RK3128 驱动开发

说明手册

版本说明

日期	作者	描述
2015-7-19	龙腾	原始版本

销售与服务网络：

联系人：龙腾　　手机：18344023798

QQ: 1552381751　　邮箱：1552381751@qq.com

购买方式：淘宝 http://shop108029883.taobao.com

QQ 技术群

—嵌入式学习交流社区（2000 人）: 172455191

—Firefly 开源平台官方群（500 人）: 43113919

目录

1.1 ADC 使用

1.1.1 前言

FirePrime 开发板有一个 3 通道 (0/1/2)、10 比特精度的 SAR ADC (Successive Approximation Register，逐次逼近寄存器)，其中：

- ADCIN0: 在扩展板中引出
- ADCIN1: 内部作 Recovery 键检测
- ADCIN2: 在扩展板中引出

本文主要介绍 ADC 的基本使用方法。

1.1.2 配置 DTS 节点

FirePrime ADC 的 DTS 节点在 kernel/arch/arm/boot/dts/rk312x.dtsi 文件中定义，如下所示：

```
adc: adc@2006c000 {
        compatible = "rockchip,saradc";
        reg = <0x2006c000 0x100>;
        interrupts = <GIC_SPI 17 IRQ_TYPE_LEVEL_HIGH>;
        #io-channel-cells = <1>;
        io-channel-ranges;
        rockchip,adc-vref = <1800>;
        clock-frequency = <1000000>;
        clocks = <&clk_saradc>, <&clk_gates7 14>;
        clock-names = "saradc", "pclk_saradc";
        status = "disabled";
};
```

用户只需在 rk3128-fireprime.dts 文件中添加通道定义，并将其 status 改为 "okay" 即可：

```
&adc {
    status = "okay";
adc_test {
        status = "okay";
        compatible = "rk-adc-test";
        io-channels = <&adc 0>;
    };
};
```

此处添加一个测试设备 adc_test，为下面的驱动测试例程所使用。

ADC 的驱动源码为 drivers/iio/adc/rockchip_adc.c

1.1.3 获取 ADC 通道

```
struct iio_channel *chan;
chan = iio_channel_get(&pdev->dev, "adc0");
```

adc0 为通道名称，可用的通道列表在 rockchip_adc.c 中定义：

```
static const struct iio_chan_spec rk_adc_iio_channels[] = {
        ADC_CHANNEL(0, "adc0"),
        ADC_CHANNEL(1, "adc1"),
        ADC_CHANNEL(2, "adc2"),
        ADC_CHANNEL(6, "adc6"),
};
```

1.1.3.1 读取 AD 采集到的原始数据

```
int val, ret;
ret = iio_read_channel_raw(chan, &val);
```

调用 iio_read_channel_raw 函数读取 ADC 采集的原始数据并存入 val 中。

1.1.3.2 计算采集到的电压

使用标准电压将 AD 转换的值转换为用户所需要的电压值。其计算公式如下：

```
Vref / (2^n-1) = Vresult / raw
```

注：

1. Vref 为标准电压

2. n 为 AD 转换的位数
3. Vresult 为用户所需要的采集电压
4. raw 为 AD 采集的原始数据

例如，标准电压为 1.8V，AD 采集位数为 10 位，AD 采集到的原始数据为 568，则：

```
Vresult = (1800mv * 568) / 1023;
```

