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/)

CLOCK SUBSYSTEM

April 20, 2019 (Http://Www.Mysixue.Com/?P=129) JL (Http://Www.Mysixue.Com/?Author=1)

1. Architecture introduction

The clock system is a subsystem in the Linux kernel that specifically manages the clock .
Clock is very important in embedded systems , as it is like the pulse of people , driven device operation .
Any CPU, are required to provide it with an external crystal oscillator , the crystal oscillator is used to provide a clock ; any CPU peripherals inside the sheet , but also the operation clock required : e.g. GPIO controller , first of all provides the working clock to it , Only then can its registers be accessed .

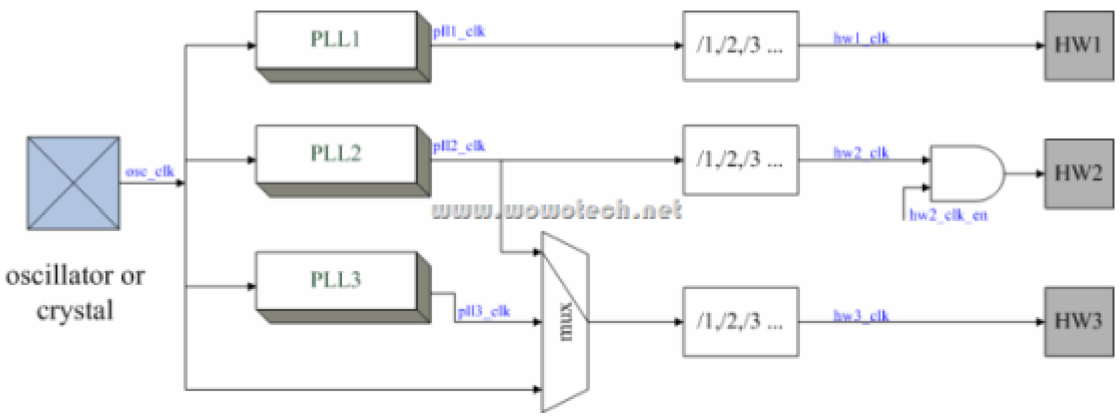
If you look at an ARM CPU chip manual , you will definitely find a chapter that specifically describes the system clock , which is generally called the clock tree .
The chip manual describes how a CPU 's clock system is designed from a hardware perspective , and the Clock subsystem abstracts this design from the software level .

In this chapter , we first take a look from a hardware point of a clock tree is an example , and then himself think about how the software level design , and finally look at clock subsystem is how to do .

2.1 Clock tree

Nowadays , mainstream CPUs that can run Linux all have a very complex clock tree . If we take a spec of a processor and check the chapters related to clock , there must be a very large and complex tree diagram . This diagram is related to clock The components of these devices , and the clock components output by these devices .

The figure below is a simple example :



Clock related devices include

- ➢ for generating a clock of Oscillator (active oscillator , also known as harmonic oscillator) or the Crystal (passive oscillator , also known as crystal)
- ➢ PLL for frequency multiplication (Phase Locked Loop , Phase Locked Loop)
- ➢ for dividing frequency divider
- ➢ Mux for multiple selection
- ➢✌■♎ gate used for clock enable control
- ➢ using clock hardware modules (may be referred to Consumer) , e.g. HW1, it may be GPIO controller

The device is used to generate a specific clock, such as osc_clk. The characteristics of these devices are as follows :

- ➢ input (parent) and output (Children) perspective
  - ■ Some devices without parent, one or more output . For example osc_clk
  - ■ Some devices have a parent, one or more output . For example the PLL1 , its parent is osc_clk, only the output of a pll1_clk
  - ■♦❑❍♏ devices have multiple parents and one or more outputs . For example, MUX, which has multiple parents , has only one output hw3_clk
- ➢☞❒❑❍ the point of view of frequency
  - ■ certain clock frequency can be adjusted , we can set the frequency division factor and the output frequency is adjusted .

    This type of clock is the most common .
  - ■ certain clock frequency is fixed , and can not switch , such osc_clk. The most common is 24M or 25M.

    This type of clock is called fixed_rate_clock
  - ■ certain clock frequency is fixed , but it can be switched .

    This type of clock is called gate_clock
  - ■ certain clock has a fixed multiplication and division factor , its output frequency tracking parent change changes .

    This type of clock is called fixed_factor_clock

In the clock tree on the figure , some clock provider , we can call provider, e.g. oscillator, PLLs; some clock users , we can call Consumer, e.g. HW1, HW2, HW3.
In ARM CPU inside , clock tree system is used to provide various clock ; peripherals on the sheets consume these clocks . E.g. clock tree system is responsible for providing a clock to the GPIO controller , GPIO controller provides it to the consumption operation clock .

In the process of device driver development , a problem we often encounter is : you want to turn the clock of a module .

For example, when developing a GPIO driver , in the probe function of the driver , we need to enable the working clock of the GPIO module .
From the software level , we are to provide a mechanism , so that consumer can easilyGet / enable / configure / disable a clock .

Next , let's look at how to abstract at the software level .

2.2 Software abstraction

In the previous section, we introduced the clock tree , and introduced the providers and consumers of the clock . Inside a CPU chip , there will be many providers, and there will also be many consumers. What the software needs to do is to manage all these providers, and to The consumer provides the simplest possible interface so that the consumer can acquire / enable / configure / close a clock .

Therefore , we can design a pool where all providers can register with the pool and add themselves to the pool . In the pool, a linked list can be used to link all providers together , and different providers are distinguished by different names . When a consumer You need to get a certain clock time , by name query to the pond .

In this pool inside , each provider can be abstracted into a single element , so we best design a data structure , to represent each element .

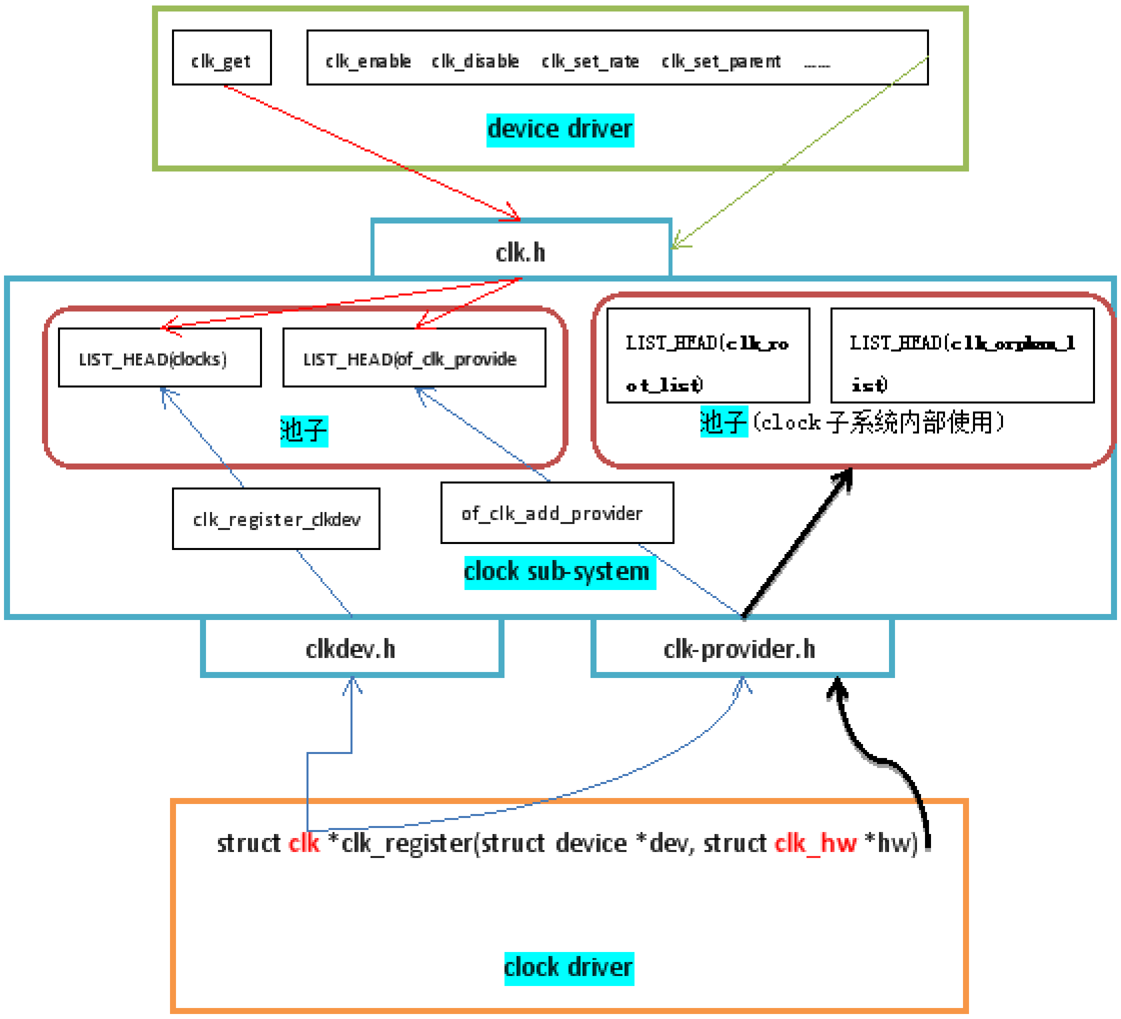
The general logic is this , the clock subsystem of the Linux kernel is basically doing these things .

2.3 clock subsystem

Linux kernel clock subsystem , according to their functions , can be roughly divided into three portions :

- provide a downward registration interface , so that the respective clocks can be registered into the clock subsystem
- in the core layer maintains a pond , manages all registered incoming clocks. This part of the implementation is general logic , independent of specific hardware .
- up , i.e. as the respective consumer clocks module device driver, providing access to / enable / configuration / off clock generic API

The structure diagram of the clock subsystem is as follows , some details you may not understand now . But it doesn't matter , you will understand after reading the following chapters .
In the following chapters , we will also divide it into the above three parts to describe . During the reading process , you can read and understand the picture below , which will be more clear .



