute data) , these parameters are called sw params.
In struct snd_pcm_runtime in , we have to sw params definition :

```
/* -- SW params -- */
int tstamp_mode;                /* mmap timestamp is updated */
unsigned int period_step;
snd_pcm_uframes_t start_threshold;
snd_pcm_uframes_t stop_threshold;
snd_pcm_uframes_t silence_threshold; /* Silence filling happens when
                                        noise is nearest than this */
snd_pcm_uframes_t silence_size; /* Silence filling size */
snd_pcm_uframes_t boundary;     /* pointers wrap point */

snd_pcm_uframes_t silence_start; /* starting pointer to silence area */
snd_pcm_uframes_t silence_filled; /* size filled with silence */

union snd_pcm_sync_id sync;     /* hardware synchronization ID */
```

struct snd_pcm_sw_params

struct snd_pcm_sw_params (https://elixir.bootlin.com/linux/v4.20/source/include/uapi/sound/asound.h#L419) : used to describe a set of software parameters . When the user wants to modify a space sw params time , ready for the set of parameters , then it can be provided to the kernel . Conversely , when the user wants to obtain the spatial parameter set , it requests the kernel Just return the parameter set .

```
struct snd_pcm_sw_params {
        int tstamp_mode;                        /* timestamp mode */
        unsigned int period_step;
        unsigned int sleep_min;                 /* min ticks to sleep */
        snd_pcm_uframes_t avail_min;            /* min avail frames for wakeup */
        snd_pcm_uframes_t xfer_align;           /* obsolete: xfer size need to be a multiple */
        snd_pcm_uframes_t start_threshold;      /* min hw_avail frames for automatic start */
        snd_pcm_uframes_t stop_threshold;       /* min avail frames for automatic stop */
        snd_pcm_uframes_t silence_threshold;    /* min distance from noise for silence filling */
        snd_pcm_uframes_t silence_size;         /* silence block size */
        snd_pcm_uframes_t boundary;             /* pointers wrap point */
        unsigned int proto;                     /* protocol version */
        unsigned int tstamp_type;               /* timestamp type (req. proto >= 2.0.12) */
        unsigned char reserved[56];             /* reserved for future */
};
```

The meaning of the parameters in the " struct snd_pcm_runtime " a have described them , not repeat them here .

Incidentally , the user space via the ioctl SNDRV_PCM_IOCTL_S_W_PARAMS (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L2878) set / on the parameters set .

hw params constraints

Constraints mean that we can set some rules in the kernel , each rule has a check function , and the rule can specify which parameters we care about . Then all the rules are stringed together in a linked list .
When the user space sets a parameter , the kernel will traverse all the rules. If it finds that the parameter set that a rule cares about contains currently being modified , it will call the check function of the rule . Only when the check function passes , this parameter allows you to modify it . this is called constraints (restrictions) sense .

The kernel uses struct snd_pcm_hw_rule to describe a rule, and uses struct snd_pcm_hw_constraints to concatenate all the rules.

struct snd_pcm_hw_rule

struct snd_pcm_hw_rule (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L236) : represents a rule.

```
struct snd_pcm_hw_rule {
        unsigned int cond;
        int var;
        int deps[4];

        snd_pcm_hw_rule_func_t func;
        void *private;
};
```

- ✓ **CO Nd** : set some conditions . If a rule is set up cond, then only when snd_pcm_ pcm_hw_params.flags also set the same cond time , will be rule.func check . Otherwise, ignore this rule. Code refer to this judgment (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L402) . (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L402)
- ✓ **deps [4]** : on behalf of the rule set of parameters of interest , when users modify any parameter space parameter set , will trigger this rule is executed . Looks like the biggest concern only 5 parameters ?
- ✓ **FUNC** : check function , checking function , general updates **var** parameter points . Meaning that , if you want to modify the space d [] eps when a parameter , the kernel space must first see if we can modify var pointed parameters , if not , it also does not allow users to modify space .

struct snd_pcm_hw_constraints

struct snd_pcm_hw_constraints (https://elixir.bootlin.com/linux/v4.20/source/include/sound/pcm.h#L245) : equivalent to a pool , concatenating all the rules.

```
struct snd_pcm_hw_constraints {
        struct snd_mask masks[SNDRV_PCM_HW_PARAM_LAST_MASK -
                              SNDRV_PCM_HW_PARAM_FIRST_MASK + 1];
        struct snd_interval intervals[SNDRV_PCM_HW_PARAM_LAST_INTERVAL -
                                      SNDRV_PCM_HW_PARAM_FIRST_INTERVAL + 1];
        unsigned int rules_num;
        unsigned int rules_all;
        struct snd_pcm_hw_rule *rules;
};
```

- ✓ *rules: Linked list , concatenating all rules.
- ✓ rules_num: How many rules are actually in the pool .
- ✓ rules_all: The pool has a capacity of how much , when the pool is not enough capacity , the system will automatically expansion . Every expansion of 16 Ge rule space .

Add and check rules of the time

Add rules of the time :

- ✓ snd_pcm_open_substream -> snd_pcm_hw_constraints_init (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L2164) -> snd_pcm_hw_rule_add adds a lot of system default rules. (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L2164)
- ✓ In addition , when necessary , we can also call snd_pcm_hw_rule_add to (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L1123) add some custom rules. (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L1123)

Check the rules of the time :

- ✓ ioctl -> ... -> snd_pcm_hw_refine -> constrain_params_by_rules (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L350) will scan and check all rules (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L350)

1.5.3.3 API

Only the API provided by the P CM middle layer for the underlying driver is introduced here .

ALSA kernel code, there are many useful tools function , for example, ring buffer assigned snd_pcm_lib_malloc_pages, acquiring the buffer free space snd_pcm_playback_avail the like , we have in the "Data structure" described in section , herein being not repeat . Follow-up If you have any questions, you can put these functions into this section .

snd_pcm_new

snd_pcm_new (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm.c#L835) is defined in source/sound/core/pcm.c .

```
/**
 * snd_pcm_new - create a new PCM instance
 * @card: the card instance
 * @id: the id string
 * @device: the device index (zero based)
 * @playback_count: the number of substreams for playback
 * @capture_count: the number of substreams for capture
 * @rpcm: the pointer to store the new pcm instance
 *
 * Creates a new PCM instance.
 *
 * The pcm operators have to be set afterwards to the new instance
 * via snd_pcm_set_ops().
 *
 * Return: Zero if successful, or a negative error code on failure.
 */
int snd_pcm_new(struct snd_card *card, const char *id, int device,
                int playback_count, int capture_count, struct snd_pcm **rpcm)
{
        return _snd_pcm_new(card, id, device, playback_count, capture_count,
                        false, rpcm);
}
EXPORT_SYMBOL(snd_pcm_new);
```

The main function of this function is to create a P CM instance , including :

- ✓ Construct a struct snd_pcm data structure to represent a PCM instance
  - calling snd_pcm_new_stream (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm.c#L796) (pcm, SNDRV_PCM_STREAM_PLAYBACK, playback_count) to build playback stream, and create playback_count a substream. (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm.c#L796)
  - calling snd_pcm_new_stream (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm.c#L801) (pcm, SNDRV_PCM_STREAM_CAPTURE, capture_count) to build capture stream, and create capture_count a substream. (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm.c#L801)
- ✓ Finally , call snd_device_new to add this instance as a logical device to the card->devices linked list .