1.1.3.3 驱动测试例程

以下为完整的读取 ADC 的驱动例程：

```
#include <linux/module.h>
#include <linux/err.h>
#include <linux/platform_device.h>
#include <linux/types.h>
#include <linux/adc.h>
#include <linux/string.h>
#include <linux/iio/iio.h>
#include <linux/iio/machine.h>
#include <linux/iio/driver.h>
#include <linux/iio/consumer.h>

struct iio_channel *adc_test_channel;

static ssize_t show_measure(struct device *dev, struct device_attribute *attr,
            char *buf)
{
    int val, ret;
    size_t count = 0;

    ret = iio_read_channel_raw(adc_test_channel, &val);
    if (ret < 0) {
        count += sprintf(&buf[count], "read channel() error: %d\n", ret);
    } else {
        count += sprintf(&buf[count], "read channel(): %d\n", val);
    }
    return count;
}

static struct device_attribute measure_attr =
    __ATTR(measure, S_IRUGO, show_measure, NULL);

static int rk_adc_test_probe(struct platform_device *pdev)
```

```
{
    struct iio_channel *channels;
    channels = iio_channel_get_all(&pdev->dev);
    if (IS_ERR(channels)) {
        pr_err("get adc channels fails\n");
        goto err;
    }

   adc_test_channel =  &channels[0];

   if (device_create_file(&pdev->dev, &measure_attr)) {
        pr_err("device create file failed\n");
        goto err;
    }
  err:
    return -1;
}

static int rk_adc_test_remove(struct platform_device *pdev)
{
    device_remove_file(&pdev->dev, &measure_attr);
    iio_channel_release(adc_test_channel);
    adc_test_channel = NULL;
    return 0;
}

static const struct of_device_id rk_adc_test_match[] = {
    { .compatible = "rk-adc-test" },
    {},
};

MODULE_DEVICE_TABLE(of, rk_adc_test_match);

static struct platform_driver rk_adc_test_driver = {
    .probe      = rk_adc_test_probe,
    .remove     = rk_adc_test_remove,
    .driver = {
        .name = "rk-adc-test",
        .owner  = THIS_MODULE,
        .of_match_table = rk_adc_test_match,
    }
};

module_platform_driver(rk_adc_test_driver);
```

将以上源码保存为 drivers/iio/adc/rockchip-adc-test.c ，并在 drivers/iio/adc/Makefile 后加入：

```
obj-$(CONFIG_ROCKCHIP_ADC) += rk_adc_test.o
```

编译并烧写内核和 resource.img，启动后即可在终端下运行以下命令来读取 ADC0 的值：

```
while true; do cat /sys/devices/2006c000.adc/adc_test.*/measure; sleep 1; done
```

注意，该例程并没有采用 iio_channel_get 来获取通道，而是调用 iio_channel_get_all, 读取 io-channels 属性所声明的通道列表，后取首个通道：

```
    struct iio_channel *channels;
    channels = iio_channel_get_all(&pdev->dev);
    if (IS_ERR(channels)) {
        pr_err("get adc channels fails\n");
        goto err;
    }

adc_test_channel =  &channels[0];
```

1.1.4 ADC 常用函数接口

```
struct iio_channel *iio_channel_get(struct device *dev, const char
*consumer_channel);
```

功能：获取 iio 通道描述

参数：

1. dev: 使用该通道的设备描述指针
2. consumer_channel: 通道名称

```
void iio_channel_release(struct iio_channel *chan);
```

功能：释放 iio_channel_get 函数获取到的通道

参数：

1. chan：要被释放的通道描述指针

```
int iio_read_channel_raw(struct iio_channel *chan, int *val);
```

功能：读取 chan 通道 AD 采集的原始数据。

参数：

1. chan：要读取的采集通道指针
2. val：存放读取结果的指针

1.2 LED 使用

1.2.1 前言

FirePrime 开发板上有 2 个 LED 灯，如下表所示：

LED	GPIO
Blue	GPIO8_C7
Yellow	GPIO8_C6

可通过使用 LED 设备子系统或者直接操作 GPIO 控制该 LED。

1.2.2 以设备的方式控制 LED

标准的 Linux 专门为 LED 设备定义了 LED 子系统。 在 FirePrime 开发板中的两个 LED 均以设备的形式被定义。

用户可以通过 `/sys/class/leds/` 目录控制这两个 LED。

更详细的说明请参考 leds-class.txt 。

开发板上的 LED 的默认状态为：

- Blue: 系统上电时打开
- Yellow：用户自定义

用户可以通过 echo 向其 trigger 属性输入命令控制每一个 LED：

```
root@firefly:~ # echo none >/sys/class/leds/firefly:blue:power/trigger
root@firefly:~ # echo default-on >/sys/class/leds/firefly:blue:power/trigger
```