2. clock provider — how to register Clocks

3.1 Introduction

In the previous article, we introduced the clock tree . The main problem to be explained in this chapter is how to register the clocks generated by the clock tree into the clock subsystem of the Linux kernel . In other words , how to write a clock driver.

In the ARM CPU internal , management clock tree is also a separate module , generally called PCM (P rogrammable Clock Management), write clock driver is actually written in PCM 's driver.
PCM is the CPU peripherals on a chip , so it will borrow the platform this mechanism . So we need to have platform_device to describe the device , at the same time to have the corresponding platform_driver to control the device . The so-called control , write read PCM 's Register to enable / disable the clock , set the clock frequency and so on . in platform_driver the probe function , there is an important feature , it is to call the clock subsystem provides the API, the clock register subsystem .

Next , let me take a look at the general steps of writing a clock driver .

3.2 General steps for writing clock driver

As you can see from the previous article , first you have to prepare a platform_device.After introducing the device tree mechanism , platform_device is replaced by dts , so we need to describe the clock tree in dts .

Write platform_device (DTS node)

So how to write this DTS ? There are usually two ways :
Way a : the clock tree all Clock , abstracted as a virtual device with a DTS node representation .

```
/* arch/arm/boot/dts/exynos4210.dtsi */
the clock : clock - controller@0x10030000 {
        compatible = "samsung ,exynos4210 -clock" ;
         reg = < 0x10030000 0x20000 >;
         #clock - cells = < 1 >;
} ;
```

This method is very similar to writing a common on-chip peripheral DTS , such as the DTS of a GPIO controller . Compare , is it very similar .
From this example the inside , we can see a clock of DTS node basic syntax :

- compatible , which determines the driver matching this node
- reg is the register used to describe PCM
- clock-cells # , this is a clock provider node unique , indicated that in reference to this clock time , need to use a few 32 bits to describe . What does it mean ?

  Assuming that the clock has only one output clock , then `#clock - cells = < 0 >` , we refer to this clock time , only indicating that this clock can be .
  Refer to this clock What does it mean ? Reference refers to a clock of the consumer end . For example GPIO module requires working clock , then we write GPIO of DTS node time , you need to specify its operating clock is the number , this process is a reference , write a simple Example :

```
gpio: gpio-controller@xxxx {
compatible = " yyyy " ;
    reg = < … .  … .>;
    ...
    clocks = <&theclock>; /* Specify / quote a clock */
}
```

  Assuming that the clock has multiple output clock , then `#clock - cells = < 0 >` Certainly not , because we are in the lead with this clock time , you need to specify which output clock in the end use .
  At this time `#clock - cells` **should be** `< 1 >` , when quoting this clock, you have to write :

```
gpio: gpio-controller@xxxx {
compatible = " yyyy " ;
    reg = < … .  … .>;
    ...
    clocks = <&theclock   num >; /* Specify / reference a certain clock, num is a 32 -bit integer , indicating which output clock to use */
}
```

- theclock , this DTS node of Lable, clock Consumer can refer to that name in accordance with clock

Method 2 : Abstract each clock in the clock tree as a DTS node and organize it in a tree structure . Compared with Method 1 , this method can more clearly abstract the clock tree structure , which is logically also It is more reasonable , so it is recommended that everyone use this method .

Give an example of method two :

```
/* arch/arm/boot/dts/sun4i-a10.dtsi */
clocks {
    #address - cells = < 1 >;
    #size - cells = < 1 >;
    ranges ;

    osc24M : osc24M@01c20050 {
        #clock - cells = < 0 >;
        compatible = "allwinner ,sun4i -osc-clk" ;
        reg = < 0x01c20050 0x4 >;
        clock - frequency = < 24000000 >;
};

    osc32k : osc32k {
        #clock - cells = < 0 >;
        compatible = "fixed-clock" ;
        clock - frequency = < 32768 >;
};

    pll1 : pll1@01c20000 {
        #clock - cells = < 0 >;
        compatible = "allwinner ,sun4i -pll1-clk" ;
        reg = < 0x01c20000 0x4 >;
        clocks = <& osc24M >;
};

/* dummy is 200M */
cpu : cpu@01c20054 {
        #clock - cells = < 0 >;
        compatible = "allwinner ,sun4i -cpu-clk" ;
        reg = < 0x01c20054 0x4 >;
        clocks = <& osc32k >, <& osc24M >, <& pll1 >, <& dummy >;
};

    AXI : AXI @ 01c20054 {
        #clock - cells = < 0 >;
        compatible = "allwinner ,sun4i -axi-clk" ;
        reg = < 0x01c20054 0x4 >;
        clocks = <& cpu >;
};
```

```

     axi_gates : axi_gates@01c2005c {
         #clock - cells = < 1 >;
         compatible = "allwinner ,sun4i -axi-gates-clk" ;
         reg = < 0x01c2005c 0x4 >;
         clocks = <& axi >;
         clock - output - names = “axi_dram” ;
};

     ahb : ahb@01c20054 {
         #clock - cells = < 0 >;
         compatible = "allwinner ,sun4i -ahb-clk" ;
         reg = < 0x01c20054 0x4 >;
         clocks = <& axi >;
};

     ahb_gates : ahb_gates@01c20060 {
         #clock - cells = < 1 >;
         compatible = "allwinner ,sun4i -ahb-gates-clk" ;
         reg = < 0x01c20060 0x8 >;
         clocks = <& ahb >;
         clock - output - names = “ahb_usb0” , “ahb_ehci0” ,
"ahb_ohci0" , "ahb_ehci1" , "ahb_ohci1" , "ahb_ss" ,
"ahb_dma" , "ahb_bist" , "ahb_mmc0" , "ahb_mmc1" ,
"ahb_mmc2" , "ahb_mmc3" , "ahb_ms" , "ahb_nand" ,
"ahb_sdram" , "ahb_ace" , "ahb_emac" , "ahb_ts" ,
"ahb_spi0" , "ahb_spi1" , "ahb_spi2" , "ahb_spi3" ,
"ahb_pata" , "ahb_sata" , "ahb_gps" , "ahb_ve" ,
"ahb_tvd" , "ahb_tve0" , "ahb_tve1" , "ahb_lcd0" ,
"ahb_lcd1" , "ahb_csi0" , "ahb_csi1" , "ahb_hdmi" ,
"ahb_de_be0" , "ahb_de_be1" , "ahb_de_fe0" ,
"ahb_de_fe1" , "ahb_mp" , "ahb_mali400" ;
};

     apb0 : apb0@01c20054 {
         #clock - cells = < 0 >;
         compatible = "allwinner ,sun4i -apb0-clk" ;
         reg = < 0x01c20054 0x4 >;
         clocks = <& ahb >;
};

     apb0_gates : apb0_gates@01c20068 {
         #clock - cells = < 1 >;
         compatible = "allwinner ,sun4i -apb0-gates-clk" ;
         reg = < 0x01c20068 0x4 >;
         clocks = <& apb0 >;
         clock - output - names = “apb0_codec” , “apb0_spdif” ,
"apb0_ac97" , "apb0_iis" , "apb0_pio" , "apb0_ir0" ,
"apb0_ir1" , "apb0_keypad" ;
};

/* dummy is pll62 */
     apb1_mux : apb1_mux@01c20058 {
         #clock - cells = < 0 >;
         compatible = "allwinner ,sun4i -apb1-mux-clk" ;
         reg = < 0x01c20058 0x4 >;
         clocks = <& osc24M >, <& dummy >, <& osc32k >;
};

     apb1 : apb1@01c20058 {
         #clock - cells = < 0 >;
         compatible = "allwinner ,sun4i -apb1-clk" ;
         reg = < 0x01c20058 0x4 >;
         clocks = <& apb1_mux >;
};

     apb1_gates : apb1_gates@01c2006c {
         #clock - cells = < 1 >;
         compatible = "allwinner ,sun4i -apb1-gates-clk" ;
         reg = < 0x01c2006c 0x4 >;
         clocks = <& apb1 >;
         clock - output - names = “apb1_i2c0” , “apb1_i2c1” ,
"apb1_i2c2" , "apb1_can" , "apb1_scr" ,
"apb1_ps20" , "apb1_ps21" , "apb1_uart0" ,
"apb1_uart1" , "apb1_uart2" , "apb1_uart3" ,
"apb1_uart4" , "apb1_uart5" , "apb1_uart6" ,
"apb1_uart7" ;
};
} ;
```

- `osc24M` represents `24M` crystal oscillator
- `osc32k` represents `32k` slow clock
- `pll1` , its parent clock is `osc24M`, only one output clock
- `cpu` , its parent clock has multiple **<&** `osc32k` **>**, **<&** `osc24M` **>**, **<&** `pll1` **>**, **<&** `dummy` **>** , **and** only one output clock
- `ahb_gates` , it has only one parent clock , `which` is `ahb` , but there are many output clocks
- ......The latter are all similar , `so I wo` n't explain them one by one

As you can see , this method can indeed clearly describe the tree relationship of multiple clocks .

Write platform_driver

With platform_device after , the next you have to write platform_driver, the driver most important thing is inside the clock registered subsystem .

How to register ?
The clock subsystem defines the data structure that the clock driver needs to implement , and provides the registered function . We only need to prepare the relevant data structure , and then call the registration function to register .
The definitions of these data structures and interface functions are in : include/linux/ clk-provider.h

The data structure that needs to be implemented is struct clk_hw , and the registration function that needs to be called is struct clk *clk_register(struct device *dev, struct clk_hw *hw) . The details of the data structure and registration function are explained later .

Registration function will always give you a clk struct pointer of type , in Linux 's clock subsystem , using a struct clk represents a clock. Your CPU clock tree in the number of clock, there will be a number corresponding struct clk .

After you get the returned result , you need to call another API: of_clk_add_provider , and throw the returned result you just got into the pool through this API , so that the consumer can get a certain clock from this pool .

