目录

[1 按键驱动 1](#_Toc533843639)

[1.1 如何查看按键和event\*的对应关系： 1](#_Toc533843640)

[1.2如何添加按键设备：board-mx6q\_sabresd.c 1](#_Toc533843641)

[1.3 如何找到对应的引脚： 2](#_Toc533843642)

[1.4 按键测试程序 2](#_Toc533843643)

[1.5 注册配置GPIO的函数 4](#_Toc533843644)

[1.6 gpio-keys.c中的gpio配置 5](#_Toc533843645)

[1.7 gpio-keys.c 驱动简单分析 5](#_Toc533843646)

[2 LED驱动 10](#_Toc533843647)

[2.1 board-mx6q\_sabresd.c中添加Led设备 10](#_Toc533843648)

[2.2 LED 驱动程序概述 11](#_Toc533843649)

[2.3 LED相关结构体 12](#_Toc533843650)

[2.4 添加led实例 12](#_Toc533843651)

[2.5 LED平台设备 13](#_Toc533843652)

[2.6 应用实例 13](#_Toc533843653)

[3 LCD驱动程序 14](#_Toc533843654)

[3.1 驱动架构 14](#_Toc533843655)

[3.2 uboot启动参数设置例子 14](#_Toc533843656)

[3.3 板子启动时注册的设备过程: 15](#_Toc533843657)

[3.3.1设备数据信息结构体 15](#_Toc533843658)

[3.3.2注册设备 16](#_Toc533843659)

[3.3.3 注册mxc\_sd\_fb过程 16](#_Toc533843660)

[3.3.4注册mxc\_lcdif 16](#_Toc533843661)

[3.3.5 注册mxc\_ldb 16](#_Toc533843662)

[3.3.6 获取分配显存大小 17](#_Toc533843663)

[3.4 mxc\_lcdif驱动分析： 17](#_Toc533843664)

[3.5 mxc\_dispdrv.c功能分析 17](#_Toc533843665)

[3.6 mxc\_ipuv3\_fb.c分析： 19](#_Toc533843666)

[名词解释vpu,ipu,gpu 23](#_Toc533843667)

# 1 按键驱动

驱动源码：drivers/input/keyboard/gpio-keys.c

按键设备：”gpio-keys”,

## 1.1 如何查看按键和event\*的对应关系：

命令行输入：cat /proc/bus/input/devices

[root@EmbedSky /]# cat /proc/bus/input/devices

I: Bus=0019 Vendor=0001 Product=0001 Version=0100

N: Name="gpio-keys"

P: Phys=gpio-keys/input0

S: Sysfs=/devices/platform/gpio-keys/input/input0

U: Uniq=

H: Handlers=event0

B: PROP=0

B: EV=3

B: KEY=100040 0 0 10000000

## 1.2如何添加按键设备：board-mx6q\_sabresd.c

mx6\_sabresd\_board\_init

imx6q\_add\_device\_buttons();

platform\_device\_add\_data(&sabresd\_button\_device,&sabresd\_button\_data,sizeof(sabresd\_button\_data));

platform\_device\_register(&sabresd\_button\_device);

#### 按键占用的资源：

#define GPIO\_BUTTON(gpio\_num, ev\_code, act\_low, descr, wake, debounce) \

{ \

.gpio = gpio\_num, \ //io口编号，所有io一起编号

.type = EV\_KEY, \ //按键类

.code = ev\_code, \ //键码

.active\_low = act\_low, \ //什么电平表示按下

.desc = "btn " descr, \

.wakeup = wake, \

.debounce\_interval = debounce, \ //延时防抖时间

}

static struct gpio\_keys\_button sabresd\_buttons[] =

{

GPIO\_BUTTON(SABRESD\_GPIO\_VOL\_DOWN, KEY\_ENTER, 1, "enter", 0, 1),//KEY\_HOME

GPIO\_BUTTON(SABRESD\_GPIO\_VOL\_UP, KEY\_HOME, 1, "exit", 0, 1),

GPIO\_BUTTON(SABRESD\_GPIO\_POWER, KEY\_POWER, 1, "power", 1, 1),

}

static struct gpio\_keys\_platform\_data sabresd\_button\_data = {

.buttons = sabresd\_buttons,

.nbuttons = ARRAY\_SIZE(sabresd\_buttons),

};

static struct platform\_device sabresd\_button\_device =

{

.name = "gpio-keys",

};

## 1.3 如何找到对应的引脚：

#define SABRESD\_GPIO\_VOL\_UP IMX\_GPIO\_NR(4, 8) //volume ++

#define IMX\_GPIO\_NR(bank, nr) (((bank) - 1) \* 32 + (nr))

数据手册155

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| KEY\_COL1 | U7 | NVCC\_GPIO | GPIO | ALT5 | GPIO4\_IO08 | Input | PU (100K) |

**Table 4-1. Pin Muxing (continued)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Pad Name** | **Mode** | **Instance** | **Port** | **Pad Settings** |
| KEY\_COL1 | ALT0 | ecspi1 | MISO | Hyst. Enable - CFG(Enabled) Drive Strength - CFG(R0DIV6) Pull Up / Down Config. - CFG(100KOhm PU) Pull / Keep Enable - CFG(Enabled) Open Drain Enable - CFG(Disabled) Speed - CFG(100MHz) Pull / Keep Select - CFG(Pull) Slew Rate - CFG(SLOW) |
| ALT1 | enet | MDIO |  |  |
| ALT2 | audmux | AUD5\_TXFS |  |  |
| ALT3 | kpp | COL[1] |  |  |
| ALT4 | uart5 | TXD\_MUX |  |  |
| ALT5 | gpio4 | GPIO[8] |  |  |
| ALT6 | usdhc1 | VSELECT |  |  |

## 1.4 按键测试程序

// 按键测试程序，读取按键是否按下  
#include <stdio.h>

#include <linux/input.h>

#include <fcntl.h>

#include <sys/time.h>

#include <unistd.h>

**void** keyboard\_test()

{

**int** fd=open("/dev/event0",O\_RDWR);

**if**( fd <= 0 )

    {

**printf**("Can not open gpio-keys\n");

    }**else**

    {

**printf**("open gpio-keys\n");

    }

**struct** input\_event \*event;

**char** buf[128] = {0};

     fd\_set rfds;

**while**(1)

{

FD\_ZERO(&rfds);

      FD\_SET(fd, &rfds);

**int** ret = select(fd + 1,&rfds, NULL,NULL,NULL);

**if**(ret < 0)

**printf**("select error\n");

//如果有数据可读，那么select 返回时，fd会在rfds集合中。记住这个rfds和select设置的不//是一个

**if**( FD\_ISSET(fd, &rfds) )

         {

**int** readn = read(fd, buf, **sizeof**(**struct** input\_event));

**if** (readn <= 0)

            {

**printf**("gpio-keys read error %d\n", readn);

**continue**;

             }

**struct** input\_event \*my\_event=(**struct** input\_event\*)buf;

**if**(my\_event->type == EV\_KEY)

            {

**switch**( my\_event->code )

                {

**case** 28:

**if**(my\_event->value==1)

**printf**(" KEY\_ENTER down\n");

**else** **printf**(" KEY\_ENTER up\n");

**break**;

**case** 102:

**if**(my\_event->value==1)

**printf**(" KEY\_HOME down\n");

**else** **printf**(" KEY\_HOME up\n");

**break**;

**case** 116:

**if**(my\_event->value==1)

**printf**(" KEY\_POWER down\n");

**else** **printf**(" KEY\_POWER up\n");

**break**;

**default**:

**break**;

                }

            }

        }

    }

}

**int** main()

{

    keyboard\_test()

**return** 0;

}

#### 实验

[root@EmbedSky key]# ./key\_input\_test

open gpio-keys

KEY\_POWER down

KEY\_POWER up

## 1.5 注册配置GPIO的函数

mx6\_init\_irq

mx6q\_register\_gpios()

mxc\_gpio\_init(mxc\_gpio\_ports, 7);//7组gpio

port[i].chip.direction\_input = mxc\_gpio\_direction\_input;

port[i].chip.direction\_output = mxc\_gpio\_direction\_output;

port[i].chip.get = mxc\_gpio\_get;//获取io口电平

port[i].chip.set = mxc\_gpio\_set;

port[i].chip.base = i \* 32;

port[i].chip.ngpio = 32;

gpiochip\_add(&port[i].chip) //注册

gpio\_desc[id].chip = chip//每一组gpio是一个chip

**static** **struct** mxc\_gpio\_port mxc\_gpio\_ports[] = {

    {

        .chip.label = "gpio-0",

        .base = IO\_ADDRESS(GPIO1\_BASE\_ADDR),

        .irq = MXC\_INT\_GPIO1\_INT15\_0\_NUM,

        .irq\_high = MXC\_INT\_GPIO1\_INT31\_16\_NUM,

        .virtual\_irq\_start = MXC\_GPIO\_IRQ\_START

    },

};