用户还可以使用 cat 命令获取 trigger 的可用值：

```
root@firefly:~ # cat /sys/class/leds/firefly:blue:power/trigger
none [ir-power-click] test_ac-online test_battery-charging-or-full
test_battery-charging
test_battery-full test_battery-charging-blink-full-solid test_usb-online mmc0
mmc1 mmc2
backlight default-on rfkill0 rfkill1 rfkill2
```

1.2.3 在内核中操作 LED

在内核中操作 LED 的步骤如下：

1、在 dts 文件中定义 LED 节点“leds”

在 kernel/arch/arm/boot/dts/rk3128-fireprime.dts 文件中定义 LED 节点，具体定义如下：

```
leds {
    compatible = "gpio-leds";
    power {
        label = "firefly:blue:power";
        linux,default-trigger = "ir-power-click";
    default-state = "on";
    gpios = <&gpio1 GPIO_C7 GPIO_ACTIVE_LOW>;
    };
    user {
        label = "firefly:yellow:user";
        linux,default-trigger = "ir-user-click";
    default-state = "off";
    gpios = <&gpio1 GPIO_C6 GPIO_ACTIVE_LOW>;
    };
};
```

注意：compatible 的值要跟 drivers/leds/leds-gpio.c 中的 .compatible 的值要保持一致。

2、在驱动文件包含头文件

```
#include <linux/leds.h>
```

3、在驱动文件中控制 LED。

（1）、定义 LED 触发器

```
DEFINE_LED_TRIGGER(ledtrig_ir_click);
```

（2）、注册该触发器

```
led_trigger_register_simple("ir-power-click", &ledtrig_ir_click);
```

（3）、控制 LED 的亮灭。

```
led_trigger_event(ledtrig_ir_click, LED_FULL); //亮
led_trigger_event(ledtrig_ir_click, LED_OFF); //灭
```

1.3 GPIO 使用

1.3.1 简介

GPIO，全称 General-Purpose Input/Output（通用输入输出），是一种软件运行期间能够动态配置和控制的通用引脚。

FirePrime 有 4 组 GPIO bank： GPIO0，GPIO1, GPIO2, GPIO3。每组又以 A0~A7, B0~B7, C0~C7, D0~D7 作为编号区分。

每个 GPIO 口除了通用输入输出功能外，还可能有其它复用功能，例如 GPIO1_C2，可以复用成以下功能之一：

- GPIO1_C2
- SDMMC0_D0
- UART2_TX

每个 GPIO 口的驱动电流、上下拉和重置后的初始状态都不尽相同，详细情况请参考《RK3128 规格书》中的 "RK3128 function IO description" 一章。

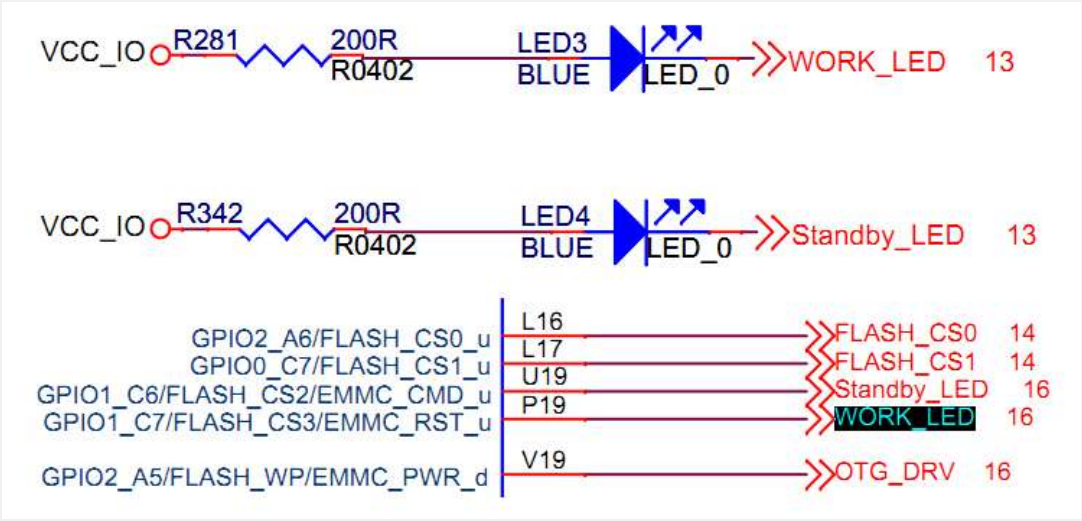
FirePrime 的 GPIO 驱动是在以下 pinctrl 文件中实现的：

```
kernel/drivers/pinctrl/pinctrl-rockchip.c
```