The pool concept we had about the foregoing , in addition to the above-described methods , you can use another method to struct clk added to the inside of the pool .
If you want to use this method , you will use several other APIs provided by the clock subsystem . These APIs are defined in : include/linux/ clkdev.h
The most important one API is int clk_register_clkdev (struct CLK *, c onst char *, const char *, ...) , you can order your clock driver calls inside this API, to just get the results returned by this API throw Go inside the pool .

Why do these two methods exist ? This question must be answered from the consumer 's point of view .
We use the GPIO give you an example , GPIO controller need to work the clock , the clock hypothesis called gpio_clk, it is a provider. You need to put this provider registered into the clock subsystem , and to be used to describe this gpio_clk of struct clk added to the pool Inside .
In the driver code of the GPIO controller , we need to obtain the gpio_clk clock and enable it . The process of obtaining is to query the pool .
How to query ? You can directly give a name, and then query the pool through this name ; you can also use clocks = <&theclock>; in the DTS node of the GPIO to specify which clock to use , and then query the pool in this way .
If the consumer is queried by name , the corresponding API added to the pool is clk_register_clkdev
If the consumer is queried through DTS , the corresponding API added to the pool is of_clk_add_provider

So I'm on my clock driver in the end should be inside with which API to add to the pool clk it ?
You should use both at the same time , so that the consumer side can work no matter which query method is used .

After reading this , I suggest you look back at the system block diagram of the clock subsystem , and consider it in conjunction with the block diagram .

Next , we will introduce these data structures and related APIs in detail .

3.3 Main data structure

Through the previous article , we know a main data structure struct clk_hw that the clock driver needs to implement .
There are several other important data structures related to it : struct clk_init_data and struct clk_ops .
Let's take a look at these data structures .

struct clk_hw

Header file : include/linux/clk-provider.h

struct clk_hw	Comment
struct clk_core *core	clk_core is a data structure of the core layer of the clock subsystem . It is created and ma intained by the core layer code . A clk_core corresponds to a specific clock.
struct clk *clk	clk is clock subsystem of a data structure of the core layer , the same core layer created a nd maintained by the code , which also corresponds to a particular clock. clk_core and clk details , on the " Clock Core " described in chapter
const struct clk_init_data *init	clk_init_data is a data structure that the provider needs to implement and fill . To write a clock driver, the main task is to implement it

struct clk_init_data

Header file : include/linux/clk-provider.h

struct clk_init_data	Comment
const char *name	The clock 's name, the system will present a number of clock, each clock will have a na me , it can be used to distinguish the different names clock, and therefore must be uniqu e
const struct clk_ops *ops	And clock -related control ops, for example, enable / disable; set_rate like . When the consumer end want to operate a clock time , eventually it will call to the clock of clk_ops
const char **parent_names	All parents of the clock . By name, you can find out who the parent is
u8 num_parents	A clock may have multiple parents, num_parents indicates how many
unsigned long flags	Flag , indicating that the clock some of the features , the core layer according to differen t code flags take different actions . Optional values are as follows : #define **CLK_SET_RATE_GATE** BIT(0) /* must be gated across rate change */ #define **CLK_SET_PARENT_GATE** BIT(1) /* must be gated across re-parent */ #define **CLK_SET_RATE_PARENT** BIT(2) /* propagate rate change up one level */ #define **CLK_IGNORE_UNUSED** BIT(3) /* do not gate even if unused */ #define **CLK_IS_ROOT** BIT(4) /* root clk, has no parent */ #define **CLK_IS_BASIC** BIT(5) /* Basic clk, can't do a to_clk_foo() */ #define **CLK_GET_RATE_NOCACHE** BIT(6) /* do not use the cached clk rate */ #define **CLK_SET_RATE_NO_REPARENT** BIT(7) /* don't re-parent on rate change */ #define **CLK_GET_ACCURACY_NOCACHE** BIT(8) /* do not use the cached clk accuracy */

struct clk_ops

Header file : include/linux/clk-provider.h

The following ops have detailed comments in the .h file , you can read the source code for more information .

struct clk_ops	**Comment**
int (* **prepare**)(struct clk_hw *hw)	Why is there a prepare interface , ca n't it be enabled directly ? From a hardware point of view , some of the clock, if you want to enable it , need to wai t for some time . For example, the frequency multiplier PLL, after you enable the frequency multiplier , y ou need to wait a few milliseconds for the frequency multiplier to work smoothly . Because you have to wait , it is possible to sleep in the software , which is hibernation . However , there are many situations in the Linux kernel that cannot sleep , such as interr upting the server program . If you put all operations in the enable function , the enable function may sleep , so the e nable function cannot be called in the interrupt service routine . But the actual situation is that many times , we require to turn on / off a certain clock in the interrupt service routine. What to do ? Split into 2 functions , prepare and enable. prepare responsible for enabling clock preparatory work before , prepare which can sle ep , once prepare to return , it means that clock has been completely ready , you can dir ectly open / closed enable responsible for opening the clock, it can not sleep , so the interrupt service routin e can be invoked enable the
void (* **unprepare**)(struct clk_hw *h w)	prepare the inverse function , un-do prepare inside to do all the things
int (* **is_prepared**)(struct clk_hw *hw)	is_prepared , to determine whether the clock has been prepared , it may not be provided . clock framework core maintains a prepare count (the count clk_prepare incremented w hen calling , the clk_unprepare minus one o'clock) , and determines whether the count according to the prepared
void (* **unprepare_unused**)(struct clk _hw *hw)	Automatic unprepare unused clocks clock framework core to provide a clk_disable_unused interfaces , system initialization late_call call , for closing unused Clocks , the interface calls the corresponding clock of . unprepare_unused and .disable_unused function
int (* **enable**)(struct clk_hw *hw)	Enable clock, this function cannot sleep When this function returns , the representative consumer terminal can receive a stable , usable clock waveform a .
void (* **disable**)(struct clk_hw *hw)	Disable clock, this function cannot sleep
int (* **is_enabled**)(struct clk_hw *hw)	Similar to is_prepared
void (* **disable_unused**)(struct clk_hw *hw)	Similar to unprepare_unused
int (* **save_context**)(struct clk_hw *h w)	Save the context of the clock in prepration for poweroff
void (* **restore_context**)(struct clk_hw *hw)	Restore the context o f the clock after a restoration of power

unsigned long(* **recalc_rate**)(struct clk _hw *hw, unsigned long parent_rate)	Take the parent clock rate as the parameter , recalculate and return the clock rate
long (* **round_rate**)(struct clk_hw *h w, unsigned long rate, unsigned long * parent_rate)	Given a target rate as input, returns the closest rate actually supported by the clock The interface is a bit special , in return rounded rate at the same time , will pass a point er , the return round after the parent of Rate . This CLK_SET_RATE_PARENT flag related When the clock Consumer call clk_round_rate obtain an approximate rate when , if the clock is not provided .round_rate functions , there are two methods : ➢ not set CLK_SET_RATE_PARENT when flag , directly back to the clock of the cache r ate ➢ If you set CLK_SET_RATE_PARENT flag , you are asked parent , namely call clk_rou nd_rate get parent Clock , closest to the offer of rate values . What does this mean ? That is , if the parent clock can get an approximate rate val ue , then by changing the parent clock , you can get the required clock
long (* **determine_rate**)(struct clk_hw *hw, unsigned long rate, unsigned long min_rate, unsigned long max_rate, unsigned long *best_parent_rate, struct clk_hw **best_parent_hw)	Similar to round_rate , it is temporarily unclear what the difference is .
int (* **set_parent**)(struct clk_hw *hw, u8 index)	Change the input source of this clock Some clocks may have multiple parents, so which one to use ? Determined by the para meter index
u8 (* **get_parent**)(struct clk_hw *hw)	Queries the hardware to determine the parent of a clock The return value is a u8 types of variables , it is an index, can be found by which the cor responding parent. All parents are stored in .parent_names or .parents arrays inside In fact , this function reads the hardware register , and then converts the value of the reg ister to the corresponding index
int (* **set_rate**)(struct clk_hw *hw, uns igned long rate, unsigned long parent_rate)	Change the rate of this clock The requested rate is specified by the second argument, which should typically be the r eturn of .round_rate call The third argument gives the parent rate which is likely helpful fo r most .set_rate impl ementation
int (* **set_rate_and_parent**)(struct clk _hw *hw, unsigned long rate, unsigned long parent_rate, u8 index)	Change the rate and the parent of this clock This callback is optional (and **unnecessary**) for clocks with 0 or 1 parents as well as for clocks that can tolerate switching the rate and the parent separately via calls to .set_pa rent and .set_rate
unsigned long (* **recalc_accuracy**)(str uct clk_hw *hw, unsigned long parent_accuracy)	Recalculate the accuracy of this clock The clock accuracy is expressed in ppb (parts per billion)

int (* **get_phase**)(struct clk_hw *hw)	Queries the hardware to get the current phase of a clock Returned values are 0-359 degrees on success, negative error codes on failure
int (* **set_phase**)(struct clk_hw *hw, i nt degrees)	Shift the phase this clock signal in degrees specified by the second argument Valid values for degrees are 0-359 Return 0 on success, otherwise -EERROR
void (* **init**)(struct clk_hw *hw)	Perform platform-specific initialization magic However, from the code comments , the kernel recommends that you do not implement this interface function , and it may be abandoned later.
int (* **debug_init**)(struct clk_hw *hw, s truct dentry *dentry)	debugfs related , not detailed here See the code comments for details

3.4 Main API description

The Linux clock subsystem provides several important APIs. Let me look at the details of these APIs one by one .

clk_register / devm_clk_register

Header file : include/linux/clk-provider.h
Implementation file : drivers/clk/clk.c

```
/**
* clk_register - allocate a new clock, register it and return an opaque cookie
* @dev: device that is registering this clock
* @ hw : link to hardware-specific clock data
*
* clk_register is the primary interface for populating the clock tree with new
* clock nodes. It returns a pointer to the newly allocated struct clk which
* cannot be dereferenced by driver code but may be used in conjuction with the
* rest of the clock API. In the event of an error clk_register will return an
* error code; drivers must test for an error code after calling clk_register.
*/
struct clk * clk_register ( struct device * dev , struct clk_hw * hw );
struct clk * devm_clk_register ( struct device * dev , struct clk_hw * hw );

void clk_unregister ( struct clk * clk );
void devm_clk_unregister ( struct device * dev , struct clk * clk );
```

clk_register a clock register subsystem provides the clock of the most basic API functions , it described later other API S is its packaging .
devm_clk_register is clk_register of devm version , devm mechanism described in detail in "device model" in an article .
It is also very simple to register a clock with the system , prepare the clk_hw structure , and then call the clk_register interface .