## 1.6 gpio-keys.c中的gpio配置

以SABRESD\_GPIO\_VOL\_UP按键为例子：

#define IMX\_GPIO\_NR(bank, nr) (((bank) - 1) \* 32 + (nr))

#define SABRESD\_GPIO\_VOL\_UP IMX\_GPIO\_NR(4, 8) // gpio\_num

GPIO\_BUTTON(SABRESD\_GPIO\_VOL\_UP, KEY\_HOME, 1, "exit", 0, 1),// .gpio = gpio\_num,

gpio\_keys\_setup\_key(pdev, bdata, button);

gpio\_direction\_input(button->gpio);//button->gpio是io口编号

gpio\_direction\_input(unsigned gpio)

gpio -= chip->base;// gpio换算计算goiox组内的编号

chip->direction\_input(chip, gpio);//实际函数是mxc\_gpio\_direction\_input()

mxc\_gpio\_direction\_input (chip, gpio);

\_set\_gpio\_direction(chip, offset, 0);// offset=gpio

if (dir)

l |= 1 << offset;

else

l &= ~(1 << offset);

\_\_raw\_writel(l, port->base + GPIO\_GDIR);//配置为输入引脚

## 1.7 gpio-keys.c 驱动简单分析

**struct** gpio\_button\_data {

**struct** gpio\_keys\_button \*button;

**struct** input\_dev \*input;

**struct** timer\_list timer;

**struct** work\_struct work;

**int** timer\_debounce;    /\* in msecs \*/

**bool** disabled;

};

**struct** gpio\_keys\_drvdata {

**struct** input\_dev \*input;

**struct** mutex disable\_lock;

    unsigned **int** n\_buttons;

**int** (\*enable)(**struct** device \*dev);

**void** (\*disable)(**struct** device \*dev);

**struct** gpio\_button\_data data[0];

};

**static** **void** gpio\_keys\_report\_event(**struct** gpio\_button\_data \*bdata)

{

**struct** gpio\_keys\_button \*button = bdata->button;

**struct** input\_dev \*input = bdata->input;

    unsigned **int** type = button->type ?: EV\_KEY;

**int** state = (gpio\_get\_value\_cansleep(button->gpio) ? 1 : 0) ^ button->active\_low;

**if** (type == EV\_ABS) {

**if** (state)

            input\_event(input, type, button->code, button->value);

    } **else** {

        input\_event(input, type, button->code, !!state);

    }

    input\_sync(input);

}

**static** **void** gpio\_keys\_work\_func(**struct** work\_struct \*work)

{

**struct** gpio\_button\_data \*bdata =

        container\_of(work, **struct** gpio\_button\_data, work);

    gpio\_keys\_report\_event(bdata);//上报事件

}

**static** **void** gpio\_keys\_timer(unsigned **long** \_data)

{

**struct** gpio\_button\_data \*data = (**struct** gpio\_button\_data \*)\_data;

    schedule\_work(&data->work);

}

**static** irqreturn\_t gpio\_keys\_isr(**int** irq, **void** \*dev\_id)

{

**struct** gpio\_button\_data \*bdata = dev\_id;

**struct** gpio\_keys\_button \*button = bdata->button;

    BUG\_ON(irq != gpio\_to\_irq(button->gpio));

**if** (bdata->timer\_debounce)//执行此分支

        mod\_timer(&bdata->timer,

            jiffies + msecs\_to\_jiffies(bdata->timer\_debounce));

**else**

        schedule\_work(&bdata->work);

**return** IRQ\_HANDLED;

}

**static** **int** \_\_devinit gpio\_keys\_setup\_key(**struct** platform\_device \*pdev,

**struct** gpio\_button\_data \*bdata,

**struct** gpio\_keys\_button \*button)

{

**const** **char** \*desc = button->desc ? button->desc : "gpio\_keys";

**struct** device \*dev = &pdev->dev;

    unsigned **long** irqflags;

**int** irq, error;

    //注册定时器函数，发现不需要add\_timer

    setup\_timer(&bdata->timer, gpio\_keys\_timer, (unsigned **long**)bdata);

    INIT\_WORK(&bdata->work, gpio\_keys\_work\_func);//初始化工作队列

    error = gpio\_request(button->gpio, desc);

    error = gpio\_direction\_input(button->gpio);//button->gpio是io口编号

**if** (button->debounce\_interval) {//设置了debounce\_interval

        error = gpio\_set\_debounce(button->gpio,

                      button->debounce\_interval \* 1000); //没有设置该函数

        /\* use timer if gpiolib doesn't provide debounce \*/

**if** (error < 0)//执行此处

            bdata->timer\_debounce = button->debounce\_interval;

    }

    irq = gpio\_to\_irq(button->gpio);

    irqflags = IRQF\_TRIGGER\_RISING | IRQF\_TRIGGER\_FALLING;

    /\*

     \* If platform has specified that the button can be disabled,

     \* we don't want it to share the interrupt line.

     \*/

**if** (!button->can\_disable)

        irqflags |= IRQF\_SHARED;

    /\*

     \* Resume power key early during syscore instead of at device

     \* resume time.

     \* Some platform like Android need to konw the power key is pressed

     \* then to reume the other devcies

     \*/

**if** (button->wakeup)

        irqflags |= IRQF\_NO\_SUSPEND | IRQF\_EARLY\_RESUME;

    error = request\_any\_context\_irq(irq, gpio\_keys\_isr, irqflags, desc, bdata);

**return** 0;

fail3:

    gpio\_free(button->gpio);

fail2:

**return** error;

}

**static** **int** gpio\_keys\_open(**struct** input\_dev \*input)

{

**struct** gpio\_keys\_drvdata \*ddata = input\_get\_drvdata(input);

**return** ddata->enable ? ddata->enable(input->dev.parent) : 0;

}

**static** **void** gpio\_keys\_close(**struct** input\_dev \*input)

{

**struct** gpio\_keys\_drvdata \*ddata = input\_get\_drvdata(input);

**if** (ddata->disable)

        ddata->disable(input->dev.parent);

}

**static** **int** \_\_devinit gpio\_keys\_probe(**struct** platform\_device \*pdev)

{

    /\*

    pdata=sabresd\_button\_data

    \*/

**struct** gpio\_keys\_platform\_data \*pdata = pdev->dev.platform\_data;

**struct** gpio\_keys\_drvdata \*ddata;

**struct** device \*dev = &pdev->dev;

**struct** input\_dev \*input;

**int** i, error;

**int** wakeup = 0;

    /\*分配了 gpio\_keys\_drvdata，返回值是指向gpio\_keys\_drvdata的指针

    顺便分配了gpio\_button\_data，每一个gpio\_button\_data存储一个按键的数据

    \*/

    ddata = kzalloc(**sizeof**(**struct** gpio\_keys\_drvdata) +

            pdata->nbuttons \* **sizeof**(**struct** gpio\_button\_data),

            GFP\_KERNEL);

    input = input\_allocate\_device();

**if** (!ddata || !input) {

        dev\_err(dev, "failed to allocate state\n");

        error = -ENOMEM;

**goto** fail1;

    }

    ddata->input = input;

    ddata->n\_buttons = pdata->nbuttons;

    ddata->enable = pdata->enable; //使能按键的函数，没有使用

    ddata->disable = pdata->disable; //使能按键的函数，没有使用

    mutex\_init(&ddata->disable\_lock); //使能按键的函数，没有使用

    platform\_set\_drvdata(pdev, ddata);//设备关联驱动数据

    input\_set\_drvdata(input, ddata);

    input->name = pdata->name ? : pdev->name;//设备名字gpio-keys

    input->phys = "gpio-keys/input0";

    input->dev.parent = &pdev->dev;

    input->open = gpio\_keys\_open;

    input->close = gpio\_keys\_close;

 input->id.bustype = BUS\_HOST;

    input->id.vendor = 0x0001;

    input->id.product = 0x0001;

    input->id.version = 0x0100;

    /\* Enable auto repeat feature of Linux input subsystem \*/

**if** (pdata->rep)

        \_\_set\_bit(EV\_REP, input->evbit);//是否使能按键自动重复功能

    //设置GPIO

**for** (i = 0; i < pdata->nbuttons; i++) {

**struct** gpio\_keys\_button \*button = &pdata->buttons[i];//

**struct** gpio\_button\_data \*bdata = &ddata->data[i];

        unsigned **int** type = button->type ?: EV\_KEY;

        bdata->input = input;

        bdata->button = button;

        error = gpio\_keys\_setup\_key(pdev, bdata, button);