Please note that before the card is registered , we need to call snd_pcm_set_ops to set the callback function for this PCM instance , because when the user space interacts with the PCM middle layer through the device node , the PCM middle layer needs to call back the ops function implemented by the underlying driver .

snd_pcm_set_ops

snd_pcm_set_ops (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_lib.c#L485) is defined in source/sound/core/pcm_lib.c .

```c
/**
 * snd_pcm_set_ops - set the PCM operators
 * @pcm: the pcm instance
 * @direction: stream direction, SNDRV_PCM_STREAM_XXX
 * @ops: the operator table
 *
 * Sets the given PCM operators to the pcm instance.
 */
void snd_pcm_set_ops(struct snd_pcm *pcm, int direction,
                     const struct snd_pcm_ops *ops)
{
        struct snd_pcm_str *stream = &pcm->streams[direction];
        struct snd_pcm_substream *substream;

        for (substream = stream->substream; substream != NULL; substream = substream->next)
                substream->ops = ops;
}
```

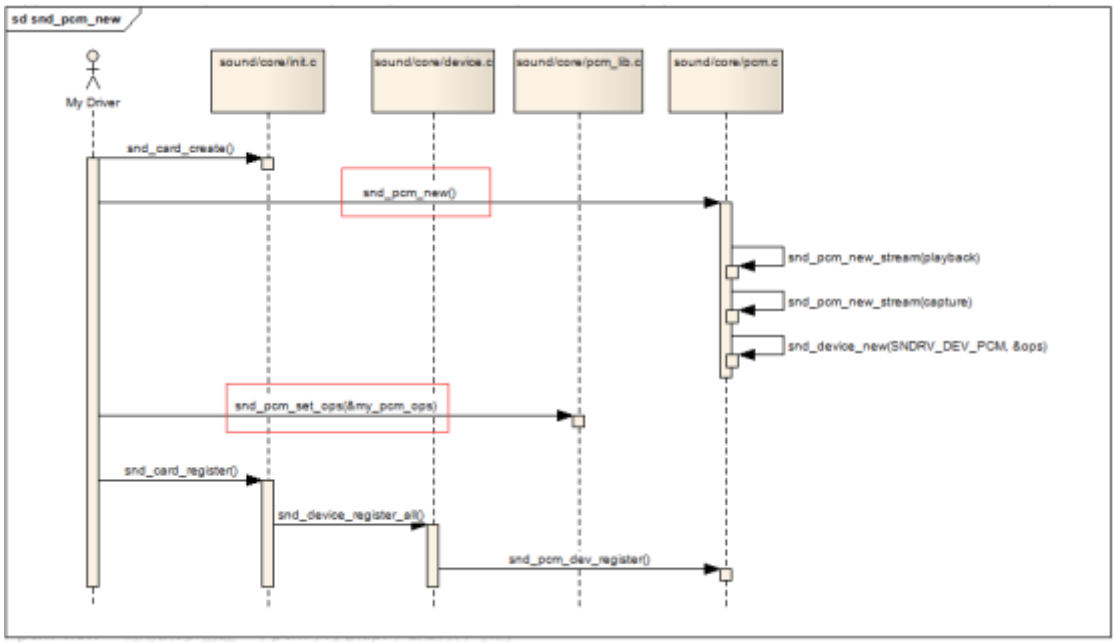
The underlying drivers achieve better struct snd_ pcm_ops after , calling this API are playback stream and capture stream to set a callback function .

1.5.3.4 The bottom driver registers with the PCM middle layer

The main work of the bottom driver is to implement the struct snd_pcm_ops structure , and then call the API registration provided by the middle layer .

Regarding how to implement snd_pcm_ops , I won't say much for the time being , you should be able to understand it by looking at the description of the ``Data Structure'' section .

The registration steps are as follows , copy an online (https://blog.csdn.net/droidphone/article/details/6308006) picture :



1.5.3.5 Creation of PCM character device

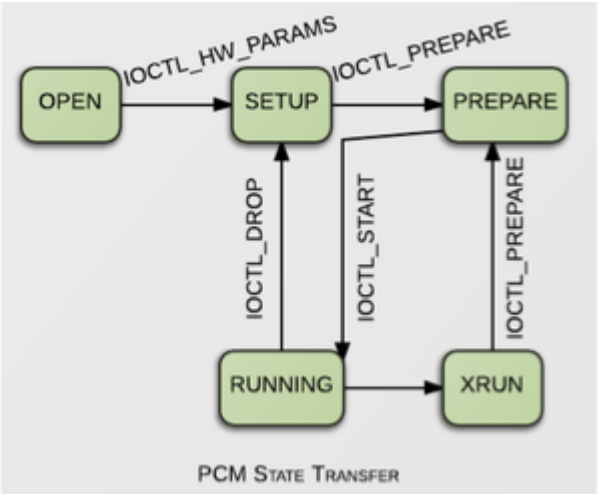
When the card is registered, it will scan each subordinate logical device and register them , and the PCM logical device created here will also be registered at that time . When the PCM logical device is registered , the ALSA system layer will call back the snd_device_ops of the logical device . The dev_register function , which is snd_pcm_dev_register (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm.c#L1100) . In this callback function , snd_register_device is called for each stream , and the corresponding device node is created in the user space .

The snd_pcm_f_ops_of_the (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L3647) P CM middle layer will be responsible for interacting with the user space , and its main functions include : (https://elixir.bootlin.com/linux/v4.20/source/sound/core/pcm_native.c#L3647)

- ✓ open / release: open or close a substream.
- ✓ read / write / mmap: The user space exchanges audio data with the PCM middle layer .
- ✓ ioctl: Provide a variety of control interfaces .

1.5.3.6 State machine of the PCM middle layer

The following figure is a transition diagram of the state of PCM :



In addition XRUN state of the outside , the other by most of the state of the user space ioctl () explicitly switch . For example :

1) The user space is open, and the kernel layer opens a substream, which is in the OPEN state .
2) The user space ioctl (SNDRV_PCM_IOCTL_HW_PARAMS) sets the hw params of runtime , and the substream switches to the SETUP state .
3) the user space ioctl (SNDRV_PCM_IOCTL_PREPARE) after , substream is switched to the PREPARE state . At this time, the representative DMA hardware ready , can begin to move the data .
4) the user space ioctl (SNDRV_PCM_IOCTL_START) after , substream is switched to the RUNNING state . At this time DMA start moving data .
5) the user space ioctl (SNDRV_PCM_IOCTL_DROP) after , substream is switched back to SETUP state .

XRUN state was divided in two : When playing , the user space did not write the data in a timely manner to cause a buffer is empty , the hardware is no data available to play lead UNDERRUN; while recording , the user space is no time to read the data led to the overflow buffer is full , hardware The recorded data has no free buffer to write, resulting in OVERRUN.

1.5.3.7 hrtimer simulates PCM cycle interrupt

In the " struct snd_pcm_runtime " tells the DMA Buffer related time codes , we mentioned that ' when DMA End move a period after the data , this time will generate a hardware interrupt , the interrupt handler calls snd_pcm_period_elapsed -> snd_pcm_update_hw_ptr0 updated hw_ptr ' , And DMA will be called to start the next round of data transmission .

However, some of the p- latform there may be some design flaws , when the DMA after the move is completed will not generate hardware interrupts , so the system how to know when transmitting over a period of data it ? Known the Sample _rate case of , I2S consume a period of data It can be calculated . This time is also the time for DMA to move data . So we can use timers to simulate this hardware interrupt :

- First trigger the DMA to move the data , and start the timer to start timing at the same time .
- When the timing to 1 * period_size / sample_rate , then I2S has completed a transmission the p- eriod of the audio data , enter the timer interrupt processing : call snd_pcm_period_elapsed () inform pcm native data cycle is done with it , while preparing for the next Data handling .

In order to better ensure the real-time performance of data transmission , it is recommended to use a high-precision timer hrtimer . For example, fsl 's imx-pcm-fiq.c implements snd_hrtimer_callback (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/fsl/imx-pcm-fiq.c#L49) .

1.5.3.8 Play PCM audio in user space

In user space , we can use the API (https://www.alsa-project.org/main/index.php/ALSA_Library_API) provided by alsa-lib to play PCM audio . The usage is very simple . The basic step is "open -> set parameters -> read and write audio data" , you can refer to this example (https://blog.csdn.net/liuxizhen2009/article/details/22378201) . (https://www.alsa-project.org/main/index.php/ALSA_Library_API)
(https://blog.csdn.net/liuxizhen2009/article/details/22378201)

In addition , A ndroid has developed a set of tinyalsa to replace alsa-lib. If you want to use tinyalsa to play audio , please refer to tinyplay.c (http://androidxref.com/9.0.0_r3/xref/external/tinyalsa/tinyplay.c)

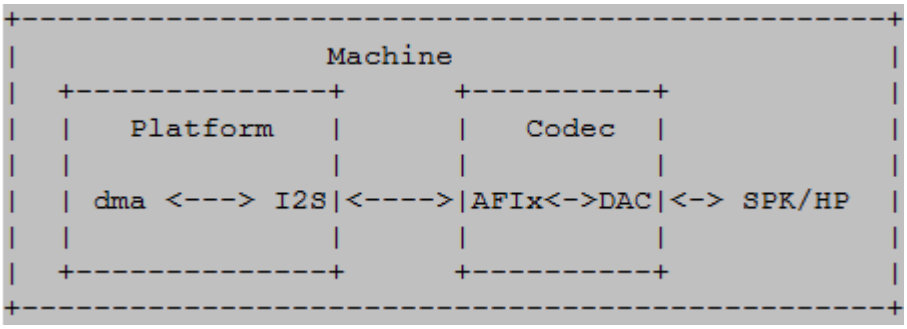
There are currently three known ways of exchanging audio data between the user space and the kernel :

- ✓♆♦♏❒ space directly calls write / read .
- ✓♆♦♏❒ space mmap kernel buffer, then write / read data , after the operation is completed, use ioctl (SNDRV_PCM_IOCTL_SYNC_PTR) to notify the kernel that the read and write is complete .
- ✓ user space directly with the ioctl SNDRV_PCM_IOCTL_WRITEI_FRAMES / SNDRV_PCM_IOCTL_READI_FRAMES / SNDRV_PCM_IOCTL_WRITEN_FRAMES / SNDRV_PCM_IOCTL_READN_FRAMES read and write data . Where I Representative Interleaved , N Representative noninterleaved .