其核心是填充 GPIO bank 的方法和参数，并调用 gpiochip_add 注册到内核中。

1.3.2 使用

开发板有两个电源 LED 灯是 GPIO 口控制的，分别是：



从电路图上看，GPIO 口输出低电平时灯亮，高电平时灯灭。

另外，扩展槽上引出了几个空闲的 GPIO 口，分别是：

文件:Rk3128 gpios in extension.png

这几个 GPIO 口可以自定义作输入、输出使用。

1.3.2.1 输入输出

下面以电源 LED 灯的驱动为例，讲述如何在内核编写代码控制 GPIO 口的输出。

首先需要在 rk3128-firerpime.dts 中增加驱动的资源描述：

```
1.   firefly-led{
2.           compatible = "firefly,led";
3.           led-work = <&gpio1 GPIO_C6 GPIO_ACTIVE_LOW>;
4.           led-power = <&gpio1 GPIO_C7 GPIO_ACTIVE_LOW>;
5.           status = "okay";
6.   };
```

这里定义了两颗 LED 灯的 GPIO 设置：

```
led-work  GPIO1_C6  GPIO_ACTIVE_LOW
led-power GPIO1_C7  GPIO_ACTIVE_LOW
```

GPIO_ACTIVE_LOW 表示低电平有效(灯亮)，如果是高电平有效，需要替换为 GPIO_ACTIVE_HIGH 。

之后在驱动程序中加入对 GPIO 口的申请和控制则可：

```
#ifdef CONFIG_OF
```

```
#include <linux/of.h>
#include <linux/of_gpio.h>
#endif

static int firefly_led_probe(struct platform_device *pdev)
{
    int ret = -1;
    int gpio, flag;
          struct device_node *led_node = pdev->dev.of_node;

          gpio = of_get_named_gpio_flags(led_node, "led-power", 0, &flag);
          if (!gpio_is_valid(gpio)){
                    printk("invalid led-power: %d\n",gpio);
                    return -1;
          }
    if (gpio_request(gpio, "led_power")) {
                    printk("gpio %d request failed!\n",gpio);
        return ret;
          }
          led_info.power_gpio = gpio;
          led_info.power_enable_value = (flag == OF_GPIO_ACTIVE_LOW) ? 0 : 1;
          gpio_direction_output(led_info.power_gpio, !(led_info.power_enable_v
alue));
...
on_error:
    gpio_free(gpio);
}
```

of_get_named_gpio_flags 从设备树中读取 led-power 的 GPIO 配置编号和标志 ,gpio_is_valid 判断该 GPIO 编号是否有效 ,gpio_request 则申请占用该 GPIO。如果初始化过程出错 ,需要调用 gpio_free 来释放之前申请过且成功的 GPIO 。

调用 gpio_direction_output 就可以设置输出高还是低电平，因为是 GPIO_ACTIVE_LOW ，如果要灯亮，需要写入 0 。

实际中如果要读出 GPIO，需要先设置成输入模式，然后再读取值：

```
int val;
gpio_direction_input(your_gpio);
val = gpio_get_value(your_gpio);
```

下面是常用的 GPIO API 定义：

```
#include <linux/gpio.h>
#include <linux/of_gpio.h>
```

```
enum of_gpio_flags {
        OF_GPIO_ACTIVE_LOW = 0x1,
};

int of_get_named_gpio_flags(struct device_node *np, const char *propname,
                            int index, enum of_gpio_flags *flags);

int gpio_is_valid(int gpio);

int gpio_request(unsigned gpio, const char *label);

void gpio_free(unsigned gpio);

int gpio_direction_input(int gpio);

int gpio_direction_output(int gpio, int v)
```