**if** (error)

**goto** fail2;

**if** (button->wakeup)

            wakeup = 1;

        input\_set\_capability(input, type, button->code);

    }

    error = sysfs\_create\_group(&pdev->dev.kobj, &gpio\_keys\_attr\_group);

**if** (error) {

        dev\_err(dev, "Unable to export keys/switches, error: %d\n",

            error);

**goto** fail2;

    }

    error = input\_register\_device(input);

**if** (error) {

        dev\_err(dev, "Unable to register input device, error: %d\n",

            error);

**goto** fail3;

    }

    /\* get current state of buttons \*/

**for** (i = 0; i < pdata->nbuttons; i++)

        gpio\_keys\_report\_event(&ddata->data[i]);

    input\_sync(input);

    device\_init\_wakeup(&pdev->dev, wakeup);

**return** 0;

 fail3:

    sysfs\_remove\_group(&pdev->dev.kobj, &gpio\_keys\_attr\_group);

 fail2:

**while** (--i >= 0) {

        free\_irq(gpio\_to\_irq(pdata->buttons[i].gpio), &ddata->data[i]);

**if** (ddata->data[i].timer\_debounce)

            del\_timer\_sync(&ddata->data[i].timer);

        cancel\_work\_sync(&ddata->data[i].work);

        gpio\_free(pdata->buttons[i].gpio);

    }

    platform\_set\_drvdata(pdev, NULL);

 fail1:

    input\_free\_device(input);

    kfree(ddata);

**return** error;

}

**static** **struct** platform\_driver gpio\_keys\_device\_driver = {

    .probe      = gpio\_keys\_probe,

    .**remove**     = \_\_devexit\_p(gpio\_keys\_remove),

    .driver     = {

        .name   = "gpio-keys",

        .owner  = THIS\_MODULE,

    }

};

**static** **int** \_\_init gpio\_keys\_init(**void**)

{

**return** platform\_driver\_register(&gpio\_keys\_device\_driver);

}

**static** **void** \_\_exit gpio\_keys\_exit(**void**)

{

    platform\_driver\_unregister(&gpio\_keys\_device\_driver);

}

# 2 LED驱动

#### 更详细的分析参看linux驱动笔记12章

E9本身没有led,无法实验，在2440开发板中实验

## 2.1 board-mx6q\_sabresd.c中添加Led设备

#define GPIO\_LED(gpio\_led, name\_led, act\_low, state\_suspend, trigger) \

{ \

.gpio = gpio\_led, \

.name = name\_led, \

.active\_low = act\_low, \

.retain\_state\_suspended = state\_suspend, \

.default\_state = 0, \

.default\_trigger = "max8903-"trigger, \

}

static struct gpio\_led imx6q\_gpio\_leds[] =

{

GPIO\_LED(SABRESD\_CHARGE\_NOW, "chg\_now\_led", 0, 1,"charger-charging"),

};

static struct gpio\_led\_platform\_data imx6q\_gpio\_leds\_data =

{

.leds = imx6q\_gpio\_leds,

.num\_leds = ARRAY\_SIZE(imx6q\_gpio\_leds),

};

static struct platform\_device imx6q\_gpio\_led\_device =

{

.name = "leds-gpio",

.id = -1,

.num\_resources = 0,

.dev = {

.platform\_data = &imx6q\_gpio\_leds\_data,

}

};

static void \_\_init imx6q\_add\_device\_gpio\_leds(void)

{

platform\_device\_register(&imx6q\_gpio\_led\_device);

}

#### 结构体gpio\_led\_platform\_data

struct gpio\_led\_platform\_data {

int num\_leds;

const struct gpio\_led \*leds;

#define GPIO\_LED\_NO\_BLINK\_LOW 0 /\* No blink GPIO state low \*/

#define GPIO\_LED\_NO\_BLINK\_HIGH 1 /\* No blink GPIO state high \*/

#define GPIO\_LED\_BLINK 2 /\* Please, blink \*/

int (\*gpio\_blink\_set)(unsigned gpio, int state, //设置led闪烁函数

unsigned long \*delay\_on,

unsigned long \*delay\_off);

};

## 2.2 LED 驱动程序概述

leds-gpio封装得十分好，只需要提供可正常使用的GPIO即可。另外还具备触发器功能，其实就是控制LED的亮灭(及频率)。比如default-on是点亮LED灯的触发器，没有取消前一直亮着。heartbeat是心跳触发器，经笔者实践，此触发器是快速闪烁2次，然后灭掉，灭掉时间较亮的时间长。timer为定时触发器，即1HZ内亮灭。其它还有如ide硬盘、mmc、CPU触发器，就不一一介绍了。

leds驱动位于drivers/leds目录。leds-gpio驱动名称为“leds-gpio”，驱动文件为drivers/leds/leds-gpio.c。触发器驱动位于drivers/leds/trigger目录。

内核配置

#### 配置内核支持该驱动

Device Drivers->

-\*- LED Support --->

{\*} LED Class Support # 与用户空间交互的

<M> LED Support for GPIO connected LEDs # 可为模块，也可编译到内核中

-\*- LED Trigger support ---> #触发器，最好编译到内核中（即选项“\*”）

<\*> LED Timer Trigger

<\*> LED One-shot Trigger

<\*> LED Heartbeat Trigger

<\*> LED backlight Trigger

[\*] LED CPU Trigger

<\*> LED GPIO Trigger

<\*> LED Default ON Trigger

## 2.3 LED相关结构体

驱动开发者使用gpio\_led对LED进行赋值，包括LED名称、GPIO引脚号、灯亮是哪个电平，还有默认状态。gpio\_led结构体定义如下：

struct gpio\_led {

const char \*name; // 名称，会生成/sys/.../leds/name目录

const char \*default\_trigger; // 默认触发器，可写可不写，在命令行可以重新赋值

unsigned gpio; // GPIO引脚号

unsigned active\_low : 1; // 为1表示低电平LED点亮

unsigned retain\_state\_suspended : 1;

unsigned default\_state : 2; // 默认状态

/\* default\_state should be one of LEDS\_GPIO\_DEFSTATE\_(ON|OFF|KEEP) \*/

}

struct gpio\_led\_platform\_data {

int num\_leds; // 一共有多少个LED灯

const struct gpio\_led \*leds; // 上面的结构体指针

#define GPIO\_LED\_NO\_BLINK\_LOW 0 /\* No blink GPIO state low \*/

#define GPIO\_LED\_NO\_BLINK\_HIGH 1 /\* No blink GPIO state high \*/

#define GPIO\_LED\_BLINK 2 /\* Please, blink \*/

int (\*gpio\_blink\_set)(unsigned gpio, int state,

unsigned long \*delay\_on,

unsigned long \*delay\_off); // LED闪烁回调函数，可置为NULL

};

## 2.4 添加led实例

static struct gpio\_led gpio\_leds[] = {

{

.name = "red",

.gpio = 33, // GP2\_1 GPIO\_NO = Group \* 32 + Id

.default\_state = LEDS\_GPIO\_DEFSTATE\_ON, // 默认LED亮

.active\_low = 1, // 低电平亮 //.default\_trigger = "timer", // 触发器 },

{

.name = "green",

.gpio = 34,

.default\_state = LEDS\_GPIO\_DEFSTATE\_ON,

.active\_low = 1,

//.default\_trigger = "heartbeat",

},

};

static struct gpio\_led\_platform\_data gpio\_led\_info = {

.leds = gpio\_leds,

.num\_leds = ARRAY\_SIZE(gpio\_leds),

};

## 2.5 LED平台设备

leds-gpio驱动定义如下(drivers/leds/leds-gpio.c)：

static struct platform\_driver gpio\_led\_driver = {

.probe = gpio\_led\_probe,

.remove = gpio\_led\_remove,

.driver = {

.name = "leds-gpio",

.owner = THIS\_MODULE,

.of\_match\_table = of\_match\_ptr(of\_gpio\_leds\_match),

},

};

module\_platform\_driver(gpio\_led\_driver);

从gpio\_led\_driver结构体中可以看到驱动名称为leds-gpio。因此要使用这个驱动，必须另外定义一个platform设备，并调用函数platform\_device\_register注册。本文实例如下：

static struct platform\_device leds\_gpio = {

.name = "leds-gpio",

.id = -1,

.dev = {

.platform\_data = &gpio\_led\_info,

.release = platformdev\_release,

},

};

其中name表示设备名称，必须为“leds-gpio”，这样才能匹配并加载成功。最后提一下release成员，在较新的内核中必须对此进行赋值，否则会有错误信息提示：

Device 'leds-gpio' does not have a release() function, it is broken and must be fixed.