2 AS o C

2.1 Introduction

From a hardware point of view , the typical audio design is : a Codec chip is connected to the I2S interface of a CPU on a circuit board , and the Codec is connected to an external headset or power amplifier . As shown in the following figure :



Take playback as an example . Under such a hardware structure , several modules are involved :

- ✓ DMA: responsible for the user space of the audio data are moved to I2S of the FIFO .
- ✓ I2S: Responsible for sending audio data with a certain sampling frequency, sampling depth, and number of channels , also called dai (Digital Audio Interface) .
- ✓ AFIx: Responsible for receiving audio data at a certain sampling frequency, sampling depth, and number of channels , also known as dai.
- ✓ DAC: The data is converted by DAC and sent to headphones and other playback .

Although we can use the first chapter of ALSA architecture , to achieve a PCM logic device , and controls the number of modules in the code logic device , but doing so will cause poor reusability of code .

To solve the problem of reusability , kernel introduced the AS oC architecture , the architecture is present in the PCM under the intermediate layer , except for the PCM underlying drive achieved were split . From the perspective of the user space is , do not see any difference , The user space still faces the interactive interface provided by the ALSA framework . The following three modules are abstracted in the underlying ASoC :

- ✓ Platform: This module is responsible DMA control and the I2S control , the CPU is responsible for the preparation of vendors in this part of the code .
- ✓ Codec: This module is responsible AFI the X- control and the DAC control section (can also be said to control the functions of the chip itself) , the Codec is responsible for the preparation of vendors in this part of the code .
- ✓ Machine: Used to describe a circuit board , it indicates which Platform and which Codec is used on this circuit board , and the circuit board manufacturer is responsible for writing this part of the code .

From the data structure point is , the ASoC internal core defines the following data structure , note that they are created and maintained by the inner core layer :

- ✓ struct snd_soc_platform (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc.h#L1023) : to abstract a platform, the role is described in a CPU of the DMA apparatus and operation functions . The system may have a plurality of Platforms, which are mounted in the global list head static LIST_HEAD (PLATFORM_LIST (https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/soc-core.c#L59)) below , a different platform to name Distinguish . (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc.h#L1023) (https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/soc-core.c#L59)
- ✓ struct snd_soc_codec (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc.h#L942) : used to abstract a Codec. A Codec may have multiple dai interfaces . The function of this structure is to describe the working logic of C odec that has nothing to do with the specific dai , such as controls / widgets / audio routing description information, clock configuration, the IO control . the system may have a plurality of C odecs, which are mounted in the global list head static LIST_HEAD (codec_list

(https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/soc-core.c#L60)) below , different Codec to name distinguished . (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc.h#L942) (https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/soc-core.c#L60)

- ✓ struct snd_soc_dai (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc-dai.h#L288) : used to describe a dai, either CPU side DAI (the I2S) , may be Codec side DAI (AFIX) . A CPU may have a plurality of I2S, a Codec there may be a plurality of the AFI X , so the system will have a number of dai, which are mounted in the global list head static LIST_HEAD (dai_list (https://elixir.bootlin.com/linux/v3.14.79/source/sound/soc/soc-core.c#L59)) below , different dai to name distinguished . (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc-dai.h#L288) (https://elixir.bootlin.com/linux/v3.14.79/source/sound/soc/soc-core.c#L59)

From the perspective of the underlying driver , AS oC defines some interfaces that need to be implemented at the bottom and the corresponding registration functions :

- ✓ for the DMA (Platform) : the CPU manufacturers need to fill struct snd_soc_platform_driver (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc.h#L1000) and struct snd_pcm_ops (" the PCM logic device" section of the structure are described) , and then calls snd_soc_register_platform (https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/soc-core.c#L3590) to ASoC registration core layer , for example Atmel-PCM-pdc.c (https://elixir.bootlin.com/linux/v3.19.8/source/sound/soc/atmel/atmel-pcm-pdc.c#L325) . (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc.h#L1000) (https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/soc-core.c#L3590) (https://elixir.bootlin.com/linux/v3.19.8/source/sound/soc/atmel/atmel-pcm-pdc.c#L325)
- ✓ for I2S (cpu_dai) : the CPU manufacturers need to fill the struct snd_soc_dai_driver (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc-dai.h#L247) and struct snd_soc_dai_ops (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc-dai.h#L146) , then call snd_soc_register_dai (https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/soc-core.c#L3153) to AS oC registration core layer , for example atmel_ssc_dai.c (https://elixir.bootlin.com/linux/v3.9.11/source/sound/soc/atmel/atmel_ssc_dai.c) . (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc-dai.h#L247) (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc-dai.h#L146) (https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/soc-core.c#L3153) (https://elixir.bootlin.com/linux/v3.9.11/source/sound/soc/atmel/atmel_ssc_dai.c)
- ✓ for Codec (codec_dai) : Codec manufacturers need to fill struct snd_soc_codec_driver (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc.h#L961) (used to describe Codec internal operating logic) and struct snd_soc_dai_driver (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc-dai.h#L247) , struct snd_soc_dai_ops (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc-dai.h#L146) (used to describe the AFI X) , then call snd_soc_register_codec (https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/soc-core.c#L3751) to the AS oC registration core layer , e.g. wm9081.c (https://elixir.bootlin.com/linux/v4.16.18/source/sound/soc/codecs/wm9081.c#L1358) . (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc.h#L961) (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc-dai.h#L247) (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc-dai.h#L146) (https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/soc-core.c#L3751) (https://elixir.bootlin.com/linux/v4.16.18/source/sound/soc/codecs/wm9081.c#L1358)
- ✓ For Machine (CODEC) : circuit board business needs to be filled struct snd_soc_dai_link (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc.h#L1032) , struct snd_soc_ops (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc.h#L781) , then prepare a struct

snd_soc_card (https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc.h#L1155) the dai_link wrapped up , and then call snd_soc_register_card (https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/soc-core.c#L2900) register this snd_soc_card. For example atmel_wm8904.c (https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/atmel/atmel_wm8904.c#L147) .

(https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc.h#L1032)
(https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc.h#L781)
(https://elixir.bootlin.com/linux/v4.17.19/source/include/sound/soc.h#L1155)
(https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/soc-core.c#L2900)
(https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/atmel/atmel_wm8904.c#L147)

When the bottom is called snd_soc_register_card time , ASoC core layer can be found in the global list in dai_link specified Platform , cpu_dai , codec_dai , CODEC, and the establishment of a struct snd_soc_pcm_runtime to save the corresponding relationship . Then core will snd_ card_new create a sound card , snd_pcm_new create pcm Logical device , and finally register the sound card with snd_card_register . After that, the user space can see the device node . When the user space accesses the device node , the ASoC core layer will finally respond , and the core layer will passsnd_soc_pcm_runtime finds the corresponding platform , cpu_dai , codec_dai , codec , and calls back the interface functions implemented by them as needed .