1.3.2.2 复用

如何定义 GPIO 有哪些功能可以复用，在运行时又如何切换功能呢？以 I2C1 为例作简单的介绍。

查规格表可知，I2C1_SDA 与 I2C1_SCL 的功能定义如下：

Pad#	func0	func1
I2C4_SDA/GPIO7_C1	gpio7c1	i2c4tp_sda
I2C4_SCL/GPIO7_C2	gpio7c2	i2c4tp_scl

在 /kernel/arch/arm/boot/dts/rk312x.dtsi 里有：

```
        i2c1: i2c@20056000 {
                compatible = "rockchip,rk30-i2c";
                reg = <0x20056000 0x1000>;
                interrupts = <GIC_SPI 25 IRQ_TYPE_LEVEL_HIGH>;
                #address-cells = <1>;
                #size-cells = <0>;
                pinctrl-names = "default", "gpio";
                pinctrl-0 = <&i2c1_sda &i2c1_scl>;
                pinctrl-1 = <&i2c1_gpio>;
```

```
                gpios = <&gpio0 GPIO_A3 GPIO_ACTIVE_LOW>, <&gpio0 GPIO_A2
GPIO_ACTIVE_LOW>;
                clocks = <&clk_gates8 5>;
                rockchip,check-idle = <1>;
                status = "disabled";
        };
```

此处，跟复用控制相关的是 pinctrl- 开头的属性：

- pinctrl-names 定义了状态名称列表： default (i2c 功能) 和 gpio 两种状态。
- pinctrl-0 定义了状态 0 (即 default) 时需要设置的 pinctrl: i2c1_sda 和 i2c1_scl
- pinctrl-1 定义了状态 1 (即 gpio)时需要设置的 pinctrl: i2c1_gpio

这些 pinctrl 在 /kernel/arch/arm/boot/dts/rk312x-pinctrl.dtsi 中定义：

```
/ {
        pinctrl: pinctrl@20008000 {
                compatible = "rockchip,rk312x-pinctrl";
      ...
                gpio0_i2c1 {
                        i2c1_sda:i2c1-sda {
                                rockchip,pins = <I2C1_SDA>;
                                rockchip,pull = <VALUE_PULL_DEFAULT>;
                        };

                        i2c1_scl:i2c1-scl {
                                rockchip,pins = <I2C1_SCL>;
                                rockchip,pull = <VALUE_PULL_DEFAULT>;
                        };

                        i2c1_gpio: i2c1-gpio {
                                rockchip,pins = <FUNC_TO_GPIO(I2C1_SDA)>,
<FUNC_TO_GPIO(I2C1_SCL)>;
                                rockchip,pull = <VALUE_PULL_DEFAULT>;
                        };
                };
      ...
   }
 }
```

I2C1_SDA, I2C1_SCL 的定义在 /kernel/arch/arm/boot/dts/include/dt-bindings/pinctrl/rockchip-rk312x.h 中：

```
#define GPIO0_A3 0x0a30
#define I2C1_SDA 0x0a31
#define MMC1_CMD 0x0a32
```

```
#define GPIO0_A2 0x0a20
#define I2C1_SCL 0x0a21
```

FUN_TO_GPIO 的定义在 /kernel/arch/arm/boot/dts/include/dt-bindings/pinctrl/rockchip.h 中：

```
#define FUNC_TO_GPIO(m)           ((m) & 0xfff0)
```

也就是说 FUNC_TO_GPIO(I2C1_SDA) == GPIO0_A3, FUNC_TO_GPIO(I2C1_SCL) == GPIO7_A2 。

像 0x0a31 这样的值是有编码规则的：

```
0 a3 1
| |  `- func
| `---- offset
`------ bank
0x0a31 就表示 GPIO0_A3 func1, 即 I2C1_SDA 。
```

在复用时，如果选择了 "default" （即 i2c 功能),系统会应用 i2c1_sda 和 i2c1_scl 这两个 pinctrl，最终得将 GPIO0_A3 和 GPIO0_A2 两个针脚切换成对应的 i2c 功能；而如果选择了 "gpio" ，系统会应用 i2c1_gpio 这个 pinctrl，将 GPIO0_A3 和 GPIO0_A2 两个针脚还原为 GPIO 功能。