最后，注册leds设备——建议在板子的GPIO正常工作之后再进行注册。

platform\_device\_register(&leds\_gpio); // 注册leds设备注意，如果是以modules形式动态加载的话，必须要适合的地方如remove函数在卸载leds设备：

platform\_device\_unregister(&leds\_gpio); // 卸载leds设备

## 2.6 应用实例

LED设备和驱动都正常情况下，系统启动后，会产生/sys/bus/platform/devices/leds-gpio/leds目录，其下分别有red和green两个子目录。可以分别对不同的红色LED和绿色LED做操作。

亮灭LED

将1或0写入brightness文件即可控制亮灭。

示例如下：

echo 0 > /sys/bus/platform/devices/leds-gpio/leds/green/brightnessecho 0 > /sys/bus/platform/devices/leds-gpio/leds/red/brightness echo 1 > /sys/bus/platform/devices/leds-gpio/leds/green/brightnessecho 1 > /sys/bus/platform/devices/leds-gpio/leds/red/brightness

触发器

直接查看trigger文件，即可知道当前系统支持的触发器，示例：

cat /sys/bus/platform/devices/leds-gpio/leds/red/trigger [none] timer oneshot heartbeat backlight gpio cpu0 default-on mmc0 mmc1 mmc2

在前面的驱动中注释掉了trigger，所以现在是none。

设置触发器很简单，使用ecoh将需要的触发器名称写入trigger文件即可。注意，写入的字符串一定是trigger文件已经存在的，否则会提示参数非法。写入心跳触发器示例：

echo heartbeat > /sys/bus/platform/devices/leds-gpio/leds/red/trigger此时板子上红灯应会闪烁。

再次查看：

cat /sys/bus/platform/devices/leds-gpio/leds/red/trigger none timer oneshot [heartbeat] backlight gpio cpu0 default-on mmc0 mmc1 mmc2设置值已经生效了。

# 3 LCD驱动程序

## 3.1 驱动架构

1. mxc\_ipuv3\_fb.c 是imx6具体显示模块平台核心的驱动，最终用于把fb\_info结构体注册到framebuffer驱动核心层中。

2> mxc\_dispdrv.c 通用注册handle接口，用于统一管理imx6显示模块的各种接口，

比如lvds，lcd，hdmi等显示接口。

3> ldb.c是LVDS 驱动的具体实现过程。

4> mxc\_dvi.c 液晶显示器dvi接口驱动，vga是模拟接口，这个是数字接口

5> mxc\_lcdif.c RGB接口lcd驱动

## 3.2 uboot启动参数设置例子

lcd设置例子：

displayArgs=video=mxcfb0:dev=lcd,CLAA-WVGA,if=RGB24,bpp=32 video=mxcfb1:off video=mxcfb2:off fbmem=48M

lvds 设置例子：

video=mxcfb0:dev=ldb,LDB-WSVGA,if=RGB24,bpp=32 video=mxcfb1:off video=mxcfb2:off ldb=sin0 fbmem=28M fb0base=0x27b00000 '

## 3.3 板子启动时注册的设备过程:

### 3.3.1设备数据信息结构体

static struct ipuv3\_fb\_platform\_data sabresd\_fb\_data[] =

{

{

.disp\_dev = "ldb",

.interface\_pix\_fmt = IPU\_PIX\_FMT\_RGB666,

.mode\_str = "LDB-XGA",

.default\_bpp = 16,

.int\_clk = false,

.late\_init = false,

},

{

.disp\_dev = "hdmi",

.interface\_pix\_fmt = IPU\_PIX\_FMT\_RGB24,

.mode\_str = "1920x1080M@60",

.default\_bpp = 32,

.int\_clk = false,

.late\_init = false,

},

{

.disp\_dev = "lcd",

.interface\_pix\_fmt = IPU\_PIX\_FMT\_RGB24,

.mode\_str = "TQ-TN92",

.default\_bpp = 24,

.int\_clk = false,

.late\_init = false,

},

};

static struct fsl\_mxc\_lcd\_platform\_data lcdif\_data =

{

.ipu\_id = 0, //共有两个ipu，使用第0个

.disp\_id = 0, //每个ipu有两路使用第0路

// .default\_ifmt = IPU\_PIX\_FMT\_RGB565,

.default\_ifmt = IPU\_PIX\_FMT\_RGB24,

};

static struct fsl\_mxc\_ldb\_platform\_data ldb\_data =

{

.ipu\_id = 0,

.disp\_id = 1,

.ext\_ref = 1,

.mode = LDB\_SEP1,

.sec\_ipu\_id = 0,

.sec\_disp\_id = 0,

};

static struct imx\_ipuv3\_platform\_data ipu\_data[] =

{

{

.rev = 4,

.csi\_clk[0] = "clko\_clk",

.bypass\_reset = false,

}, {

.rev = 4,

.csi\_clk[0] = "clko\_clk",

.bypass\_reset = false,

},

};

### 3.3.2注册设备

imx6q\_add\_ipuv3(0, &ipu\_data[0]);

if (cpu\_is\_mx6q())

{

imx6q\_add\_ipuv3(1, &ipu\_data[1]);

for (i = 0; i < 4 && i < ARRAY\_SIZE(sabresd\_fb\_data); i++)

imx6q\_add\_ipuv3fb(i, &sabresd\_fb\_data[i]); //注册显示设备

}

imx6q\_add\_mipi\_dsi(&mipi\_dsi\_pdata);

imx6q\_add\_lcdif(&lcdif\_data);

imx6q\_add\_ldb(&ldb\_data);

### 3.3.3 注册mxc\_sd\_fb过程

imx6q\_add\_ipuv3fb(i, &sabresd\_fb\_data[i]);

imx\_add\_ipuv3\_fb(id, pdata)

imx\_add\_platform\_device\_dmamask("mxc\_sdc\_fb", id,NULL, 0, pdata, sizeof(\*pdata),

DMA\_BIT\_MASK(32));

platform\_device\_alloc(name, id);//name="mxc\_sd\_fb"

pa = kzalloc(sizeof(struct platform\_object) + strlen(name), GFP\_KERNEL);

if (pa) {

strcpy(pa->name, name);

pa->pdev.name = pa->name;//注册的设备名字是"mxc\_sd\_fb"

pa->pdev.id = id;

device\_initialize(&pa->pdev.dev);

}

platform\_device\_add(pdev);//注册设备

//注册的platform最终名字

dev\_set\_name(&pdev->dev, "%s.%d", pdev->name, pdev->id);

device\_add(&pdev->dev);

#### 在用户空间查找注册设备名字：

[root@EmbedSky ]# cd /sys/devices/platform/

[root@EmbedSky platform]# ls

imx-ocotp.0 mxc\_sdc\_fb.0 soc-audio.5

imx-pcie mxc\_sdc\_fb.1 uevent

imx-pcm-audio.1 mxc\_sdc\_fb.2 usb-wakeup.0

### 3.3.4注册mxc\_lcdif

imx6q\_add\_lcdif(&lcdif\_data)

platform\_device\_register\_resndata(NULL, "mxc\_lcdif",0, NULL, 0, pdata, sizeof(\*pdata));

platform\_device\_alloc(name, id);//id=0

pa->pdev.name = pa->name;

platform\_device\_add(pdev);

if (pdev->id != -1)//只有一个设备，但是id=0,执行此分支

dev\_set\_name(&pdev->dev, "%s.%d", pdev->name, pdev->id);

else

dev\_set\_name(&pdev->dev, "%s", pdev->name);

device\_add(&pdev->dev);

[root@EmbedSky platform]# ls

mxc\_lcdif.0

### 3.3.5 注册mxc\_ldb

imx6q\_add\_ldb(&ldb\_data);

imx\_add\_ldb(&imx6q\_ldb\_data, pdata);

imx\_add\_platform\_device("mxc\_ldb", -1,res, ARRAY\_SIZE(res), pdata, sizeof(\*pdata));

imx\_add\_platform\_device\_dmamask(name, id, res, num\_resources, data, size\_data, 0);

platform\_device\_alloc(name, id);

platform\_device\_add(pdev);

if (pdev->id != -1)

dev\_set\_name(&pdev->dev, "%s.%d", pdev->name, pdev->id);

else//id=-1,执行此分支

dev\_set\_name(&pdev->dev, "%s", pdev->name);

device\_add(&pdev->dev);