ASoC system architecture also kept the evolution , from v 4.18 start , although ASoC still divided into Platform , cpu_dai , codec_dai , CODEC these modules , but describe their data structure has undergone major changes :

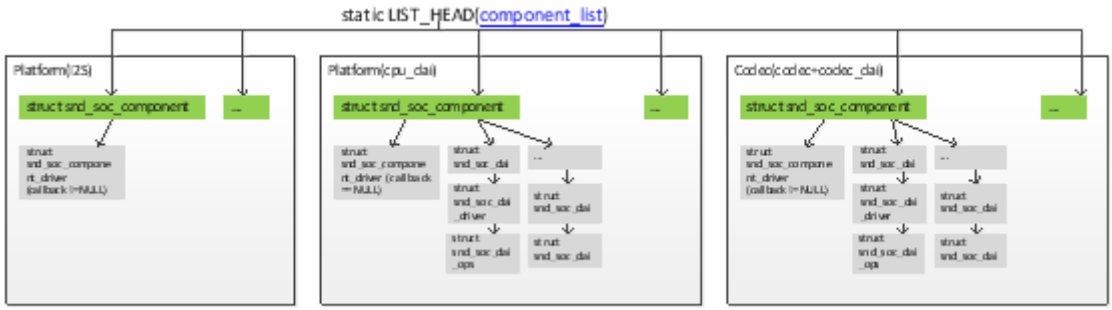
- ✓ First , ASoC unified data structure to describe the platform and CODEC : a core layer with struct snd_soc_component replace the original struct snd_soc_platform (described platform) and struct snd_soc_codec (described CODEC); the underlying driver is a struct snd_soc_component_driver replace the original struct snd_soc_platform_driver and struct snd_soc_codec_driver .
- ✓ Next , description dai data structures, although there is no change , the core layer remains with struct snd_soc_dai , underlying drive still use struct snd_soc_dai_driver , struct snd_soc_dai_ops . But dai associated data structures are packaged into snd_soc_component down unified managed . When the bottom drive register a dai At the time , the core layer will create a snd_soc_component , and then mount dai under this data structure .
- ✓ In other words , whether Platform , CODEC or DAI , the core layer are now unified with struct snd_soc_component to manage , a component corresponding to a module . Thus the original PLATFORM_LIST , codec_list , dai_list these 3 tables heads do not exist , only A static LIST_HEAD (component_list (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-core.c#L54)) header . (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-core.c#L54)
- ✓ Finally , the core layer provides to the underlying registration API also unified , regardless Platform , CODEC or dai, are now unified with devm_snd_soc_register_component / snd_soc_register_component registered to .
- ✓ Finally , there is no change at the Machine level . The machine driver still builds a struct snd_soc_card and then calls snd_soc_register_card to register . And the core layer has not changed much in the implementation logic of the registration function .

In general , the new kernel unifies several data structures and registration functions to make it more refreshing . However, it is inevitable that the new data structure is relatively bloated , because it needs to integrate the needs of multiple modules . In different module drivers, it may only be necessary to fill in the new data structure according to the needs .

This chapter introduces them based on the new data structure . Let's start the journey!

2.2 Data structure

Let's first come to an overall diagram of the data structure :



v4.18 later version , the kernel with the struct snd_soc_component to unify all of the abstract modules , but a different module is slightly different implementation details . For example :

- ✓ ♦ ♦ used to describe the component of Platform (I2S) , which mainly relies on the subordinate snd_soc_component_driver structure, and there is no dai- related structure under this component ;
- ✓ but is used to describe Platform (cpu_dai) of the component , mainly by the subordinate snd_soc_dai , snd_soc_dai_driver , snd_soc_dai_ops structure , snd_soc_component_driver the component generally only a name, other callback functions are not implemented ;
- ✓ used to describe Codec of component is contains snd_soc_component_driver and dai two parts , both of which component status under very important .

The system will have a number of component, they are mounted in the global list head component_list below .

The details of each data structure are introduced below . These data structures can be divided into three categories at a macro level : the data structure used in the core layer of AS oC ; the data structure used in the Platform & Codec driver; the data structure used in the Machine driver .

2.2.1 ASoC core layer

struct snd_soc_component

struct snd_soc_component (https://elixir.bootlin.com/linux/v4.20/source/include/sound/soc.h#L856) : created and maintained by the inner core layer , used to represent a component. When the register driving the bottom Platform , CODEC , DAI time , the core layer creates a corresponding snd_soc_component.

```
struct snd_soc_component {
        const char *name;
        int id;
        const char *name_prefix;
        struct device *dev;
        struct snd_soc_card *card;

        unsigned int active;

        unsigned int ignore_pmdown_time:1; /* pmdown_time is ignored at stop */
        unsigned int registered_as_component:1;
        unsigned int suspended:1; /* is in suspend PM state */

        struct list_head list;
        struct list_head card_aux_list; /* for auxiliary bound components */
        struct list_head card_list;

        const struct snd_soc_component_driver *driver;

        struct list_head dai_list;
        int num_dai;

        int (*read)(struct snd_soc_component *, unsigned int, unsigned int *);
        int (*write)(struct snd_soc_component *, unsigned int, unsigned int);

        struct regmap *regmap;
        int val_bytes;

        struct mutex io_mutex;

        /* attached dynamic objects */
        struct list_head dobj_list;

#ifdef CONFIG_DEBUG_FS
        struct dentry *debugfs_root;
#endif

        /*
        * DO NOT use any of the fields below in drivers, they are temporary and
        * are going to be removed again soon. If you use them in driver code the
        * driver will be marked as BROKEN when these fields are removed.
        */

        /* Don't use these, use snd_soc_component_get_dapm() */
        struct snd_soc_dapm_context dapm;

        struct snd_soc_codec *codec;

        int (*probe)(struct snd_soc_component *);
        void (*remove)(struct snd_soc_component *);
        int (*suspend)(struct snd_soc_component *);
        int (*resume)(struct snd_soc_component *);
        int (*pcm_new)(struct snd_soc_component *, struct snd_soc_pcm_runtime *);
        void (*pcm_free)(struct snd_soc_component *, struct snd_pcm *);

        int (*set_sysclk)(struct snd_soc_component *component,
                          int clk_id, int source, unsigned int freq, int dir);
        int (*set_pll)(struct snd_soc_component *component, int pll_id,
                       int source, unsigned int freq_in, unsigned int freq_out);
        int (*set_jack)(struct snd_soc_component *component,
                        struct snd_soc_jack *jack,  void *data);
        int (*set_bias_level)(struct snd_soc_component *component,
                              enum snd_soc_bias_level level);

        /* machine specific init */
        int (*init)(struct snd_soc_component *component);

#ifdef CONFIG_DEBUG_FS
        void (*init_debugfs)(struct snd_soc_component *component);
        const char *debugfs_prefix;
#endif
};
```

- ✓ list: used to mount yourself under the global linked list component_list .
- ✓ card_list: used to mount the snd_soc_card-> component_dev_list under .
- ✓ driver: points to the subordinate snd_soc_component_driver, this structure is generally implemented by the underlying driver .
- ✓ dai_list : The head of the linked list , which links all the dais of the subordinates
- ✓ num_dai: the number of subordinates dai
- ✓ probe ... set_bias_level: Look at the comments , they will be removed soon .
- ✓ init: It will be called back during the initialization phase , see the `` initialization flowchart " for details .

struct snd_soc_dai

struct snd_soc_dai (https://elixir.bootlin.com/linux/v4.20/source/include/sound/soc-dai.h#L288) : created and maintained by the inner core layer , used to represent a DAI . may be cpu_dai or codec_dai.

```
/*
 * Digital Audio Interface runtime data.
 *
 * Holds runtime data for a DAI.
 */
struct snd_soc_dai {
        const char *name;
        int id;
        struct device *dev;

        /* driver ops */
        struct snd_soc_dai_driver *driver;

        /* DAI runtime info */
        unsigned int capture_active:1;          /* stream is in use */
        unsigned int playback_active:1;         /* stream is in use */
        unsigned int probed:1;

        unsigned int active;

        struct snd_soc_dapm_widget *playback_widget;
        struct snd_soc_dapm_widget *capture_widget;

        /* DAI DMA data */
        void *playback_dma_data;
        void *capture_dma_data;

        /* Symmetry data - only valid if symmetry is being enforced */
        unsigned int rate;
        unsigned int channels;
        unsigned int sample_bits;

        /* parent platform/codec */
        struct snd_soc_codec *codec;
        struct snd_soc_component *component;

        /* CODEC TDM slot masks and params (for fixup) */
        unsigned int tx_mask;
        unsigned int rx_mask;

        struct list_head list;
};
```

- ✓ name: the dai name , A the SoC core layer by name to differentiate dai.
- ✓ id: It should be an id number automatically assigned by the core layer .
- ✓ driver: points to the subordinate snd_soc_dai_driver , this structure is generally implemented by the underlying driver .
- ✓ list: used to mount yourself under component->dai_list .

struct snd_soc_pcm_runtime

struct snd_soc_pcm_ R & lt untime (https://elixir.bootlin.com/linux/v4.20/source/include/sound/soc.h#L1152) : created and maintained by the inner core layer , when Machin E driver calls the snd_soc_register_card after , during initialization , the core layer of the data structure is created , see "Initialization Flow" .

A runtime corresponds to a snd_soc_dai_link, there are several dai_link will create several runtime. All runtime will mount snd_soc_card-> rtd_list below .