However , clock Framework than doing this good , which is based on clk_register , but also encapsulates the other interfaces , in the clock -time Subsystem registration , even struct clk_hw do not need to care about , but direct use of human language similar way .

In other words , the actual writing clock driver time , we do not directly use clk_register interfaces , just call one of the following API can .
Below we introduce these APIs one by one .

clk_register_fixed_rate

clock There are different types , some clock frequency is fixed , non-adjustable , for example, an external crystal oscillator , the frequency is fixed (24M / 25M). This type of clock is fixed_rate clock.
fixed_rate Clock has a fixed frequency , can not switch, can not adjust the frequency, can not be selected parent , the need to provide any clk_ops back transfer function , is the simplest class of a Clock .

If you want to register this type of clock, direct calls this API , pass the appropriate parameters to those API to API. Details are as follows :

Header file : include/linux/clk-provider.h
Implementation file : drivers/clk/ clk-fixed-rate.c

```
/*
* DOC: Basic clock implementations common to many platforms
*
* Each basic clock hardware type is comprised of a structure describing the
* clock hardware, implementations of the relevant callbacks in struct clk_ops,
* unique flags for that hardware type, a registration function and an
* alternative macro for static initialization
*/

/**
* struct clk_fixed_rate - fixed-rate clock
* @ hw : handle between common and hardware-specific interfaces
* @fixed_rate: constant frequency of clock
*/
struct clk_fixed_rate {
    struct       clk_hw hw ;
    unsigned long    fixed_rate ;
    unsigned long    fixed_accuracy ;
    u8       flags ;
};

extern const struct clk_ops clk_fixed_rate_ops ;
struct clk * clk_register_fixed_rate ( struct device * dev , const char * name
,
        const char * parent_name , unsigned long flags ,
        unsigned long fixed_rate );
struct clk *clk_register_fixed_rate_with_accuracy ( struct device * dev ,
        const char * name , const char * parent_name , unsigned long flags ,
        unsigned long fixed_rate , unsigned long fixed_accuracy );
```

```
void of_fixed_clk_setup ( struct device_node * np );
```

If want to register a fixed rate clock, except that you can in your own clock driver which calls manually clk_register_fixed_rate addition , there is a more simple way , that is to use DTS.

You can describe a fixed rate clock in DTS as follows :

```
        osc32k : osc32k {
            #clock - cells = < 0 >;
compatible = " fixed-clock " ;
clock - frequency = < 32768 >;
        clock-accuracy = xxxx;
        clock-output-names = xxxx;
};
```

➢❄♒♏ key point is that compatible must be " fixed-clock "

- " the Drivers / CLK / CLK-Fixed-rate.c " in of_fixed_clk_setup will be responsible for matching the compatible, the C file is the clock subsystem implementation .
- of_fixed_clk_setup will parse 3 parameters : clock-frequency , clock-accuracy , clock-output-names
  The clock-frequency is required, and the other 2 parameters are optional .
  After the parameters are parsed , clk_register_fixed_rate_with_accuracy will be called to register a clock with the system .

clk_register_gate

This type of clock can only be switched (.enable/.disable callbacks will be provided) and can be registered using the following interface :

Header file : include/linux/clk-provider.h
Implementation file : drivers/clk/ clk- gate.c

```
/**
* struct clk_gate - gating clock
*
* @ hw : handle between common and hardware-specific interfaces
* @reg: register controlling gate
* @bit_idx: single bit controlling gate
* @flags: hardware-specific flags
* @lock: register lock
*
* Clock which can gate its output. Implements .enable & .disable
*
* Flags:
* CLK_GATE_SET_TO_DISABLE - by default this clock sets the bit at bit_idx to
 * enable the clock. Setting this flag does the opposite: setting the bit
 * disable the clock and clearing it enables the clock
* CLK_GATE_HIWORD_MASK - The gate settings are only in lower 16-bit
 * of this register, and mask of gate bits are in higher 16-bit of this
 * register . While setting the gate bits, higher 16-bit should also be
 * updated to indicate changing gate bits.
*/
struct clk_gate {
    struct clk_hw hw ;
    void __iomem    * reg ;
    u8 bit_idx ;
    u8       flags ;
    spinlock_ t  * lock ;
};

#define CLK_GATE_SET_TO_DISABLE     BIT( 0)
#define CLK_GATE_HIWORD_MASK        BIT( 1)

extern const struct clk_ops clk_gate_ops ;
struct clk * clk_register_gate ( struct device * dev , const char * name ,
        const char * parent_name , unsigned long flags ,
        void __iomem * reg , u8 bit_idx ,
        u8 clk_gate_flags , spinlock_t * lock );
void clk_unregister_gate ( struct clk * clk );
```

clk_register_gate , its parameter list is as follows :

- name : the name of the clock
- parent_name : the name of the parent clock , if not, leave it blank
- flags : Refer to the description of flags in the struct clk_hw structure in section 3.3
- reg : The register address (virtual address) that controls the clock switch
- bit_idx : REG in , first several bit control clock opening / off
- clk_gate_flags : gate clock unique parameter . One of the optional values is CLK_GATE_SET_TO_DISABLE , which means 1 means on or 0 means on , similar to flip bit .
- Lock: If the clock needs to switch the mutex , may provide one the spinlock .

The clock subsystem does not define a DTS processing method similar to the fixed rate clock .
If you want to borrow DTS, it is also very simple . As follows :

```
    ehrpwm1_tbclk : ehrpwm1_tbclk@44e10664 {
        #clock-cells = <0>;
        compatible = "ti,gate-clock" ;
        clocks = <& l4ls_gclk >;
        ti , bit - shift = < 1 >;
        reg = < 0x0664 >;
    };
```

Note that it's compatible, own realization of a driver, that matches this compatible. Then the driver parses inside DTS relevant parameters and call clk_register_gate API can .

clk_register_divider / clk_register_divider_table

This type of clock can set the frequency division value (thus providing .recalc_rate/.set_rate/.round_rate callbacks) , which can be registered through the following two interfaces :

Header file : include/linux/clk-provider.h
Implementation file : drivers/clk/ clk-divider.c

```
/**
* struct clk_divider - adjustable divider clock
*
* @ hw : handle between common and hardware-specific interfaces
* @reg: register containing the divider
* @shift: shift to the divider bit field
* @width: width of the divider bit field
* @table: array of value/divider pairs, last entry should have div = 0
* @lock: register lock
*
* Clock with an adjustable divider affecting its output frequency. Implements
* .recalc_rate, .set_rate and .round_rate
*
* Flags:
* CLK_DIVIDER_ONE_BASED - by default the divisor is the value read from the
 * register plus one. If CLK_DIVIDER_ONE_BASED is set then the divider is
 * the raw value read from the register, with the value of zero considered
 * invalid , unless CLK_DIVIDER_ALLOW_ZERO is set.
* CLK_DIVIDER_POWER_OF_TWO - clock divisor is 2 raised to the value read from
 * the hardware register
* CLK_DIVIDER_ALLOW_ZERO - Allow zero divisors. For dividers which have
 * CLK _DIVIDER_ONE_BASED set, it is possible to end up with a zero divisor.
 * Some hardware implementations gracefully handle this case and allow a
 * zero divisor by not modifying their input clock
 * ( divide by one / bypass).
* CLK_DIVIDER_HIWORD_MASK - The divider settings are only in lower 16-bit
 * of this register, and mask of divider bits are in higher 16-bit of this
 * register . While setting the divider bits, higher 16-bit should also be
 * updated to indicate changing divider bits.
* CLK_DIVIDER_ROUND_CLOSEST - Makes the best calculated divider to be rounded
 * to the closest integer instead of the up one.
* CLK_DIVIDER_READ_ONLY - The divider settings are preconfigured and should
 * not be changed by the clock framework.
*/
struct clk_divider {
    struct clk_hw hw ;
    void __iomem    * reg ;
    u8       shift ;
    u8       width ;
    u8       flags ;
    const struct clk_div_table  * table ;
    spinlock_ t  * lock ;
    u32      context ;
};

#define CLK_DIVIDER_ONE_BASED       BIT( 0)
#define CLK_DIVIDER_POWER_OF_TWO    BIT( 1)
#define CLK_DIVIDER_ALLOW_ZERO      BIT( 2)
#define CLK_DIVIDER_HIWORD_MASK     BIT( 3)
#define CLK_DIVIDER_ROUND_CLOSEST   BIT( 4)
#define CLK_DIVIDER_READ_ONLY       BIT( 5)

extern const struct clk_ops clk_divider_ops ;

unsigned  long  divider_recalc_rate  (  struct  clk_hw  *  hw  ,  unsigned  long
parent_rate ,
        unsigned int val , const struct clk_div_table * table ,
        unsigned long flags );
long divider_round_rate ( struct clk_hw * hw , unsigned long rate ,
        unsigned long * prate , const struct clk_div_table * table ,
        u8 width , unsigned long flags );
int divider_get_val ( unsigned long rate , unsigned long parent_rate ,
        const struct clk_div_table * table , u8 width ,
        unsigned long flags );

struct clk * clk_register_divider ( struct device * dev , const char * name ,
        const char * parent_name , unsigned long flags ,
        void __iomem * reg , u8 shift , u8 width ,
        u8 clk_divider_flags , spinlock_t * lock );
struct  clk  *  clk_register_divider_table  (  struct  device  *  dev  ,  const  char  *
name ,
        const char * parent_name , unsigned long flags ,
        void __iomem * reg , u8 shift , u8 width ,
        u8 clk_divider_flags , const struct clk_div_table * table ,
        spinlock_t * lock );
void clk_unregister_divider ( struct clk * clk );
```

clk_register_divider : This interface is used to register the clock of the division ratio rule

- reg : register to control clock frequency division ratio
- shift : The shift of the bit that controls the frequency division ratio in the register
- width : controlling the frequency division ratio of the bit digits , by default , the actual divider value is a register value plus 1 .
  If there are other exceptions , use the following the flag indication.
- clk_divider_flags : Divider Clock unique Flag , include : CLK_DIVIDER_ONE_BASED :
  The actual divider value is the register value (0 is invalid unless the CLK_DIVIDER_ALLOW_ZERO flag is set)
  CLK_DIVIDER_POWER_OF_TWO :

The actual divider value is the value of the register to the power of 2

CLK_DIVIDER_ALLOW_ZERO :

The divider value can be 0 (no change, depending on hardware support)

clk_register_divider_table : This interface is used to register a clock with an irregular division ratio. Compared with the above interface, the difference is that the correspondence between the divider value and the register value is determined by a table . The prototype of the table is as follows :

```
struct clk_div_table {
    unsigned int     val ;
    unsigned int     div ;
};
```

val represents the register value , and div represents the corresponding frequency division value .