[root@EmbedSky platform]# ls

mxc\_ldb

### 3.3.6 获取分配显存大小

.fixup = fixup\_mxc\_board

struct ipuv3\_fb\_platform\_data \*pdata\_fb = sabresd\_fb\_data;

str = strstr(str, "fbmem=");

pdata\_fb[i++].res\_size[0] = memparse(str, &str);

## 3.4 mxc\_lcdif驱动分析：

#define DISPDRV\_LCD "lcd"

static struct mxc\_dispdrv\_driver lcdif\_drv = {

.name = DISPDRV\_LCD,

.init = lcdif\_init,

.deinit = lcdif\_deinit,

};

struct mxc\_lcdif\_data {

struct platform\_device \*pdev;

struct mxc\_dispdrv\_handle \*disp\_lcdif;

};

#### mxc\_lcdif\_probe函数

struct mxc\_lcdif\_data \*lcdif;

lcdif->pdev = pdev;

lcdif->disp\_lcdif = mxc\_dispdrv\_register(&lcdif\_drv);

list\_add\_tail(&new->list, & dispdrv\_list);//添加到显示驱动dispdrv\_list列表

mxc\_dispdrv\_setdata(lcdif->disp\_lcdif, lcdif);

//entry->priv = data= lcdif

dev\_set\_drvdata(&pdev->dev, lcdif);

dev\_set\_drvdata(&pdev->dev, lcdif);

//pedv->dev->p->driver\_data = data;

## 3.5 mxc\_dispdrv.c功能分析

static LIST\_HEAD(dispdrv\_list);//显示驱动链表头

struct mxc\_dispdrv\_entry {

/\* Note: drv always the first element \*/

struct mxc\_dispdrv\_driver \*drv;

bool active;

void \*priv;

struct list\_head list;

struct device \*dev;

};

struct mxc\_dispdrv\_handle {

struct mxc\_dispdrv\_driver \*drv;

};

//把新的drv挂接到dispdrv\_list

struct mxc\_dispdrv\_handle \*mxc\_dispdrv\_register(struct mxc\_dispdrv\_driver \*drv)

{

struct mxc\_dispdrv\_entry \*new;

new = kzalloc(sizeof(struct mxc\_dispdrv\_entry), GFP\_KERNEL);

new->drv = drv;

list\_add\_tail(&new->list, &dispdrv\_list);//把新的drv挂接到dispdrv\_list

return (struct mxc\_dispdrv\_handle \*)new;

}

//在dispdrv\_list查找匹配的驱动

struct mxc\_dispdrv\_handle \*mxc\_dispdrv\_gethandle(char \*name,struct mxc\_dispdrv\_setting \*setting)

{

struct mxc\_dispdrv\_entry \*entry;

list\_for\_each\_entry(entry, &dispdrv\_list, list) {

if (!strcmp(entry->drv->name, name) && (entry->drv->init)) {

//调用 lcdif\_init

ret = entry->drv->init((struct mxc\_dispdrv\_handle \*)entry, setting);

if (ret >= 0) {

entry->active = true;

found = 1;

break;

}

}

}

mutex\_unlock(&dispdrv\_lock);

//找到返回

return found ? (struct mxc\_dispdrv\_handle \*)entry:ERR\_PTR(-ENODEV);

}

int mxc\_dispdrv\_setdata(struct mxc\_dispdrv\_handle \*handle, void \*data)

{

struct mxc\_dispdrv\_entry \*entry = (struct mxc\_dispdrv\_entry \*)handle;

entry->priv = data;

}

void \*mxc\_dispdrv\_getdata(struct mxc\_dispdrv\_handle \*handle)

{

struct mxc\_dispdrv\_entry \*entry = (struct mxc\_dispdrv\_entry \*)handle;

return entry->priv;//就是lcdif

}

## 3.6 mxc\_ipuv3\_fb.c分析：

在mxc\_ipuv3\_fb.c文件中初始化mxcfb\_init函数注册mxcfb\_driver结构体，

当mxcfb\_driver结构体成员驱动name与平台设备层中name匹配相同时调用mxcfb\_probe初始化函数，imx6具体显示模块平台核心的驱动的工作主要是在mxcfb\_probe里完成。

static struct platform\_driver mxcfb\_driver = {

.driver = {

.name = MXCFB\_NAME,

},

.probe = mxcfb\_probe,

};

#### static struct fb\_ops mxcfb\_ops = {

#### .owner = THIS\_MODULE,

#### .fb\_set\_par = mxcfb\_set\_par,

#### .fb\_check\_var = mxcfb\_check\_var,

#### .fb\_setcolreg = mxcfb\_setcolreg,

#### .fb\_pan\_display = mxcfb\_pan\_display,

#### .fb\_ioctl = mxcfb\_ioctl,

#### .fb\_mmap = mxcfb\_mmap,

#### .fb\_fillrect = cfb\_fillrect,

#### .fb\_copyarea = cfb\_copyarea,

#### .fb\_imageblit = cfb\_imageblit,

#### .fb\_blank = mxcfb\_blank,

#### };

uboot传入的显示参数

displayArgs=video=mxcfb0:dev=lcd,CLAA-WVGA,if=RGB24,bpp=32 video=mxcfb1:off video=mxcfb2:off fbmem=48M

#### mxcfb\_probe函数

fbi = mxcfb\_init\_fbinfo(&pdev->dev, &mxcfb\_ops);

fbi = framebuffer\_alloc(sizeof(struct mxcfb\_info), dev);//分配framebuffer

info->par = p + fb\_info\_size;//指向fb\_info空间尾部

mxcfbi = (struct mxcfb\_info \*)fbi->par;// 指向mxcfb分配空间

fbi->fbops = ops; // mxcfb\_ops

ret = mxcfb\_option\_setup(pdev, fbi);//找不到直接退出程序

fb\_get\_options(name, &options)//找到 "mxcfb0"

if (!strncmp(opt, "dev=", 4)) //解析名字 lcd

memcpy(pdata->disp\_dev, opt + 4, strlen(opt) - 4);//uboot设置的名字”lcd”

if (!strncmp(opt, "if=", 3)) //解析数据格式RGB24

pdata->interface\_pix\_fmt = IPU\_PIX\_FMT\_RGB24;

if (!strncmp(opt, "bpp=", 4))

pdata->default\_bpp =simple\_strtoul(opt + 4, NULL, 0);//uboot传入值为32

fb\_pix\_fmt = bpp\_to\_pixfmt(pdata->default\_bpp);//pixfmt = IPU\_PIX\_FMT\_BGR32;

pixfmt\_to\_var(fb\_pix\_fmt, &fbi->var);//设置fbi可变参数中的颜色相关域

var->red = mxcfb\_pfmts[i].red;

var->green = mxcfb\_pfmts[i].green;

var->blue = mxcfb\_pfmts[i].blue;

var->transp = mxcfb\_pfmts[i].transp;

var->bits\_per\_pixel = mxcfb\_pfmts[i].bpp;

fb\_mode\_str = opt;//CLAA-WVGA

pdata->mode\_str = fb\_mode\_str;

ret = mxcfb\_dispdrv\_init(pdev, fbi);//传入的设备名字mxc\_sd\_fb.0

mxcfb\_dispdrv\_init

// disp\_dev=”lcd”,lcdif 驱动的名字也是”lcd”

mxcfbi->dispdrv = mxc\_dispdrv\_gethandle(disp\_dev, &setting);

//如果uboot传进来的参数名字与lcd\_driver里注册的名字匹配

if(!strcmp(entry->drv->name, name) && (entry->drv->init))

entry->drv->init //调用对应driver的init函数，这里就lcdif\_init了

mxcfbi->ipu\_di\_pix\_fmt = setting.if\_fmt;

mxcfbi->default\_bpp = setting.default\_bpp;

mxcfbi->ipu\_id = setting.dev\_id;

mxcfbi->ipu\_di = setting.disp\_id;//lcd\_init里面设置

ipu\_test\_set\_usage(mxcfbi->ipu\_id, mxcfbi->ipu\_di);//检测ipu是否可用

res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0);

if (res && res->start && res->end) {

fbi->fix.smem\_len = res->end - res->start + 1;

fbi->fix.smem\_start = res->start;

fbi->screen\_base = ioremap(fbi->fix.smem\_start, fbi->fix.smem\_len);

/\* Do not clear the fb content drawn in bootloader. \*/

if (!mxcfbi->late\_init)

memset(fbi->screen\_base, 0, fbi->fix.smem\_len);