In the later stage of initialization , the code (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-core.c#L2067) scans each rtd under rtd_list and calls soc_new_pcm for it to create a PCM logical device . A logical device means that the user space can exchange audio data with it , resulting in audio playback or recording .

runtime This structure is very important in the operation of the system , which corresponds to a transfer station , all operations are required in order to user space forwarded to the proper underlying concrete and is driven by it .

```
/* SoC machine DAI configuration, glues a codec and cpu DAI together */
struct snd_soc_pcm_runtime {
        struct device *dev;
        struct snd_soc_card *card;
        struct snd_soc_dai_link *dai_link;
        struct mutex pcm_mutex;
        enum snd_soc_pcm_subclass pcm_subclass;
        struct snd_pcm_ops ops;

        /* Dynamic PCM BE runtime data */
        struct snd_soc_dpcm_runtime dpcm[2];
        int fe_compr;

        long pmdown_time;

        /* runtime devices */
        struct snd_pcm *pcm;
        struct snd_compr *compr;
        struct snd_soc_codec *codec;
        struct snd_soc_platform *platform; /* will be removed */
        struct snd_soc_dai *codec_dai;
        struct snd_soc_dai *cpu_dai;

        struct snd_soc_dai **codec_dais;
        unsigned int num_codecs;

        struct delayed_work delayed_work;
#ifdef CONFIG_DEBUG_FS
        struct dentry *debugfs_dpcm_root;
#endif

        unsigned int num; /* 0-based and monotonic increasing */
        struct list_head list; /* rtd list of the soc card */
        struct list_head component_list; /* list of connected components */

        /* bit field */
        unsigned int dev_registered:1;
        unsigned int pop_wait:1;
};
```

- ✓ card: the corresponding snd_soc_card.
- ✓ dai_link: The corresponding snd_soc_dai_link.
- ✓ ops: The PCM middle layer needs the snd_pcm_ops we implemented . This field is filled by the AS oC core layer . For details of the code, please refer to the analysis in the section ``When the callback function is called" .
- ✓ CODEC , Platform: intention is to point P latform and C ODEC, but the modules are unified by snd_ soc_component to describe , so this has nothing to with the two fields , the back will be deleted .
- ✓ cpu_dai: point to cpu_dai.
- ✓ codec_dai: The original intention was to point to codec_dai, but now a dai_link can specify multiple codec_dais, so it must be described by an array . This field points to the [0] th element of the array .
- ✓ codec_dais: equivalent to an array , describing all codec_dais.
- ✓ list: used to mount yourself under card- >rtd _list .
- ✓ component_list: head of the list , for all belong to the mount rtd of the component, the component may represent P latform, may also represent C ODEC.

2.2.2 Platfrom & Codec

struct snd_soc_component_driver

struct : The underlying driver needs to fill the structure and then register with the ASoC core layer .

```
struct snd_soc_component_driver {
        const char *name;

        /* Default control and setup, added after probe() is run */
        const struct snd_kcontrol_new *controls;
        unsigned int num_controls;
        const struct snd_soc_dapm_widget *dapm_widgets;
        unsigned int num_dapm_widgets;
        const struct snd_soc_dapm_route *dapm_routes;
        unsigned int num_dapm_routes;

        int (*probe)(struct snd_soc_component *);
        void (*remove)(struct snd_soc_component *);
        int (*suspend)(struct snd_soc_component *);
        int (*resume)(struct snd_soc_component *);

        unsigned int (*read)(struct snd_soc_component *, unsigned int);
        int (*write)(struct snd_soc_component *, unsigned int, unsigned int);

        /* pcm creation and destruction */
        int (*pcm_new)(struct snd_soc_pcm_runtime *);
        void (*pcm_free)(struct snd_pcm *);

        /* component wide operations */
        int (*set_sysclk)(struct snd_soc_component *component,
                          int clk_id, int source, unsigned int freq, int dir);
        int (*set_pll)(struct snd_soc_component *component, int pll_id,
                       int source, unsigned int freq_in, unsigned int freq_out);
        int (*set_jack)(struct snd_soc_component *component,
                        struct snd_soc_jack *jack,  void *data);

        /* DT */
        int (*of_xlate_dai_name)(struct snd_soc_component *component,
                                 struct of_phandle_args *args,
                                 const char **dai_name);
        int (*of_xlate_dai_id)(struct snd_soc_component *comment,
                               struct device_node *endpoint);
        void (*seq_notifier)(struct snd_soc_component *, enum snd_soc_dapm_type,
                int subseq);
        int (*stream_event)(struct snd_soc_component *, int event);
        int (*set_bias_level)(struct snd_soc_component *component,
                              enum snd_soc_bias_level level);

        const struct snd_pcm_ops *ops;
        const struct snd_compr_ops *compr_ops;

        /* probe ordering - for components with runtime dependencies */
        int probe_order;
        int remove_order;

        /* bits */
        unsigned int idle_bias_on:1;
        unsigned int suspend_bias_off:1;
        unsigned int use_pmdown_time:1; /* care pmdown_time at stop */
        unsigned int endianness:1;
        unsigned int non_legacy_dai_naming:1;
};
```

- ✓ name: the name , the ASoC core layer with a different name to distinguish snd_soc_component . _Driver Note that this name and snd_soc_component-> name is not the same . Here name filled in by the driver writers , and component-> name generated automatically by the system .
- ✓ Controls: Define some kcontrols. Refer to the description in the section " Control Logic Device" .
- ✓ dapm_widgets , dapm_routes: and dapm related , dapm fact is kcontrol do a layer of packaging , here temporarily dwell .
- ✓ probe: It will be called back during the initialization phase , see ``Initialization Flowchart'' for details .
- ✓ pcm_new: When the PCM register a new intermediate layer PCM logic device , will callback function , see " Initialization Flow " . This function will generally for each substream is, whom assigned a DMA Buffer, i.e. " The ring buffer described in PCM Logical Device .
- ✓ pcm_free: release the allocated DMA Buffer.
- ✓ SET _sysclk , SET_PLL: when the user space through the ioctl provided hw params time , maybe function may be called back , see "time of the callback function is called" . Maybe general functions provided clk and pll corresponding register .

- ✓ ops: snd_pcm_ops structure , the ASoC core layer will call back the related functions of the structure when necessary . For details , please refer to the ``timing when the callback function is called" . For the definition of each function, please refer to the description in the section `` struct snd_pcm_ops " .

struct snd_soc_dai_driver

struct snd_soc_da i_driver (https://elixir.bootlin.com/linux/v4.20/source/include/sound/soc-dai.h#L247) : The underlying driver needs to prepare the structure when describing a dai .

```
/*
 * Digital Audio Interface Driver.
 *
 * Describes the Digital Audio Interface in terms of its ALSA, DAI and AC97
 * operations and capabilities. Codec and platform drivers will register this
 * structure for every DAI they have.
 *
 * This structure covers the clocking, formating and ALSA operations for each
 * interface.
 */
struct snd_soc_dai_driver {
        /* DAI description */
        const char *name;
        unsigned int id;
        unsigned int base;
        struct snd_soc_dobj dobj;

        /* DAI driver callbacks */
        int (*probe)(struct snd_soc_dai *dai);
        int (*remove)(struct snd_soc_dai *dai);
        int (*suspend)(struct snd_soc_dai *dai);
        int (*resume)(struct snd_soc_dai *dai);
        /* compress dai */
        int (*compress_new)(struct snd_soc_pcm_runtime *rtd, int num);
        /* Optional Callback used at pcm creation*/
        int (*pcm_new)(struct snd_soc_pcm_runtime *rtd,
                       struct snd_soc_dai *dai);
        /* DAI is also used for the control bus */
        bool bus_control;

        /* ops */
        const struct snd_soc_dai_ops *ops;
        const struct snd_soc_cdai_ops *cops;

        /* DAI capabilities */
        struct snd_soc_pcm_stream capture;
        struct snd_soc_pcm_stream playback;
        unsigned int symmetric_rates:1;
        unsigned int symmetric_channels:1;
        unsigned int symmetric_samplebits:1;

        /* probe ordering - for components with runtime dependencies */
        int probe_order;
        int remove_order;
};
```

- ✓ name: uniquely identify the dai.
- ✓ id: The system will automatically number .
- ✓ probe: It will be called back during the initialization phase , see `` Initialization Flowchart " for details .
- ✓ compress_new: generally do not implement it .
- ✓ pcm_new: generally do not need to achieve , because snd_soc_component_driver has been implemented in a pcm _new.
- ✓ OPS: a struct snd_soc_dai_ops type of structure , the dai operation function set , for dai is the construction is critical .