Similarly , the clock subsystem does not implement DTS related interfaces , but you can write your own driver to parse DTS and call API registration .

clk_register_mux

This type clock can select multiple parent , as will be realized .get_parent / .set_parent / .recalc_rate callback , can be registered by the following two interfaces :

Header file : include/linux/clk-provider.h
Implementation file : drivers/clk/ clk- mux .c

```
/**
* struct clk_mux - multiplexer clock
*
* @ hw : handle between common and hardware-specific interfaces
* @reg: register controlling multiplexer
* @shift: shift to multiplexer bit field
* @width: width of mutliplexer bit field
* @flags: hardware-specific flags
* @lock: register lock
*
* Clock with multiple selectable parents. Implements .get_parent, .set_parent
* and .recalc_rate
*
* Flags:
* CLK_MUX_INDEX_ONE - register index starts at 1, not 0
* CLK_MUX_INDEX_BIT - register index is a single bit (power of two)
* CLK_MUX_HIWORD_MASK - The mux settings are only in lower 16-bit of this
 * register , and mask of mux bits are in higher 16-bit of this register.
 * While setting the mux bits, higher 16-bit should also be updated to
 * indicate changing mux bits.
* CLK_MUX_ROUND_CLOSEST - Use the parent rate that is closest to the desired
 * frequency .
*/
```

```
struct clk_mux {
    struct clk_hw hw ;
    void __iomem    * reg ;
    u32     * table ;
    u32      mask ;
    u8       shift ;
    u8       flags ;
    spinlock_ t  * lock ;
    u8 saved_parent ;
};

#define CLK_MUX_INDEX_ONE       BIT( 0)
#define CLK_MUX_INDEX_BIT       BIT( 1)
#define CLK_MUX_HIWORD_MASK     BIT( 2)
#define CLK_MUX_READ_ONLY       BIT( 3) /* mux can't be changed */
#define CLK_MUX_ROUND_CLOSEST       BIT( 4)

extern const struct clk_ops clk_mux_ops ;
extern const struct clk_ops clk_mux_ro_ops ;

struct clk * clk_register_mux ( struct device * dev , const char * name ,
        const char ** parent_names , u8 num_parents , unsigned long flags ,
        void __iomem * reg , u8 shift , u8 width ,
        u8 clk_mux_flags , spinlock_t * lock );

struct clk * clk_register_mux_table ( struct device * dev , const char * name ,
        const char ** parent_names , u8 num_parents , unsigned long flags ,
        void __iomem * reg , u8 shift , u32 mask ,
        clk_mux_flags U8 , U32 * Table , spinlock_t * Lock );

void clk_unregister_mux ( struct clk * clk );
```

clk_register_mux : This interface can register the clock of mux control comparison rules (similar to divider clock)

- parent_names : an array of strings, used to describe all possible parent clocks
- num_parents : the number of parent clocks
- reg , shift , width : select the register, offset, and width of the parent
- clk_mux_flags : MUX Clock unique flag

  CLK_MUX_INDEX_ONE : register value instead from 0 starts, but from a start

  CLK_MUX_INDEX_BIT : The register value is a power of 2

clk_register_mux_table : This interface uses a table to register mux to control the irregular clock . The principle is similar to the divider clock and will not be described in detail.

Similarly , the clock subsystem does not implement DTS related interfaces , but you can write your own driver to parse DTS and call API registration .

clk_register_fixed_factor

This type clock having a fixed factor (i.e., multiplier and Divider) , clock frequency is determined by the parent clock frequency , multiplied by MUL , divided div , used for some of which have a fixed division factor of the clock .

Since the parent clock frequency may be changed , and thus fix factor clock also to change the frequency , and therefore also provide .recalc_rate / .set_rate / .round_rate the callback .

It can be registered through the following interface :

Header file : include/linux/clk-provider.h

Implementation file : drivers/clk/ clk-fixed-factor.c

```
void of_fixed_factor_clk_setup ( struct device_node * node );

/**
* struct clk_fixed_factor - fixed multiplier and divider clock
*
* @ hw : handle between common and hardware-specific interfaces
* @mult: multiplier
* @div: divider
*
* Clock with a fixed multiplier and divider. The output frequency is the
* parent clock rate divided by div and multiplied by mult.
* Implements .recalc_rate, .set_rate and .round_rate
*/

struct clk_fixed_factor {
    struct clk_hw hw ;
    unsigned int     mult ;
    unsigned int     div ;
};

extern struct clk_ops clk_fixed_factor_ops ;
struct clk * clk_register_fixed_factor ( struct device * dev , const char *
name ,
        const char * parent_name , unsigned long flags ,
        unsigned int mult , unsigned int div );
```

The API parameters are relatively simple , so I wo n't say more .

In addition , clock subsystem also provides this type of clock of DTS interfaces .

The of_fixed_factor_clk_setup function in clk-fixed-factor.c will be responsible for parsing DTS. Regarding the matching rules and related parameters of DTS , take a look at the source code for yourself .

clk_register_fractional_divider

This type is very similar to the divider clock , the only difference is that it can support a finer granularity (decimals can be used to represent the frequency division factor). For example, clk = parent / 1.5

The API description is as follows :

Header file : include/linux/clk-provider.h
Implementation file : drivers/clk/ clk-fractional-divider.c

```
/**
* struct clk_fractional_divider - adjustable fractional divider clock
*
* @ hw : handle between common and hardware-specific interfaces
* @reg: register containing the divider
* @mshift: shift to the numerator bit field
* @mwidth: width of the numerator bit field
* @nshift: shift to the denominator bit field
* @nwidth: width of the denominator bit field
* @lock: register lock
*
* Clock with adjustable fractional divider affecting its output frequency.
*/

struct clk_fractional_divider {
    struct clk_hw hw ;
    void __iomem    * reg ;
    u8       mshift ;
    U32      mmask ;
    u8       nshift ;
    U32      nMask ;
    u8       flags ;
    spinlock_ t  * lock ;
};

extern const struct clk_ops clk_fractional_divider_ops ;
struct clk * clk_register_fractional_divider ( struct device * dev ,
        const char * name , const char * parent_name , unsigned long flags ,
        void __iomem * reg , u8 mshift , u8 mwidth , u8 nshift , u8 nwidth ,
        u8 clk_divider_flags , spinlock_t * lock );
```

The clock subsystem does not provide related DTS analysis functions .

clk_register_composite

As the name implies , it is a combination of mux , divider , gate and other clocks , which can be registered through the following interface :

Header file : include/linux/clk-provider.h

Implementation file : drivers/clk/ clk-composite.c

```
/***
* struct clk_composite - aggregate clock of mux, divider and gate clocks
*
* @ hw : handle between common and hardware-specific interfaces
* @mux_hw: handle between composite and hardware-specific mux clock
* @rate_hw: handle between composite and hardware-specific rate clock
* @gate_hw: handle between composite and hardware-specific gate clock
* @mux_ops: clock ops for mux
* @rate_ops: clock ops for rate
* @gate_ops: clock ops for gate
*/
struct clk_composite {
    struct clk_hw hw ;
    struct clk_ops ops ;

    struct clk_hw   * mux_hw ;
    struct clk_hw   * rate_hw ;
    struct clk_hw   * gate_hw ;

    const struct clk_ops    * mux_ops ;
    const struct clk_ops    * rate_ops ;
    const struct clk_ops    * gate_ops ;
};

struct clk * clk_register_composite ( struct device * dev , const char * name ,
        const char ** parent_names , int num_parents ,
        struct clk_hw * mux_hw , const struct clk_ops * mux_ops ,
        struct clk_hw * rate_hw , const struct clk_ops * rate_ops ,
        struct clk_hw * gate_hw , const struct clk_ops * gate_ops ,
        unsigned long flags );
```

Look a bit complicated , but understanding the above 1 to 5 class Clock , here only a coolie , be patient on it .

In addition , the clock subsystem does not provide related DTS analysis functions .

clk_register_gpio_gate

Think of a gpio as a gate clock. That is to say, you can control the gpio on / off through the clock subsystem , that is, control the GPIO output high / low level .

In some cases , some modules required by a GPIO to control the enable / prohibit , for example, a Bluetooth module . And you do not need to provide the clock to this module , the template itself has a crystal exist , you can give yourself for the clock .

This time we can put the GPIO abstracted into gpio gate clock, by means of clock subsystem , to control modules enable / disable.