}

mxcfbi->ipu = ipu\_get\_soc(mxcfbi->ipu\_id);

if (!g\_dp\_in\_use[mxcfbi->ipu\_id])

mxcfb\_register(fbi);//fb0

ret = mxcfb\_register(fbi);//fb1

register\_framebuffer(fbi);

platform\_set\_drvdata(pdev, fbi);

//static DEVICE\_ATTR(fsl\_disp\_property, 644, show\_disp\_chan, swap\_disp\_chan);

//static DEVICE\_ATTR(fsl\_disp\_dev\_property, S\_IRUGO, show\_disp\_dev, NULL);

device\_create\_file(fbi->dev, &dev\_attr\_fsl\_disp\_property);

device\_create\_file(fbi->dev, &dev\_attr\_fsl\_disp\_dev\_property);

fb\_prepare\_logo(fbi, 0);

fb\_show\_logo(fbi, 0);

entry->drv->init是如何调用ldb\_disp\_init？当ldb.c中ldb\_grv结构体中的name与uboot传进来的名字相匹配，就会调用.init，

static struct mxc\_dispdrv\_driver ldb\_drv = {

.name = DISPDRV\_LDB,

.init = ldb\_disp\_init,

.deinit = ldb\_disp\_deinit,

.setup = ldb\_disp\_setup,

};

db.c中ldb\_probe()

第三步调用mxcfb\_dispdrv\_init函数，它会调用在mxc\_dispdrv.c文件的mxc\_dispdrv\_gethandle函数；根据uboot传到内核的显示设备name（ldb），在dispdrv\_list链表中匹配获取对应的driver handle；这里是获取的是ldb的handle,它的注册是ldb\_probe()里的mxc\_dispdrv\_register函数的，它将自己添加到dispdrv\_list。通过entry->drv->init函数，当从uboot获取的显示设备名字和ldb driver里注册的名字match匹配时，就会执行ldb.c文件里的ldb\_disp\_init函数（具体见另一个文档，告诉你如何实现参数匹配），然后通过结构体fb\_videomode[]来设置屏幕的分辨率、刷新频率、上下左右页边距、行扫描脉宽、场扫描脉宽等参数。ldb\_disp\_init函数是LVDS驱动具体实现函数。

l

#### lcd 初始化函数分析

struct mxc\_lcdif\_data {

struct platform\_device \*pdev;

struct mxc\_dispdrv\_handle \*disp\_lcdif;

};

static struct fb\_videomode lcdif\_modedb[] = {

{

/\* 800x480 @ 57 Hz , pixel clk @ 27MHz \*/

"CLAA-WVGA", 95, 800, 480, 22434, 25, 75, 10, 10, 20, 10,

FB\_SYNC\_CLK\_LAT\_FALL,

FB\_VMODE\_NONINTERLACED,

0,},

{

/\* 1024\*768 @ 70 Hz , pixel clk @ 75MHz \*/

"TQ-VGA\_1024768", 70, 1024, 768, 13346, 144, 24, 29, 3, 136, 6,

FB\_SYNC\_HOR\_HIGH\_ACT | FB\_SYNC\_VERT\_HIGH\_ACT,

FB\_VMODE\_NONINTERLACED,

0,},

};

struct fb\_videomode {

const char \*name; //显示设备名，内核通过 u-boot 参数中 video 字段与此 name 作为匹配标识

u32 refresh; //刷新率

u32 xres; //行像素个数

u32 yres; //列像素个数

u32 pixclock; //每个像素时钟周期的长度，单位是皮秒

u32 left\_margin; //HBPD

u32 right\_margin; //HFPD

u32 upper\_margin; //VBPD

u32 lower\_margin; //VFBD

u32 hsync\_len; //行同步脉宽

u32 vsync\_len; //场同步脉宽

u32 sync; //同步信号极性

u32 vmode; //默认设置为 FB\_VMODE\_NONINTERLACED

u32 flag; //默认设置为 FB\_MODE\_IS\_DETAILED

};

setting.if\_fmt = plat\_data->interface\_pix\_fmt;// PU\_PIX\_FMT\_RGB24,

setting.dft\_mode\_str = plat\_data->mode\_str; //CLAA-WVGA

setting.default\_bpp = plat\_data->default\_bpp;// 32

setting.fbi = fbi;

lcdif\_init(struct mxc\_dispdrv\_handle \*disp,struct mxc\_dispdrv\_setting \*setting

struct mxc\_lcdif\_data \*lcdif = mxc\_dispdrv\_getdata(disp);

entry->priv//就是mxc\_lcdif\_data结构体

struct fsl\_mxc\_lcd\_platform\_data \*plat\_data= lcdif->pdev->dev.platform\_data;//初始化

struct fb\_videomode \*modedb = lcdif\_modedb;//上面定义的结构体

int modedb\_sz = lcdif\_modedb\_sz; //结构体项个数

setting->dev\_id = plat\_data->ipu\_id;//ipu\_id=0

setting->disp\_id = plat\_data->disp\_id;//disp\_id=0

// 查找一个有效的视频模式，ret>0表示找到

ret = fb\_find\_mode(&setting->fbi->var, setting->fbi, setting->dft\_mode\_str,

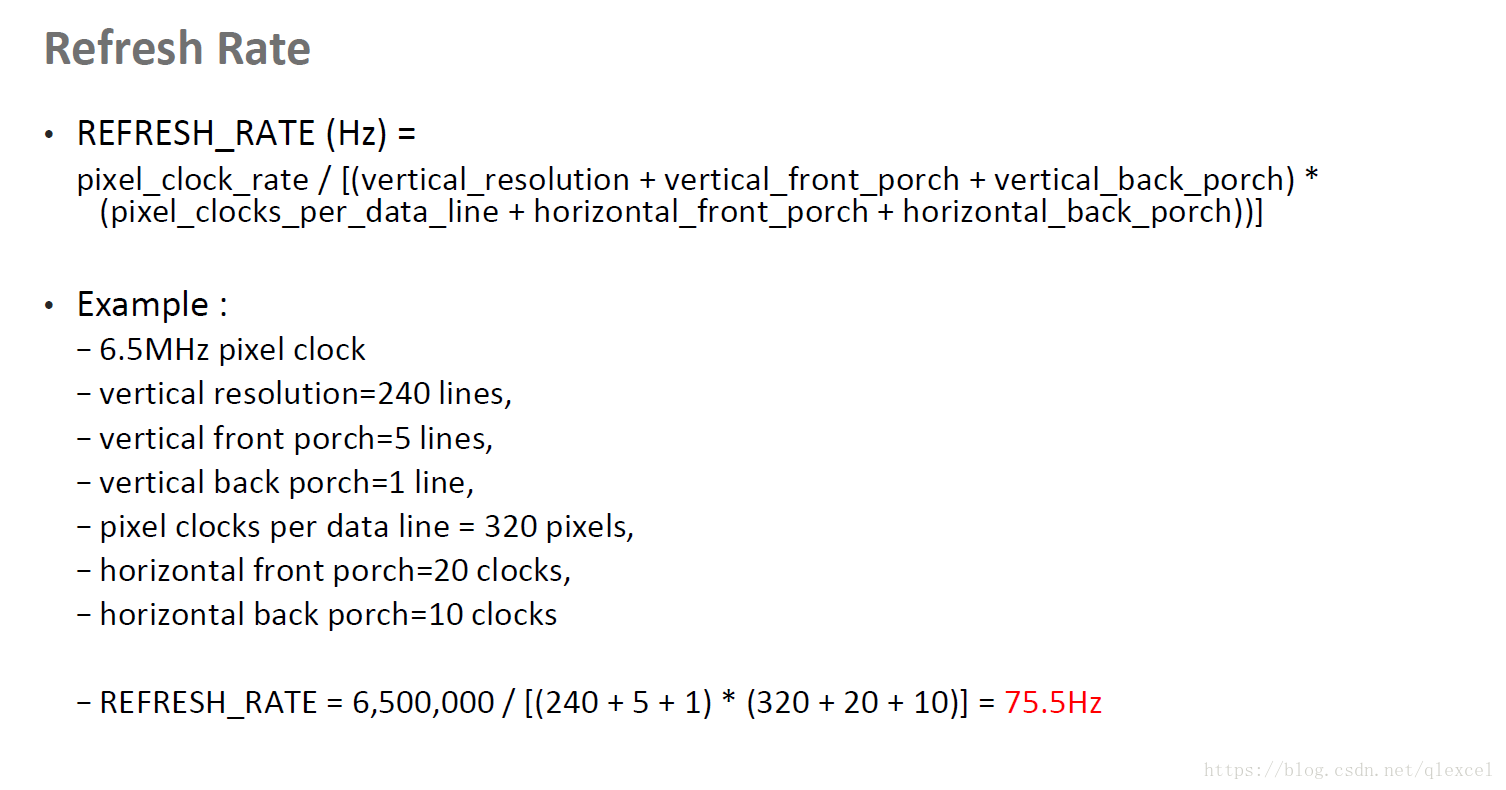
modedb, modedb\_sz, NULL, setting->default\_bpp);

if (!ret) {

fb\_videomode\_to\_var(&setting->fbi->var, &modedb[0]);

setting->if\_fmt = plat\_data->default\_ifmt;