struct snd_soc_dai_ops

struct snd_soc_dai_ops (https://elixir.bootlin.com/linux/v4.20/source/include/sound/soc-dai.h#L146) : The underlying driver needs to prepare the structure when describing a dai , which represents the set of operation functions of dai .

```
struct snd_soc_dai_ops {
	/*
	 * DAI clocking configuration, all optional.
	 * Called by soc_card drivers, normally in their hw_params.
	 */
	int (*set_sysclk)(struct snd_soc_dai *dai,
		int clk_id, unsigned int freq, int dir);
	int (*set_pll)(struct snd_soc_dai *dai, int pll_id, int source,
		unsigned int freq_in, unsigned int freq_out);
	int (*set_clkdiv)(struct snd_soc_dai *dai, int div_id, int div);
	int (*set_bclk_ratio)(struct snd_soc_dai *dai, unsigned int ratio);

	/*
	 * DAI format configuration
	 * Called by soc_card drivers, normally in their hw_params.
	 */
	int (*set_fmt)(struct snd_soc_dai *dai, unsigned int fmt);
	int (*xlate_tdm_slot_mask)(unsigned int slots,
		unsigned int *tx_mask, unsigned int *rx_mask);
	int (*set_tdm_slot)(struct snd_soc_dai *dai,
		unsigned int tx_mask, unsigned int rx_mask,
		int slots, int slot_width);
	int (*set_channel_map)(struct snd_soc_dai *dai,
		unsigned int tx_num, unsigned int *tx_slot,
		unsigned int rx_num, unsigned int *rx_slot);
	int (*set_tristate)(struct snd_soc_dai *dai, int tristate);

	/*
	 * DAI digital mute - optional.
	 * Called by soc-core to minimise any pops.
	 */
	int (*digital_mute)(struct snd_soc_dai *dai, int mute);
	int (*mute_stream)(struct snd_soc_dai *dai, int mute, int stream);

	/*
	 * ALSA PCM audio operations - all optional.
	 * Called by soc-core during audio PCM operations.
	 */
	int (*startup)(struct snd_pcm_substream *,
		struct snd_soc_dai *);
	void (*shutdown)(struct snd_pcm_substream *,
		struct snd_soc_dai *);
	int (*hw_params)(struct snd_pcm_substream *,
		struct snd_pcm_hw_params *, struct snd_soc_dai *);
	int (*hw_free)(struct snd_pcm_substream *,
		struct snd_soc_dai *);
	int (*prepare)(struct snd_pcm_substream *,
		struct snd_soc_dai *);
	/*
	 * NOTE: Commands passed to the trigger function are not necessarily
	 * compatible with the current state of the dai. For example this
	 * sequence of commands is possible: START STOP STOP.
	 * So do not unconditionally use refcounting functions in the trigger
	 * function, e.g. clk_enable/disable.
	 */
	int (*trigger)(struct snd_pcm_substream *, int,
		struct snd_soc_dai *);
	int (*bespoke_trigger)(struct snd_pcm_substream *, int,
		struct snd_soc_dai *);
	/*
	 * For hardware based FIFO caused delay reporting.
	 * Optional.
	 */
	snd_pcm_sframes_t (*delay)(struct snd_pcm_substream *,
		struct snd_soc_dai *);
};
```

- ✓ the SET _xxx: When a user space via ioctl set hw params when , will these callback functions . See "The timing of the callback function is called" .
- ✓ digital_mute , mute_stream: silence .
- ✓ startup .... delay: The ASoC core layer will call back them when needed . For details , please refer to "Timing of Callback Functions" .

2.2.3 M achine

struct snd_soc_card

struct snd_soc_card (https://elixir.bootlin.com/linux/v4.20/source/include/sound/soc.h#L1026) : The Machine driver needs to fill it , and then call snd_soc_register_card to register a sound card . After that, the system will create a sound card , create a PCM logic device , and finally generate a device node . The structure is very long , we only introduce a few commonly used fields :

- ✓ probe: It will be called back during the initialization phase , see ``Initialization Flowchart'' for details .
- ✓ struct snd_soc_dai_link * **dai_link** : Represents multiple dai_link data structures , each dai_link describes a correspondence between I2S and AFIx , which is the most important element under the card .
- ✓ int NUM_LINKS: on behalf of the card define how many dai_links. The link is statically defined , called / * Predefined links * / .
- ✓ add_dai_link: The ASoC core layer will call back this function . In this function, we can dynamically add some dai_links. These links are called dynamically defined . Generally, dynamic uses are seldom used , and this article will not discuss them .
- ✓ struct list_head rtd_list : used to mount all belonging to the card 's snd_soc_pcm_runtime, card how many under dai_link, there will be a number of runtime.
- ✓ controls , dapm_widgets , dapm_routes: You can specify some controls and dapm s, and they will be automatically added to the system during the card registration phase .

struct snd_soc_dai_link

struct snd_soc_dai_link (https://elixir.bootlin.com/linux/v4.20/source/include/sound/soc.h#L1032) : Machine driver needs to fill it , is used to describe a bridge , the bridge from the perspective of the hardware circuit board indicates which cpu_dai with which codec_dai connected , and indicates the use of a circuit board which platfrom and which codec.

```
struct snd_soc_dai_link {
        /* config - must be set by machine driver */
        const char *name;                       /* Codec name */
        const char *stream_name;                /* Stream name */
        /*
         * You MAY specify the link's CPU-side device, either by device name,
         * or by DT/OF node, but not both. If this information is omitted,
         * the CPU-side DAI is matched using .cpu_dai_name only, which hence
         * must be globally unique. These fields are currently typically used
         * only for codec to codec links, or systems using device tree.
         */
        const char *cpu_name;
        struct device_node *cpu_of_node;
        /*
         * You MAY specify the DAI name of the CPU DAI. If this information is
         * omitted, the CPU-side DAI is matched using .cpu_name/.cpu_of_node
         * only, which only works well when that device exposes a single DAI.
         */
        const char *cpu_dai_name;
        /*
         * You MUST specify the link's codec, either by device name, or by
         * DT/OF node, but not both.
         */
        const char *codec_name;
        struct device_node *codec_of_node;
        /* You MUST specify the DAI name within the codec */
        const char *codec_dai_name;

        struct snd_soc_dai_link_component *codecs;
        unsigned int num_codecs;

        /*
         * You MAY specify the link's platform/PCM/DMA driver, either by
         * device name, or by DT/OF node, but not both. Some forms of link
         * do not need a platform.
         */
        const char *platform_name;
        struct device_node *platform_of_node;
        int id; /* optional ID for machine driver link identification */

        const struct snd_soc_pcm_stream *params;
        unsigned int num_params;

        unsigned int dai_fmt;           /* format to set on init */

        enum snd_soc_dpcm_trigger trigger[2]; /* trigger type for DPCM */

        /* codec/machine specific init - e.g. add machine controls */
        int (*init)(struct snd_soc_pcm_runtime *rtd);

        /* optional hw_params re-writing for BE and FE sync */
        int (*be_hw_params_fixup)(struct snd_soc_pcm_runtime *rtd,
                        struct snd_pcm_hw_params *params);

        /* machine stream operations */
        const struct snd_soc_ops *ops;
        const struct snd_soc_compr_ops *compr_ops;

        /* Mark this pcm with non atomic ops */
        bool nonatomic;

        /* For unidirectional dai links */
        unsigned int playback_only:1;
        unsigned int capture_only:1;

        /* Keep DAI active over suspend */
        unsigned int ignore_suspend:1;

        /* Symmetry requirements */
        unsigned int symmetric_rates:1;
        unsigned int symmetric_channels:1;
        unsigned int symmetric_samplebits:1;

        /* Do not create a PCM for this DAI link (Backend link) */
        unsigned int no_pcm:1;

        /* This DAI link can route to other DAI links at runtime (Frontend)*/
        unsigned int dynamic:1;

        /* DPCM capture and Playback support */
        unsigned int dpcm_capture:1;
        unsigned int dpcm_playback:1;

        /* DPCM used FE & BE merged format */
        unsigned int dpcm_merged_format:1;

        /* pmdown_time is ignored at stop */
        unsigned int ignore_pmdown_time:1;

        struct list_head list; /* DAI link list of the soc card */
        struct snd_soc_dobj dobj; /* For topology */
```

- ✓ cpu_name , cpu_of_ node , cpu_ dai_name: used to specify a cpu_dai, a dai_link can only specify a cpu_dai.

  c pu_of_node and cpu_name cannot be specified at the same time , cpu_dai_name must be specified , see " soc_init_dai_link " for details .