Header file : include/linux/clk-provider.h
Implementation file : drivers/clk/ clk-gpio-gate.c

```
/***
* struct clk_gpio_gate - gpio gated clock
*
* @ hw : handle between common and hardware-specific interfaces
* @gpiod: gpio descriptor
*
* Clock with a gpio control for enabling and disabling the parent clock.
* Implements .enable, .disable and .is_enabled
*/

struct clk_gpio {
    struct clk_hw hw ;
    struct gpio_desc * gpiod ;
};

extern const struct clk_ops clk_gpio_gate_ops ;
struct clk * clk_register_gpio_gate ( struct device * dev , const char * name ,
        const char * parent_name , unsigned gpio , bool active_low ,
        unsigned long flags );

void of_gpio_clk_gate_setup ( struct device_node * node );
```

The API parameters are relatively simple , so I wo n't say more .

In addition , clock subsystem also provides this type of clock of DTS interfaces .

xxx _unregister

The above xxx_register functions have corresponding xxx_unregister.
unregister function prototype has mentioned in the above code , much to say here , are interested can read the source code itself .

clk_register_clkdev

Header file : include/linux/clkdev.h
Implementation file : drivers/clk/ clk dev.c
Prototype : `int clk_register_clkdev ( struct clk *, const char *, const char *, …);`

Above clk_register_xxx interfaces will be the clock subsystem register a clock, and return to a representative of the clock of the struct clk structure .
The role of `clk_register_clkdev` is to add the returned `struct clk` to a certain pool .

The pond is actually a linked list of friends , the list defined in `clkdev.c` inside .
In the pool, `name` is mainly used as a keyword `to` distinguish different `clk`.

When the `consumer` end want to query a `clk` time , attempts from inside this list by `clock name` to retrieve .

of_clk_add_provider

Header file : include/linux/clk-provider.h
Implementation file : drivers/clk/ clk.c
Prototype :

```
int of_clk_add_provider ( struct device_node * np ,
            struct clk *(* clk_src_get )( struct of_phandle_args * args ,
                           void * data ),
            void * data );
```

The main purpose of this `API` is also to add the obtained `struct clk` structure to a pool .

The pond is also a list , defined in `clk.c` inside .
Above the pond difference is that , here is `device_node` as a key to distinguish between different `clk`.

When the `consumer` end want to query a `clk` time , will be in `DTS node` inside through `Clocks = <& xxxx >` to refer to a `clock`.
By referring to the `phandle` of this `clock` , you can find the corresponding `device_node`. Then you can retrieve the required `clk` from the pool through `device_node` .

In fact , these two pools correspond to the two ways on our `consumer` side : one is to obtain `clk` by `name` without `DTS`; `the` other is to use `DTS`.
As the `Linux` kernel vigorously promotes `DTS`, `the` second method will gradually become the mainstream .

3. clock core — how to manage Clocks

4.1 Introduction

In Chapter 3 , we described one of the functions of the clock subsystem : providing a registration interface to the underlying clock driver .
In this chapter, we describe the second function of the clock subsystem : how to manage these clocks.

When the clock driver registers a clock with the subsystem , a data structure is created inside the subsystem to represent the clock. This data structure was mentioned earlier , which is struct clk.

A struct clk corresponds to a clock. So far , we have not seen the true face of this data structure ! This chapter will focus on this data structure , fully understand it , and you will be in this chapter .

4.2 Main data structure

struct clk

The structure is defined in a C file : drivers/clk/clk.c

struct clk	Comment
struct clk_core *core	clk_core is a private data structure of the clock core layer , which is described in detail b elow
const char *dev_id	The name of the device using the clk
const char *con_id	connection ID string on device
unsigned long min_rate	Minimum rate
unsigned long max_rate	Maximum rate
struct hlist_node clks_node	This structure somewhat subverts a point of this article , that is : a struct clk structure co rresponds to a specific clock. After carefully reading the source code found , the original whenever one driver tries to acquire a clock time , clock subsystem creates a struct clk. For example, assume there is a clock, the name is pll_clk. GPIO, SPI, I2C these . 3 module s are used in this PLL_CLK as the operating clock . Then the clock subsystem core layer will have a clk_core used to describe pll_clk, when ever GPIO / SPI / I2C the driver first attempt to get pll_clk time , clock subsystem creates a struct clk structure , and the creation of the struct clk return For the corresponding driv er. The dev_id and con_id of the clk structure are specified by the corresponding driver . However, for simplicity , we tentatively in this article a struct clk corresponds to a clock it , understand it will not be anything unusual .

struct clk_core

As we said earlier , a struct clk represents a clock. A clk_core and clock have a one-to-one correspondence . So what is the difference between the two ?

struct clk more like external interface , such as when the provider to clock time Subsystem registration , it will get a struct clk * return result ; when the consumer wants to use a clock time , will be the first to obtain struct clk * This structure .

struct clk_core is clock a private data structure subsystem core layer , the core layer described generation of a specific clock. Provider or consumer without touching this data structure .

The structure is defined in a C file : drivers/clk/clk.c

struct clk_core	Comment
const char *name	The name of this clock , the provider will provide the name of the clock when registeri ng

const struct clk_ops *ops	Operate the ops of this clock , the provider will provide the ops of the clock when regist ering
struct clk_hw *hw	The clk_hw structure provided by the provider
struct module *owner	
struct clk_core *parent	A clock may have multiple parents, but only one parent is valid at the same time . This points to the clk_core corresponding to the valid parent clock
const char **parent_names	The names of all optional parent clocks
struct clk_core **parents	Clk_core of all optional parent clocks
u8 num_parents	How many parents
u8 new_parent_index	You can understand all parents as an array , and index corresponds to an element of the array
unsigned long rate	clock 's rate
unsigned long req_rate	The rate required by the consumer
unsigned long new_rate	New rate
struct clk_core *new_parent	New parent
struct clk_core *new_child	New child
unsigned long flags	This clock of flags, an optional flag in 3.3 Jie " struct clk_init_data have introduced" in the
unsigned int enable_count	Number of enabled
u nsigned int prepare_count	It is prepare times
unsigned long accuracy	The current accuracy of this clock , The clock accuracy is expressed in ppb (parts per bill ion)
int phase	The current phase of this clock , the value range is 0 – 359
struct hlist_head children	List head , to mount this clock all children
struct hlist_node child_node	Linked list node , used to hook the clock to the children linked list of the corresponding parent
struct hlist_node debug_node	List node , and debugfs related
struct hlist_head clks	A struct clk_core may correspond to multiple struct clks , and this linked list head is co nnected to all clks. For the relationship between struct clk_core and struct clk , see the description of sectio n 4.2 " struct clk "
unsigned int notifier_count	The kernel notification chain mechanism is related . When other modules in the kernel want to know a change of a clock (such as a change i n frequency, etc.), they can register with the notification chain of this clock . In this wa y, when the clock changes , a notification will be sent to all modules on the notification chain. notifier_count represents how many modules have registered with this clock
struct dentry *dentry	debugfs related
struct kref ref	Reference count

4.3 Key code analysis

clk_register

We in 3.4 section describes the clock subsystem provides to the provider of the total end of the multi-clk_register_xxx functions . These functions are to clk_register package , it is basically clk_register.

Therefore, we only focus on this API in this chapter , and other APIs who are interested can read the source code by themselves .

Before beginning the analysis , combined with the contents explained before , you can guess clk_register what things will do it ?
First , it has to create two data structures, struct clk and struct clk_core .
Then , it has to maintain the clk_core tree relationship , just like the hardware form of the clock number :

> One or more ROOT_CLOCK, ROOT_CLOCK no parent, mounted below are children, children below the mount in children.
> Why such a tree structure to maintain it ? Because we operate in a clk time , often saying the parent about : for example, can make a clk, must ensure that its parent is also enabled ; or the parent 's rate change , that its children 's rate could change . Therefore clock subsystem must maintain good tree structure , in order to facilitate the processing of these correlations .

Let's take a look at the code below :
Implementation file : drivers/clk/clk.c

```
/**
* clk_register - allocate a new clock, register it and return an opaque cookie
* @dev: device that is registering this clock
* @ hw : link to hardware-specific clock data
*
* clk_register is the primary interface for populating the clock tree with new
* clock nodes. It returns a pointer to the newly allocated struct clk which
* cannot be dereferenced by driver code but may be used in conjuction with the
* rest of the clock API. In the event of an error clk_register will return an
* error code; drivers must test for an error code after calling clk_register.
*/
struct clk * clk_register ( struct device * dev , struct clk_hw * hw )
{
    int i , ret ;
    struct clk_core * clk ;

    clk = kzalloc ( sizeof (* clk ), GFP_KERNEL );
    if (! clk ) {
        pr_ err ( "%s: could not allocate clk\n" , __func__ );
        ret = - ENOMEM ;
        goto fail_out ;
    }

    clk -> name = kstrdup_const ( hw -> init -> name , GFP_KERNEL );
    if (! clk -> name ) {
        pr_ err ( “%s: could not allocate clk->name\n” , __func__ );
        ret = - ENOMEM ;
        goto fail_name ;
    }
    clk -> ops = hw -> init -> ops ;
    if ( dev && dev -> driver )
        clk -> owner = dev -> driver -> owner ;
    clk -> hw = hw ;
    clk -> flags = hw -> init -> flags ;
    clk -> num_parents = hw -> init -> num_parents ;
    hw -> core = clk ;

    /* allocate local copy in case parent_names is __initdata */
    clk -> parent_names = kcalloc ( clk -> num_parents , sizeof ( char *),
                    GFP_KERNEL );

    if (! clk -> parent_names ) {
        pr_ err ( “%s: could not allocate clk->parent_names\n” , __func__ );
        ret = - ENOMEM ;
        goto fail_parent_names ;
    }



    /* copy each string name in case parent_names is __initdata */
    for ( i = 0 ; i < clk -> num_parents ; i ++) {
        clk -> parent_names [ i ] = kstrdup_const ( hw -> init -> parent_names [ i ],
                        GFP_KERNEL );
        if (! clk -> parent_names [ i ]) {
            pr_ err ( “%s: could not copy parent_names\n” , __func__ );
            ret = - ENOMEM ;
            goto fail_parent_names_copy ;
        }
    }

    INIT_HLIST_ HEAD ( & clk -> clks );

    hw -> clk = __clk_create_clk ( hw , NULL , NULL );
    if ( IS_ERR ( hw -> clk )) {
        pr_ err ( “%s: could not allocate per-user clk\n” , __func__ );
        ret = PTR_ERR ( hw -> clk );
        goto fail_parent_names_copy ;
    }

    ret = __clk_init ( dev , hw -> clk );
    if (! ret )
```

```
        return hw -> clk ;

    __clk_free_ clk ( hw -> clk );
    hw -> clk = NULL ;

fail_parent_names_copy :
    while (- i >= 0 )
        kfree_ const ( clk -> parent_names [ i ]);
    kfree ( clk -> parent_names );
fail_parent_names :
    kfree_ const ( clk -> name );
fail_name :
    kfree ( clk );
fail_out :
    return ERR_PTR ( ret );
}
EXPORT_SYMBOL_ GPL ( clk_register );
```

The logic of the above code is relatively simple :

- First , create clk_core structure , and then use the provider supplied clk_hw, filling clk_core each word segment .
- Then , call __clk_create_clk , the struct clk structure will be created in __clk_create_clk and its related fields will be initialized . clk_register will return a struct clk* structure pointer to the provider , and this clk* is created here .
  Below we will introduce the function __clk_create_clk separately .
- Finally , calls __clk_init , in __clk_init which will deal with clk_core tree relationship between .
  Below we will introduce the __clk_init function separately .