}



# 名词解释vpu,ipu,gpu

Video graphics system[IPU, VPU and GPU]

IPU: Image Processing Unit   
• -- Display   
• -- Camera   
• -- Image Rotation, Inversion, Color Space Conversion   
• -- Image quality enhancement   
• -- Video/graphics combining

VPU: Video Processing Unit   
• -- Video encoding & decoding   
• -- Post-filtering   
• -- Rotation & inversion

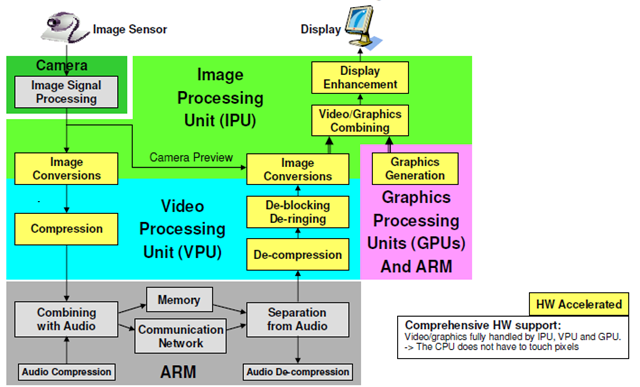
GPU: Graphics Processing Units   
• -- 2D (OpenVG 1.1)   
• -- 3D (OpenGL ES 2.0)

IPU：跟Camera，Display相关

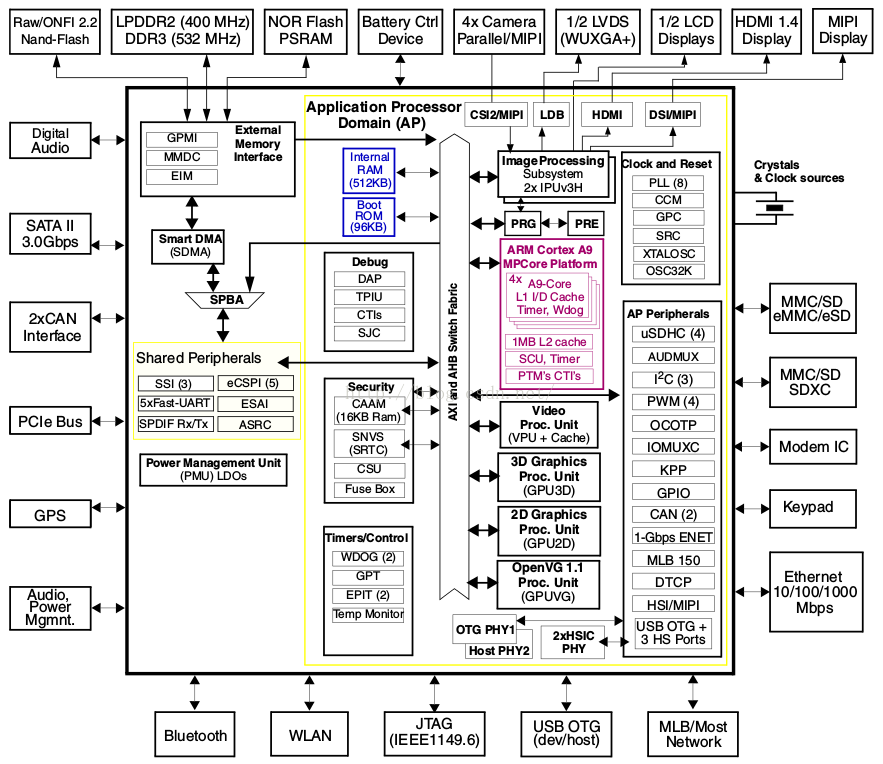
VPU：跟视频播放相关，主要包括HW codec和视频后处理等

VPU is a high performance multi-standard video codec IP that can perform the H.264 BP/MP/HP, VC-1 SP/MP/AP, MPEG-4 SP/ASP, Divx, MPEG-1/2, RV-8/9/10, and MJPEG decoding and encoding.