  Through these three names, ASoC can find the corresponding cpu_dai, see " snd_soc_find_dai " for details .
- ✓ codec_name , codec_of_node , codec_dai_name : used to specify a codec_dai, a dai_link can specify multiple codec_dais, but this old method can only specify one , so it will gradually be abandoned .
- ✓ struct snd_soc_dai_link_component codecs * , unsigned int num_codecs : This is a new specified codec_dai way , you can specify multiple codec_dai. Which struct snd_soc_dai_link_component (https://elixir.bootlin.com/linux/v4.20/source/include/sound/soc.h#L1017) include name , of_node ,

dai_name these 3 fields , used to identify a codec_link. Name field of the same to Meet the convention in " soc_init_dai_link " . (https://elixir.bootlin.com/linux/v4.20/source/include/sound/soc.h#L1017)

- ✓ platform_name , platform_of_node : Used to specify a platform . The name must meet the conventions in " soc_init_dai_link " . By scanning the component_list linked list and comparing the two names , you can find the component that represents the platform .
- ✓ init: It will be called back during the initialization phase , see the ``initialization flowchart" for details .
- ✓ struct snd_soc_ops * **ops** : The most important operation function set

struct snd_soc_ops

struct snd_soc_ops (https://elixir.bootlin.com/linux/v4.20/source/include/sound/soc.h#L738) : a set of callback functions , AS oC core layer when necessary calls back these functions , see "callback function is called the opportunity " , the module name is dai_link . This ops role of the struct snd_soc_dai_ops very similar , except that it It is for the Machine 's .

```
/* SoC audio ops */
struct snd_soc_ops {
        int (*startup)(struct snd_pcm_substream *);
        void (*shutdown)(struct snd_pcm_substream *);
        int (*hw_params)(struct snd_pcm_substream *, struct snd_pcm_hw_params *);
        int (*hw_free)(struct snd_pcm_substream *);
        int (*prepare)(struct snd_pcm_substream *);
        int (*trigger)(struct snd_pcm_substream *, int);
};
```

2.2.4 Timing when the callback function is called

As mentioned earlier, the ASoC architecture exists under the PCM middle layer , but it only splits the implementation of the PCM bottom layer driver . Therefore, in essence , it interacts with the PCM middle layer . When the user space performs operations on the device node , the final It will be forwarded by the PCM intermediate layer to the ASoC architecture for processing .

According to the knowledge in the " PCM Logic Device" section , we know that the PCM middle layer calls back to the bottom layer through snd_pcm_ops . Therefore , when ASoC registers with the PCM middle layer , it also implements snd_pcm_ops . The code is detailed in soc_new_pcm (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-pcm.c#L3096) : (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-pcm.c#L3096)

```
/* ASoC PCM operations */
if (rtd->dai_link->dynamic) {
        rtd->ops.open           = dpcm_fe_dai_open;
        rtd->ops.hw_params      = dpcm_fe_dai_hw_params;
        rtd->ops.prepare        = dpcm_fe_dai_prepare;
        rtd->ops.trigger        = dpcm_fe_dai_trigger;
        rtd->ops.hw_free        = dpcm_fe_dai_hw_free;
        rtd->ops.close          = dpcm_fe_dai_close;
        rtd->ops.pointer        = soc_pcm_pointer;
        rtd->ops.ioctl          = soc_pcm_ioctl;
} else {
        rtd->ops.open           = soc_pcm_open;
        rtd->ops.hw_params      = soc_pcm_hw_params;
        rtd->ops.prepare        = soc_pcm_prepare;
        rtd->ops.trigger        = soc_pcm_trigger;
        rtd->ops.hw_free        = soc_pcm_hw_free;
        rtd->ops.close          = soc_pcm_close;
        rtd->ops.pointer        = soc_pcm_pointer;
        rtd->ops.ioctl          = soc_pcm_ioctl;
}

for_each_rtdcon(rtd, rtdcon) {
        const struct snd_pcm_ops *ops = rtdcon->component->driver->ops;

        if (!ops)
                continue;

        if (ops->ack)
                rtd->ops.ack            = soc_rtdcon_ack;
        if (ops->copy_user)
                rtd->ops.copy_user      = soc_rtdcon_copy_user;
        if (ops->copy_kernel)
                rtd->ops.copy_kernel    = soc_rtdcon_copy_kernel;
        if (ops->fill_silence)
                rtd->ops.fill_silence   = soc_rtdcon_fill_silence;
        if (ops->page)
                rtd->ops.page           = soc_rtdcon_page;
        if (ops->mmap)
                rtd->ops.mmap           = soc_rtdcon_mmap;
}
```

For the sake of simplicity , it is assumed that several functions in the red box are used . When the user space operates the device node , these functions will eventually be called back by the PCM middle layer (for the timing of the callback, please refer to the description in the section " struct snd_pcm_ops ") . These functions Then it will call the interface function implemented by the ASoC bottom driver . The specific details are summarized as follows :

N O TE: red represent a module name , basket word representative of the function name .

soc_pcm_open (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-pcm.c#L476)

- cpu_dai ->driver->ops-> startup
- component ->driver->ops-> open
- codec_dai ->driver->ops-> startup
- rtd-> dai_link ->ops-> startup

soc_pcm_close (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-pcm.c#L687)

- cpu_dai ->driver->ops-> shutdown
- codec_dai ->driver->ops-> shutdown
- rtd-> dai_link ->ops-> shutdown
- soc_pcm_components_close(substream, NULL)
    - component ->driver->ops-> close

soc_pcm_hw_params (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-pcm.c#L913)

- rtd-> dai_link ->ops-> hw_params
    - In the machine 's hw_params in , as needed might callback function as follows
        - snd_soc_dai_set_fmt (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-core.c#L2543) (xxx_dai , ...)
            - dai->driver->ops-> set_fmt

- snd_soc_dai_set_sysclk (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-core.c#L2423) (xxx_dai , ...)
    - dai->driver->ops-> set_sysclk
    - component ->driver-> set_sysclk
- snd_soc_dai_set_clkdiv (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-core.c#L2466) (xxx_dai , ...)
    - dai->driver->ops-> set_clkdiv
- snd_soc_dai_set_pll (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-core.c#L2486) (xxx_dai , ...)
    - dai->driver->ops-> set_pll
    - component ->driver-> set_pll
- snd_soc_dai_set_channel_map (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-core.c#L2630) (xxx_dai , ...)
    - dai->driver->ops-> set_channel_map

- soc_dai_hw_params(substream, &codec_params, codec_dai)
    - rtd-> dai_link -> be_hw_params_fixup
    - dai->driver->ops-> hw_params
- soc_dai_hw_params(substream, params, cpu_dai)
    - rtd-> dai_link -> be_hw_params_fixup
    - dai->driver->ops-> hw_params
- component ->driver->ops-> hw_params

soc_pcm_prepare (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-pcm.c#L767)

- rtd-> dai_link ->ops-> prepare
- component ->driver->ops-> prepare
- codec_dai ->driver->ops-> prepare
- cpu_dai ->driver->ops-> prepare

soc_pcm_trigger (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-pcm.c#L1087)

- codec_dai ->driver->ops-> trigger
- component ->driver->ops-> trigger
- cpu_dai ->driver->ops-> trigger
- rtd-> dai_link ->ops-> trigger

soc_pcm_hw_free (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-pcm.c#L1034)

- rtd-> dai_link ->ops-> hw_free
- soc_pcm_components_hw_free(substream, NULL)
    - component ->driver->ops-> hw_free
- codec_dai ->driver->ops-> hw_free
- cpu_dai ->driver->ops-> hw_free

soc_pcm_pointer (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-pcm.c#L1161)

- component ->driver->ops-> pointer
- cpu_dai ->driver->ops-> delay
- codec_dai ->driver->ops-> delay

soc_pcm_ioctl (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-pcm.c#L2501)

- component->driver->ops->ioctl

2.3 API analysis

ASoC level drivers can be divided into Platform , CODEC , cpu_dai , codec_dai , Machine this 5 portion .

Among them, platform , codec , cpu_dai , codec_dai are now unified registration methods :

- ✓ for the platform, the underlying need ready struct snd_soc_component_driver
- ✓ For cpu_dai, the underlying need ready struct snd_soc_component_driver (usually contains only name field) and struct snd_soc_dai_driver
- ✓ for the codec and codec_dai, the underlying need ready struct snd_soc_component_driver and struct snd_soc_dai_driver .

Then unified call snd_soc_register_component or devm_ snd_soc_register_component registered to . The latter is to facilitate recycling , equivalent to smart pointers , so here only to discuss snd_soc_register_component .

machine layer to be ready struct snd_soc_card data structure , and then call snd_soc_register_card can .