__clk_create_clk

This API will create a struct clk structure .
To understand the details of this API , you must first clarify the relationship between struct clk and struct clk_core .

We have already introduced the relationship between clk and clk_core in section 4.2 `` struct clk_core " , you can look back .
Here we are doing further explanation .
Remember the two structures of struct file and struct inode that we introduced in the article "Character Device Drivers" ? Inode physically represents a file , and a file corresponds to only one inode; file represents an open file , file When it is opened several times , there will be several files.
clk and clk_core relationship with the above very type , clk_core representatives on a hardware clock, a clock corresponds to a unique a clk_core; and clk represents the use of the clock, such as GPIO driver you want to get a clock, it will get a struct clk , SPI driver wants to get the same clock, it will also get a struct clk, in addition , when this clock the provider to clock time subsystem registration , provider will get a struct clk.

Next, let's look at the code details :

```
struct clk * __clk_create_clk ( struct clk_hw * hw , const char * dev_id ,
                 const char * con_id )
{
    struct clk * clk ;

    /* This is to allow this function to be chained to others */
    if (! hw || IS_ERR ( hw ))
        return ( struct clk *) hw ;

    clk = kzalloc ( sizeof (* clk ), GFP_KERNEL );
    if (! clk )
        return ERR_PTR (- ENOMEM );

    clk -> core = hw -> core ;
    clk -> dev_id = dev_id ;
    clk -> con_id = con_id ;
    clk -> max_rate = ULONG_MAX ;

    clk_prepare_ lock ( );
    hlist_add_ head ( & clk -> clks_node , & hw -> core -> clks );
    clk_prepare_ unlock ( );

    return clk ;
}
```

The logic is relatively simple ：

- first create a `struct clk` structure ， then the relevant parameter initialization structure ， and finally put this `struct clk` mount `clk_core` of `clks` head of the list below .

__clk_init

This `API` is mainly deal with `clock` tree relationship between .
When any `clock` calls the `clk_register` interface to register ， `__clk_init` will be responsible for placing the `clock` in the proper position of the tree structure .

The code details are as follows ：

```
/**
* __clk_init - initialize the data structures in a struct clk
* @dev: device initializing this clk, placeholder for now
* @clk: clk being initialized
*
* Initializes the lists in struct clk, queries the hardware for the
* parent and rate and sets them both.
*/
int __clk_ init ( struct device * dev , struct clk * clk )
{
```

```
int i , ret = 0 ;
struct clk * orphan ;
struct hlist_node * tmp2 ;

if ( ! Clk )
return - EINVAL ;

     clk_prepare_ lock ( );

/* check to see if a clock with this name is already registered */
if ( __clk_ lookup     ( clk -> name )) {
                 pr_ debug ( "%s: clk %s already initialized\n" ,
                 __func__ , clk -> name );
                  ret = - EEXIST ;
goto out ;
}

/* check that clk_ops are sane. See Documentation/clk.txt */
if ( clk ->     ops -> set_rate &&
!( clk -> ops -> round_rate && clk -> ops              -> recalc_rate ))
{
                 pr_ warning ( "%s: %s must implement .round_rate &
.recalc_rate\n" ,
                 __func__ , clk -> name );
           ret =- EINVAL ;
         goto out ;
}

[ : IF ( CLK -> OPS -> set_parent & & ! CLK -> OPS -> get_parent     ) {
                 pr_ warning ( "%s: %s must implement .get_parent &
.set_parent\n" ,
                 __func__ , clk -> name );
           ret = - EINVAL ;
goto out ;
}

/* throw a WARN if any entries in parent_names are NULL */
for ( i = 0 ; i < clk -> num_parents ; i ++)
WARN ( ! CLK -> parent_names         [ I ],
                "%s: invalid NULL in %s's .parent_names\n" ,
                 __func__ , clk -> name );

/*
* Allocate an array of struct clk *'s to avoid unnecessary string
* look-ups of clk's possible parents. This can fail for clocks passed
* in to clk_init during early boot; thus any access to clk-> parents[]
* must always check for a NULL pointer and try to populate it if
* necessary.
*
* If clk->parents is not NULL we skip this entire block. This allows
* for clock drivers to statically initialize clk->parents.
*/
IF ( CLK -> num_parents > . 1 & & ! CLK -> Parents ) {
           clk -> parents = kzalloc ( ( sizeof ( struct clk *) * clk -> num_parents ),
                  GFP_KERNEL );
/*
* __clk_lookup returns NULL for parents that have not been
* clk_init'd; thus any access to clk-> parents[ ] must check
* for a NULL pointer. We can always perform lazy lookups for
* missing parents later on.
*/
if ( clk -> parents )
for ( i = 0 ; i < clk -> num_parents ; i ++)
                  clk -> parents [ i ] =
                      __clk_ lookup ( clk -> parent_names [ i ]);
}

      clk -> parent = __clk_init_ parent ( clk );

/*
* Populate clk->parent if parent has already been __clk_init'd. If
* parent has not yet been __clk_init'd then place clk in the orphan
* list. If clk has set the CLK_IS_ROOT flag then place it in the root
* clk list.
*
* Every time a new clk is clk_init'd then we walk the list of orphan
* clocks and re-parent any that are children of the clock currently
* being clk_init'd.
*/
if ( clk -> parent )
          hlist_add_ head ( & clk -> child_node ,
& clk -> parent -> children );
else if ( clk -> flags & CLK_IS_ROOT )
          hlist_add_ head ( & clk -> child_node , & clk_root_list );
else
          hlist_add_ head ( & clk -> child_node , & clk_orphan_list );

/*
* Set clk's rate. The preferred method is to use .recalc_rate. For
* simple clocks and lazy developers the default fallback is to use the
* parent's rate. If a clock doesn't have a parent (or is orphaned)
* then rate is set to zero.
*/
if ( clk -> ops -> recalc_rate )
         clk -> rate = clk -> ops -> recalc_ rate ( clk -> hw ,
                 __clk_get_ rate ( clk -> parent ));
else if ( clk -> parent )
         clk -> rate = clk -> parent -> rate ;
else
         clk -> rate = 0 ;

/*
* walk the list of orphan clocks and reparent any that are children of
* this clock
*/
      hlist_for_each_entry_ safe ( orphan , tmp2 , & clk_orphan_list , child_node ) {
if ( orphan -> ops -> get_parent ) {
             i = orphan -> ops -> get_ parent ( orphan -> hw );
if ( ! Strcmp ( clk -> name , orphan -> parent_names [ i ]))
                 __clk_ reparent ( orphan , clk );
continue ;
}

for ( i = 0 ; i < orphan -> num_parents ; i ++)
if ( ! Strcmp ( clk -> name , orphan -> parent_names [ i ])) {
                 __clk_ reparent ( orphan , clk );
break ;
}
}

/*
* optional platform-specific magic
*
* The .init callback is not used by any of the basic clock types, but
* exists for weird hardware that must perform initialization magic.
* Please consider other ways of solving initialization problems before
* using this callback, as it’s use is discouraged.
*/
if ( clk -> ops -> init )
         clk -> ops -> init ( clk -> hw );

     clk_debug_ register ( clk );
```

```
140 :
141 : out :
142 :      clk_prepare_ unlock ( );
143 :
144 : return ret ;
145 : }
```

The logic of this piece of code is very complicated , and the main things it does are as follows :

- 20~26 lines, call the __clk_lookup interface with the clock name as the parameter to find out whether there is a clock with the same name registered, if there is, an error will be returned. It can be seen, clock Framework to name uniquely identifies a clock , and therefore can not have the same name as the clock exists
- 28~42 lines, check the integrity of clk ops , for example: if the set_rate interface is provided, the round_rate and recalc_rate interfaces must be provided ; if the set_parent is provided, it must be provided get_parent
- 50 ~ 73 is OK, assign a struct clk * type of array, the cache clock of Parents clock . The specific method is to find the corresponding struct clk pointer according to parents_name
- Line 75 , get the current parent clock and save it in the parent pointer. For details, please refer to the following "Explanation 2 "
- Line 77~93 , according to the characteristics of the clock , add it to one of the three linked lists of clk_root_list ,clk_orphan_list, or parent->children . For details, please refer to "Explanation 1 " below " for
- Line 95~107 , calculate the initial rate of the clock , please refer to the following "Explanation 3 " for details
- 109 ~ 126 lines, attempts reparent all current orphan (orphan) Clock , specifically refer to the following "Description 4 "
- 128~137 lines, if the clock ops provides an init interface, execute it (as can be seen from the comments, the kernel does not recommend providing an init interface)

Description 1 : clock Management and query of

The clock framework has 2 global linked lists: **clk_root_list** and **clk_orphan_list** . All set CLK_IS_ROOT properties clock hangs in clk_root_list in. Other Clock , if there is valid a parent , will be linked to the parent "of Children " linked list, if there is no valid the parent , will hang clk_orphan_list in.