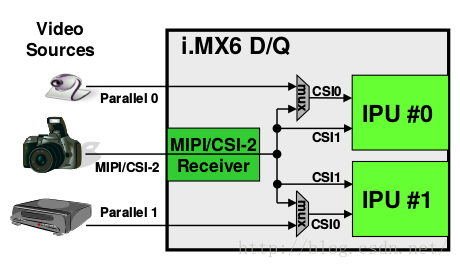
GPU：跟2D（OpenVG）和3D（OpenGL）相关

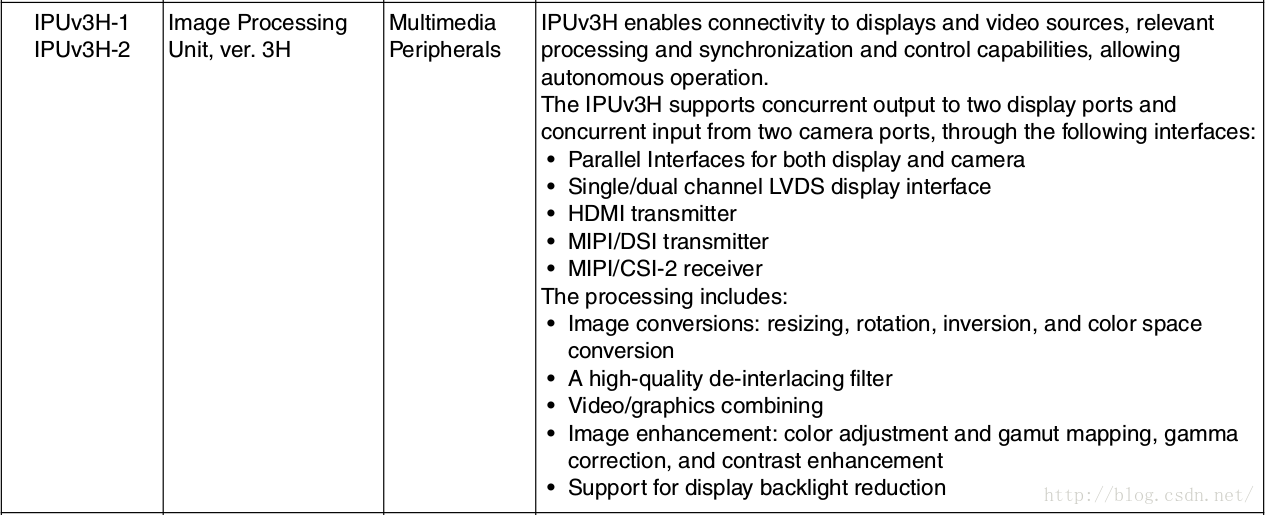


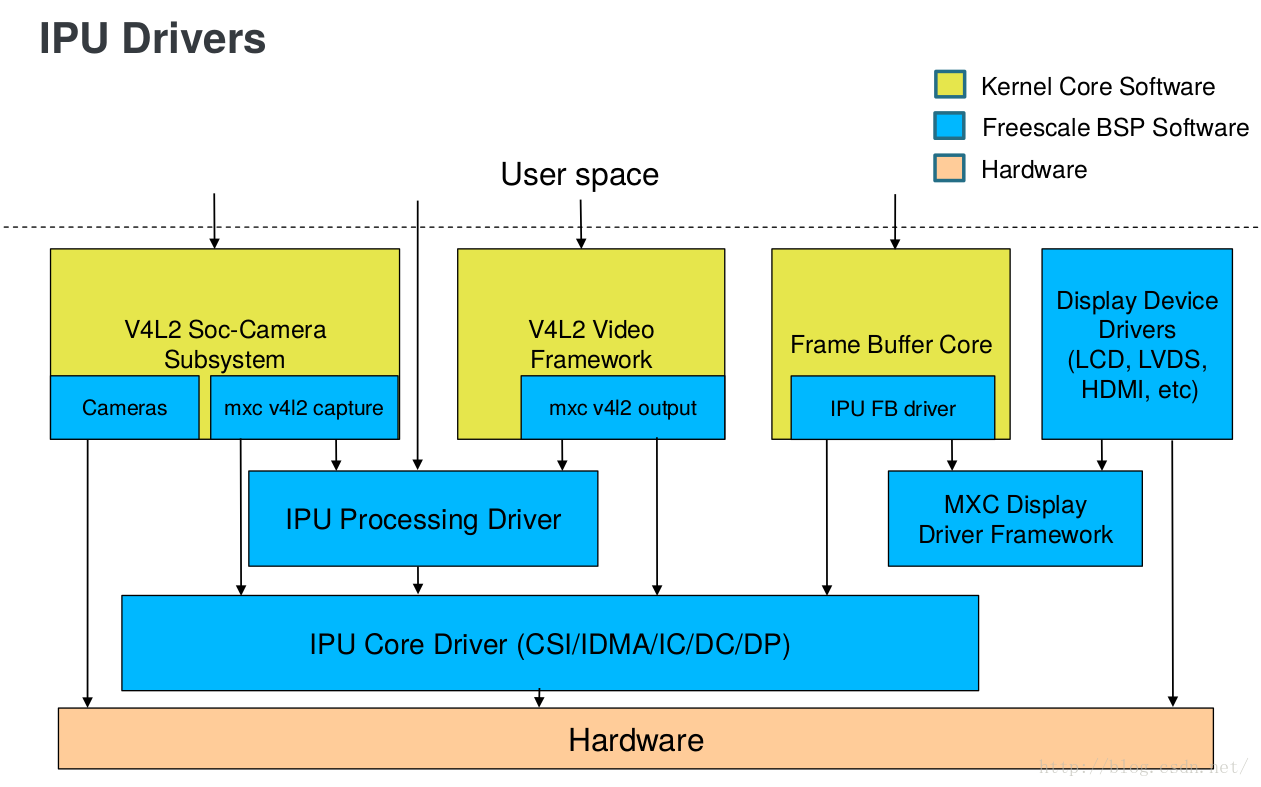
首先来看看imx6q整体系统框架图，看看IPU位于整个SOC系统中的位置



可以看出来，整个IPU挂接在AXI与AHB总线上面，通过总线，它可以与ARM，VPU，GPU和RAM等模块通信。另外，每个IPU有两个camera接口，如下所示：

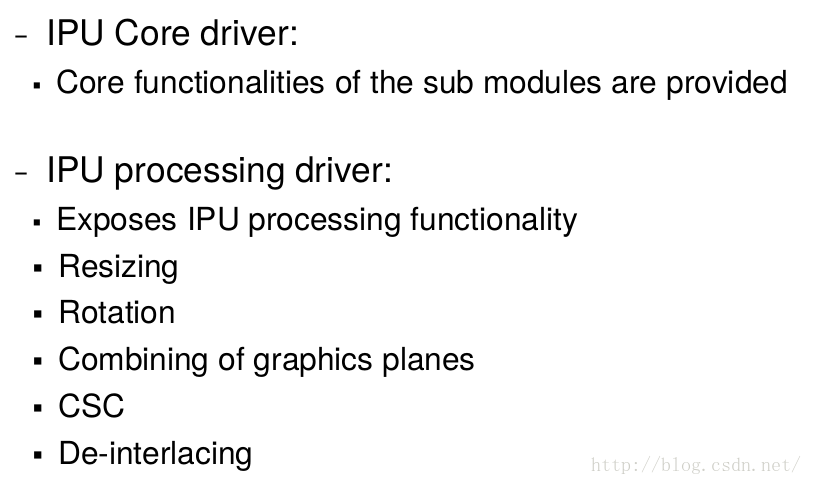


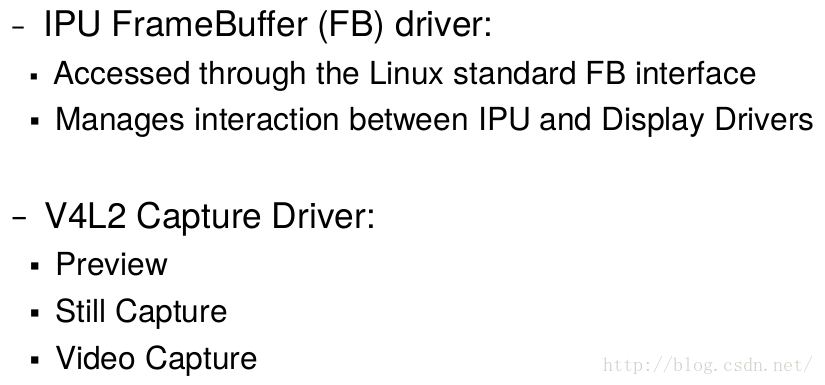




这个框架中可以看出来，对于一般的V4L2子系统，会通过IPU来与硬件通信。

图中各个模块的作用：





二）代码框架文件  
内核版本：3.14.52  
文件主要集中在：  
.../drivers/mxc/ipu3/                          描述  
    ipu\_device.c                         IPU tasks的操作等（IPU core driver）  
    ipu\_common.c                         底层的核心函数  
    ipu\_capture.c                        为ipu\_common.c提供函数操作  
    ipu\_disp.c                           IPU显示  
    ipu\_ic.c                             IC的操作函数  
    ipu\_param\_mem.h                      CPMEM设置的头文件，操作函数  
.../drivers/mxc/mipi/  
    mxc\_mipi\_csi2.c                      mipi摄像头的操作函数  
.../drivers/media/platform/mxc/  
    capture/mxc\_v4l2\_capture.c           为应用程序提供open，ioctl等操作函数  
    capture/v4l2-int-device.c            子系统模块注册等操作函数  
    capture/ipu\_prp\_enc.c                Pre-processing encoder驱动函数  
    capture/ipu\_csi\_enc.c                CSI设备驱动操作函数  
    capture/ipu\_still.c                  静态图片捕获函数  
    capture/ipu\_prp\_vf\_adc.c             Pre-processing view finder                                         (asynchronous)driver  
    capture/ipu\_prp\_vf\_sdc.c             Pre-processing view finder                                        (synchronous foreground) driver  
    capture/ipu\_fg\_overlay\_adc.c         前背景overlay异步驱动  
    capture/ipu\_bg\_overlay\_sdb.c         后背景overlay同步驱动  
    capture/ov5640.c ov5640\_mipi.c       ov5640.c子系统驱动  
  
（三） 分析步骤  
从以下几点分析这个驱动：  
1. 概述  
2. master和slave的匹配过程  
3. mxc\_v4l2\_capture.c分析  
    3.1 probe函数分析  
    3.2 vidioc\_int\_\* 类函数的调用过程  
    3.3 应用程序中函数调用的底层执行流程  
    3.4 mxc\_v4l\_open函数分析  
    3.5 cam\_data结构体详解  
    3.6 mxc\_v4l\_ioctl函数分析  
    3.7 ioctl函数中case的详细分析  
    3.8 poll函数的详细分析  
4. ipu\_common.c的分析  
    4.0 ipu\_soc,ipu\_channel\_t ,ipu\_channel\_params\_t结构体详解  
    4.1 ipu\_common.c分析---入口函数及probe函数分析  
    4.2 ipu\_common.c分析---ipu\_init\_channel函数的详细分析  
    4.3 ipu\_common.c分析---ipu\_init\_channel\_buffer函数的详细分析  
    4.4 ipu\_param\_mem.h头文件分析  
    4.5 ipu\_request\_irq函数详细分析  
    4.6 ipu\_enable\_channel函数详细分析  
    4.7 ipu\_capture.c分析  
5. 举例说明，通过mxc\_v4l2\_capture.c应用程序中的调用过程来一步一步分析  
    5.1 mxc\_v4l2\_capture.c应用程序追踪分析  
    5.2 应用程序和驱动程序中buffer的传输流程  
6. 子系统的分析  
    6.1 ipu\_prp\_enc.c详细分析  
    6.2 ipu\_csi\_enc.c详细分析  
    6.3 ipu\_still.c详细分析  
    6.4 ipu\_ic.c详细分析  
7. ipu\_device.c的分析  
    7.1 ipu\_device.c分析（一）---流程分析  
    7.2 ipu\_device.c分析（二）---具体函数分析  
8. 各个模块的分析  
    8.1 子模块分析之IDMAC  
    8.2 子模块分析之CSI  
    8.3 子模块分析之SMFC  
    8.4 子模块分析之IC

    8.5 子模块分析之VDIC

    8.6 子模块分析之CM

9. dts文件分析---以ov5640为例，修改dts文件使ov5640使用第二个IPU  
10. 对ipu\_init\_channel的理解，为以后添加channel做准备  
11. ov5640\_mipi.c分析