Two APIs are introduced below .

devm_ snd_soc_register_component

snd_soc_register_component (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-core.c#L3236) the source / sound / soc / soc- core.c definition :

```
int snd_soc_register_component(struct device *dev,
                        const struct snd_soc_component_driver *component_driver,
                        struct snd_soc_dai_driver *dai_drv,
                        int num_dai)
{
        struct snd_soc_component *component;

        component = devm_kzalloc(dev, sizeof(*component), GFP_KERNEL);
        if (!component)
                return -ENOMEM;

        return snd_soc_add_component(dev, component, component_driver,
                                     dai_drv, num_dai);
}
EXPORT_SYMBOL_GPL(snd_soc_register_component);
```

Its parameters are struct snd_soc_component_driver and struct snd_soc_dai_driver , the latter can be NULL if not . The code logic is very clear , allocate snd_soc_component storage space , and then call snd_soc_add_component, as follows :

```
int snd_soc_add_component(struct device *dev,
                          struct snd_soc_component *component,
                          const struct snd_soc_component_driver *component_driver,
                          struct snd_soc_dai_driver *dai_drv,
                          int num_dai)
{
        int ret;
        int i;

        ret = snd_soc_component_initialize(component, component_driver, dev);
        if (ret)
                goto err_free;

        if (component_driver->endianness) {
                for (i = 0; i < num_dai; i++) {
                        convert_endianness_formats(&dai_drv[i].playback);
                        convert_endianness_formats(&dai_drv[i].capture);
                }
        }

        ret = snd_soc_register_dais(component, dai_drv, num_dai);
        if (ret < 0) {
                dev_err(dev, "ASoC: Failed to register DAIs: %d\n", ret);
                goto err_cleanup;
        }

        snd_soc_component_add(component);
        snd_soc_try_rebind_card();

        return 0;

err_cleanup:
        snd_soc_component_cleanup(component);
err_free:
        return ret;
}
EXPORT_SYMBOL_GPL(snd_soc_add_component);
```

- ✓ snd_soc_component_initialize : use component_driver in the field , the new allocation of component - related fields are initialized .
- ✓ snd_soc_register_dais : If there dai_drv, is created num_dai a snd_soc_dai storage space , and initialize , and finally mount them in component-> dai_list down the list .
- ✓ snd_soc_component_add : this component mounted to the global list component_list next .
- ✓ snd_soc_try_rebind_card: for those in unbind_card_list under the card, attempt to rebind. Rebind is actually a step back to go again, "Initialization Flow" in . Here temporarily careful study .

In general , the role of snd_soc_register_component is to create the storage space of the core layer related data structure and initialize them , and then mount all the components to the component_list linked list .
When the machine calls snd_soc_register_card to register, it will traverse this list and find the components described by dai_link . For details, see the next section .

snd_soc_register_card

snd_soc_register_card (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-core.c#L2734) the source / sound / soc / soc- core.c definition .
When the machine after layer data structure can be ready to call this API up , it will start the process to create the entire sound card , and ultimately makes the user space through the device node to play / record audio data .

Its implementation is more complex , we will describe the macro processes "Initialization Flow" in , and then introduce several important functions .

Initialization flow chart

In this macro flow chart , there are several important points :

- ✓ First pay attention to the purple fonts . They are equivalent to using the API provided by the ALSA architecture to create a sound card and PCM logic device and finally register the sound card . This also shows that the upper layer of ASoC is still the original ALSA architecture . About the details and meaning of these APIs , Please refer to Chapter 1 " ALSA " .
- ✓ Secondly red font on behalf of the module name , blue font on behalf of the function name , they represent AS oC core layer underlying driver interface callback function implemented .
- ✓ Finally , **bold** font on behalf of some of the more important functions , the text will detail them . In addition to using text annotations processes .

snd_soc_register_card (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-core.c#L2734) :

- **soc_init_dai_link** : for dai_link [] the legitimacy of the specified parameters to be checked
- snd_soc_bind_card -> snd_soc_instantiate_card
    - soc_bind_dai_link
        - = RTD **soc_new_pcm_runtime** : for each dai_link, create RTD .
        - RTD-> cpu_dai = snd_soc_find_dai : found cpu_da I , and stored in rtd in .
        - snd_soc_rtdcom_add(rtd, rtd->cpu_dai->component)
        - codec_dais [] = snd_soc_find_dai : found codec_dais [], and stored in rtd in .
        - snd_soc_rtdcom_add(rtd, codec_dais[i]->component) : Find the component corresponding to the codec and add it to the rtd->component_list list .
        - rtd->codec_dai = codec_dais[0]
        - for_each_component(component) : Find the component corresponding to the platform and add it to the rtd->component_list list .
          snd_soc_rtdcom_add(rtd, component)
        - soc_add_pcm_runtime : Add this rtd to the card->rtd_list linked list , note that the platform , codec , cpu_dai , and codec_dai we need are already stored under the rtd at this time .
            - list_add_tail(&rtd->list, &card->rtd_list)
    - snd_card_new : Call the ALSA API to create a sound card .
    - card -> probe : First call back the probe function of card .
    - soc_probe_link_components : For rtd each of the next component, the callback function initialized .
        - soc_probe_component :
            - component ->driver-> probe
            - component -> init
    - soc_probe_link_dais : For rtd each under the DAI , the callback function initialized .
        - soc_probe_dai(cpu_dai , order)
            - dai->driver-> probe
        - soc_probe_dai(rtd-> codec_dai s[i], order)
            - dai->driver-> probe
        - dai_link -> init
        - soc_new_pcm : For each rtd (that is, each dai_link) , create a PCM logical device
            - snd_pcm_new
            - snd_pcm_set_ops : The ops here is responsible for interacting with user space .
            - component ->driver-> pcm_new : A ring buffer will be allocated in this callback function .
        - soc_link_dai_pcm_new(&cpu_dai, 1, rtd)
            - *dai->driver->pcm_new (generally drivers do not implement this function)*

        - soc_link_dai_pcm_new(rtd->codec_dais, …)
            - *dai->driver->pcm_new (generally drivers do not implement this function)*
    - snd_soc_add_card_controls(card, card->controls, card->num_controls);
    - snd_soc_dapm_add_routes(&card->dapm, card->dapm_routes, card->num_dapm_routes);
    - card -> late_probe (card);
    - snd_soc_dapm_new_widgets(card);
    - snd_card_register (card- >snd_card) : Register the sound card . After that, the device node appears in the user space , and the user program can interact with the bottom layer .

soc_init_dai_link

This function will dai_link legitimacy check parameters specified , e.g. dai_link not specify simultaneously cpu_name and cpu_of_node, must specify cpu_dai_name , and the like . Logic is simple , direct look at the code to : soc_init_dai_link (https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/soc-core.c#L1262) .

snd_soc_find_dai

snd_soc_find_dai (https://elixir.bootlin.com/linux/v4.17.19/source/sound/soc/soc-core.c#L952) logic is simple , traversing component_list , according of_node , name , dai_name which 3 to find the corresponding number string dai.

soc_new_pcm_runtime

soc_new_pcm_runtime is (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/soc-core.c#L361) used to allocate this data structure described in `` struct snd_soc_pcm_runtime " .

2.4 Example

Platform: atmel-pcm-pdc.c (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/atmel/atmel-pcm-pdc.c)
Cpu_dai: atmel_ssc_dai.c (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/atmel/atmel_ssc_dai.c)
Codec & Codec_dai: wm8904.c (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/codecs/wm8904.c)
Machine: atmel_wm8904.c (https://elixir.bootlin.com/linux/v4.20/source/sound/soc/atmel/atmel_wm8904.c)

2.5 Overview

After finishing this chapter , I found that the ASoC architecture is actually nothing special . In essence, it is a PCM logic device , but it further divides the underlying implementation of this logic device . After understanding the " PCM Logic Device" , look at It will be easy .

3 Alsa Tools

3.1 amixer /alsamixer , aplay, arecord …

The alsa-utils tool set provides these tools , alsa-utils can be downloaded from the alsa-project (https://www.alsa-project.org/main/index.php/Main_Page) official website .

alsamixer is a graphical version of amixer and requires ncurses support . For the usage of these tools , refer to ALSA audio tools (https://www.cnblogs.com/cslunatic/p/3227655.html)amixer, aplay, arecord (https://www.cnblogs.com/cslunatic/p/3227655.html) . (https://www.cnblogs.com/cslunatic/p/3227655.html)

ALSA (Http://Www.Mysixue.Com/?Cat=17) , Linux (Http://Www.Mysixue.Com/?Cat=5)