When querying (what the __clk_lookup interface does), search in turn:

```
clk_root_list->root_clk->children->child's children
clk_orphan_list->orphan_clk->children->child's children
```

Can

Note 2 : The current parent clock selection (__clk_init_parent)

For no parent , or only . 1 th parent of the clock , it is relatively simple, is set to NULL , or according to the parent name obtained parent of struct clk contact pointer.

For a plurality of parent 's Clock , must provide .get_parent ops , the ops for the current hardware configuration, such as a register value, all used to return the current parent of the index (i.e., the first of several parent). Then according to the index , take out the struct clk pointer corresponding to the parent clock as the current parent .

Note 3 : Initial rate calculation of clock

For the clock that provides .recalc_rate ops , the ops is first used to obtain the initial rate . If it is not provided, give the second best option and use the rate of the parent clock directly . Finally, if the clock has no parent , the initial rate can only be selected as 0

.recalc_rate ops function, based parent clock of the rate of input parameters, based on the current configuration of hardware such as register values obtained by calculation itself rate value

Note 4 : The reparent of orphan clocks
In some cases, the child clock will be registered before the parent clock . At this time, the child will become the orphan clock and be adopted in the clk_orphan_list .
Whenever a new clock is registered, the kernel will check whether the clock is the parent of a certain orphan . If it is, it will remove the orphan from clk_orphan_list and place it in the arms of the newly registered clock . This is reparent the function of , and its processing logic is:

- traversal orphan List , if orphan provided .get_parent ops , through the ops get the current parent of the index , and from parent_names removed in the parent 's name , and then the new registration clock name comparison, if the same, Oh, found parent of , Execute __clk_reparent for subsequent operations
- If no .get_parent OPS , can only be traversed own parent_names , and check for new registrations clock match, if there is, do __clk_reparent , subsequent operation
- __clk_reparent will remove this orphan from clk_orphan_list and hang it on the newly registered clock . Then call __clk_recalc_rates to recalculate the rate of itself and all its children . The calculation process and the above clock rate is similar to setting

4. clock consumer — how to use Clocks

5.1 Introduction

clock subsystem manages clocks ultimate goal , is to make device driver can easily obtain and use these clocks.

We know that the clock subsystem uses a struct clk structure to abstract a certain one clock.
When the device driver to operate a certain clock time , it needs to do two things :

- First , get the clock. Also called clk_get .
- Then , operate this clock. Such as clk_prepare / clk_enable / clk_disable / clk_set_rate / ...

For example , take the GPIO controller as an example . From a hardware perspective , the GPIO controller needs a working clock . Therefore , in the probe function of the platform_driver of the GPIO controller , you need to operate this clock .

The first step of the operation is to obtain the clock. There are two ways to obtain the clock :

The first is by direct name acquired , for example, assumed known GPIO controller operation clock name is " gpio_clk " , then we can probe the function by which clk_get (& Device , " gpio_clk ") obtained in this way .

Another way is to say that for the GPIO controller , we will have a platfor m_device, which describes the resources of the GPIO controller , that is, the register address , the interrupt number and so on . The working clock of the GPIO controller is also a kind of Resources , can we describe this resource together in platform_device ?
The answer is yes . platform_device is now described in DTS , so the DTS of this GPIO controller can be written like this :

```
gpio: gpio-controller@xxxx {
    compatible = " yyyy " ;
        reg = < ... .  ... .>;
        ...
        clocks = <&theclock>; /* Specify / reference a certain clock*/
    }
```

We use the property clocks to describe clock resources .
The corresponding platform_driver the probe function which , we can just by clk_get (& Device , NULL) in this manner for the desired clock.
In this way , we only need to know the platform_device, and then we can find the corresponding DTS node through it , and then we can get the corresponding clock resource through the " clocks " property of the DTS node .

Acquired clk later , you can pass clock subsystem provides the API to operate clk up .

clock subsystem provides to the consumer side APIs are defined in the header include / linux / clk.h inside .

In the following chapters , we will introduce these APIs in two parts .
Section 5.2 introduces APIs related to obtaining clk .
Section 5.3 introduces APIs related to operating clk .

5.2 APIs - Get clk

Header file : include/linux/clk.h
Implementation file : drivers/clk/clkdev.c

The two main APIs:

```
struct clk * clk_get ( struct device * dev , const char * id );

struct clk * devm_clk_get ( struct device * dev , const char * id );
```

devm_clk_get is the devm version of clk_get. It is used to automatically release resources . We have introduced it in the article ``Device Model'', so I wo n't talk about it here .

In addition to these two most used API addition , there are other API, as follows :

```
struct clk * clk_get_sys ( const char * dev_id , const char * con_id )

struct clk * of_clk_get ( struct device_node * np , int index );
struct clk * of_clk_get_by_name ( struct device_node * np , const char * name
);
struct clk * of_clk_get_from_provider ( struct of_phandle_args * clkspec );
```

clk_get is equivalent to a general logic , it will call the above APIs according to different situations.In actual code , basically we only use clk_get or devm_clk_get.

Next , we will take a closer look at the internal logic of clk_get .

clk_get / devm_clk_get

Header file : include/linux/clk.h

Implementation file : drivers/clk/clkdev.c

Prototype : `struct clk * clk_get ( struct device * dev , const char * id );`

The API main role is in accordance with the parameters , from clock acquisition subsystem in clk to the consumer, then the consumer can operate the clk up . Thus the API return value is used to describe a clock data structure : struct clk.

The code details are as follows :

```
struct clk * clk_get ( struct device * dev , const char * con_id )
{
    const char * dev_id = dev ? dev_ name ( dev ) : NULL ;
    struct clk * clk ;

    if ( dev ) {
        clk = __of_clk_get_by_name ( dev -> of_node , dev_id , con_id );
        if (! IS_ERR ( clk ) || PTR_ERR ( clk ) == - EPROBE_DEFER )
            return clk ;
    }

    return clk_get_sys ( dev_id , con_id );
}
EXPORT_ SYMBOL ( clk_get );
```

If you have fully understood the system block diagram in section 2.3 , then the logic here is very simple :

- If dev is not empty , then take dev->of_node as the parameter , call the of_XXX set , and query a clk from the pool of LIST_HEAD (**of_clk_providers**) .
  If the query is found , the clk is returned . This situation actually corresponds to the DTS node method described in Section 5.1 .
- If the above does not get to clk, then call clk_get_sys, from LIST_HEAD (clocks) inquiry into this pool inside clk. Keyword query is con_id, in fact, it is the clock of name.

5.3 APIs - Operation clk

Header file : include/linux/clk.h

Implementation file : drivers/clk/clk.c

```
int clk_ prepare ( struct clk * clk )
void clk_ unprepare ( struct clk * clk )

static inline int clk_ enable ( struct clk * clk )
static inline void clk_ disable ( struct clk * clk )

static inline unsigned long clk_get_ rate ( struct clk * clk )
static inline int clk_set_ rate ( struct clk * clk , unsigned long rate )
static inline long clk_round_ rate ( struct clk * clk , unsigned long rate )

static inline int clk_set_ parent ( struct clk * clk , struct clk * parent )
static inline struct clk * clk_get_ parent ( struct clk * clk )

static inline int clk_prepare_ enable ( struct clk * clk )
static inline void clk_disable_ unprepare ( struct clk * clk )
```

- clk_enable/clk_disable : start / stop clock . No sleep
- clk_prepare/clk_unprepare : preparation work before starting clock / aftermath work after stopping clock . May sleep
- clk_get_rate / clk_set_rate / clk_round_rate : Clock frequency acquisition and setting , wherein clk_set_rate may not succeed (e.g. no corresponding frequency dividing ratio) , this time will return an error . If you want to make sure that the setting is successful , the need to invoke clk_round_rate interfaces , to give And the rate that needs to be set value that is closer to the that
- clk_set_parent / clk_get_parent : get / select clock of the parent clock
- clk_prepare_enable : The clk_prepare and clk_enable combined, calling together
- clk_disable_unprepare : The clk_disable and clk_unprepare combined, calling together

PREPARE / unprepare , you enable / disable description :

The essence of these two sets of `APIs` is to divide the start / stop of the `clock` into two stages, `atomic` and `non-atomic` , to facilitate implementation and call.
Therefore, the above-mentioned "won't sleep / may sleep" has two meanings:
First, tell the underlying `Clock Driver` , please sleep may cause the operation, put `prepare / unprepare` implemented, must not put `enable / disable` the
Second, a reminder to use the upper `clock` is `Driver` , calling `prepare / unprepare` could sleep when the interface Oh, do not be in `atomic` context (for example, interrupt processing) call Oh, and calls `enable / disable` interfaces can be assured.

In addition, why does the `clock` switch need to sleep? Here is an example, such as `enable PLL clk` , after starting the `PLL` , you need to wait for it to stabilize. The `PLL` settling time is very long, this time put `CPU` surrender (the process of sleep), otherwise wasted `CPU` .

Finally, why is there a combined `clk_prepare_enable/clk_disable_unprepare` interface? If the caller can ensure that it is called in a `non-atomic` context, you can call `prepare/enable` and `disable/unprepared in sequence` . For simplicity, the `framework` helps to encapsulate these two interfaces.

5.4 Other APIs

Header file : include/linux/clk.h
Implementation file : drivers/clk/clk.c

```
int clk_notifier_ register ( struct clk * clk , struct notifier_block * nb );
int clk_notifier_ unregister ( struct clk * clk , struct notifier_block * nb );
```

Both API and kernel provides notification of chain mechanism related .
Both notify interfaces , used to register / unregister clock rate notice changes .
For example, a driver concerned about a clock , expect this clock 's rate is changed , a notification to your own , you can sign up for a the Notify .

Clock Subsystem (Http://Www.Mysixue.Com/?Cat=15) , Linux (Http://Www.Mysixue.Com/?Cat=5)