我们看看 i2c 的驱动程序 /kernel/drivers/i2c/busses/i2c-rockchip.c 是如何切换复用功能的：

```
static int rockchip_i2c_probe(struct platform_device *pdev)
{
        struct rockchip_i2c *i2c = NULL;
        struct resource *res;
        struct device_node *np = pdev->dev.of_node;
        int ret;
// ...
                i2c->sda_gpio = of_get_gpio(np, 0);
                if (!gpio_is_valid(i2c->sda_gpio)) {
                        dev_err(&pdev->dev, "sda gpio is invalid\n");
                        return -EINVAL;
                }
                ret = devm_gpio_request(&pdev->dev, i2c->sda_gpio,
dev_name(&i2c->adap.dev));
                if (ret) {
                        dev_err(&pdev->dev, "failed to request sda gpio\n");
                        return ret;
                }
                i2c->scl_gpio = of_get_gpio(np, 1);
                if (!gpio_is_valid(i2c->scl_gpio)) {
                        dev_err(&pdev->dev, "scl gpio is invalid\n");
                        return -EINVAL;
                }
                ret = devm_gpio_request(&pdev->dev, i2c->scl_gpio,
dev_name(&i2c->adap.dev));
```

```
                if (ret) {
                        dev_err(&pdev->dev, "failed to request scl gpio\n");
                        return ret;
                }
                i2c->gpio_state = pinctrl_lookup_state(i2c->dev->pins->p,
"gpio");
                if (IS_ERR(i2c->gpio_state)) {
                        dev_err(&pdev->dev, "no gpio pinctrl state\n");
                        return PTR_ERR(i2c->gpio_state);
                }
                pinctrl_select_state(i2c->dev->pins->p, i2c->gpio_state);
                gpio_direction_input(i2c->sda_gpio);
                gpio_direction_input(i2c->scl_gpio);
                pinctrl_select_state(i2c->dev->pins->p,
i2c->dev->pins->default_state);
// ...
}
```

首先是调用 of_get_gpio 取出设备树中 i2c1 结点的 gpios 属于所定义的两个 gpio:

```
gpios = <&gpio0 GPIO_A3 GPIO_ACTIVE_LOW>, <&gpio0 GPIO_A2 GPIO_ACTIVE_LOW>;
```

然后是调用 devm_gpio_request 来申请 gpio，接着是调用 pinctrl_lookup_state 来查找 “gpio” 状态，而默认状态 "default" 已经由框架保存到 i2c->dev-pins->default_state 中了。

最后调用 pinctrl_select_state 来选择是 "default" 还是 "gpio" 功能。

下面是常用的复用 API 定义：

```
#include <linux/pinctrl/consumer.h>

struct device {
//...
#ifdef CONFIG_PINCTRL
        struct dev_pin_info       *pins;
#endif
//...
};

struct dev_pin_info {
        struct pinctrl *p;
        struct pinctrl_state *default_state;
#ifdef CONFIG_PM
        struct pinctrl_state *sleep_state;
        struct pinctrl_state *idle_state;
#endif
};
```

```
struct pinctrl_state * pinctrl_lookup_state(struct pinctrl *p, const char *name);

int pinctrl_select_state(struct pinctrl *p, struct pinctrl_state *s);
```

1.4 PWM 使用

1.4.1 前言

FirePrime 开发板上有 4 路 PWM 输出，分别为 PWM0 ~ PWM3，其中：

- PWM0/GPIO0_D2: 在扩展口中引出
- PWM1/GPIO0_D3: 内部用作 AUX_DET 信号
- PWM2/GPIO0_D4: 内部用作 RTC_INT 信号
- PWM3 已经被红外收发器所使用

本章主要描述如何配置 PWM。

FirePrime 的 PWM 驱动为：

```
kernel/drivers/pwm/pwm-rockchip.c
```

1.4.2 配置步骤

配置 PWM 主要有以下三大步骤：配置 PWM DTS 节点、配置 PWM 内核驱动、控制 PWM 设备。

1.4.2.1 配置 PWM DTS 节点

在 kernel/arch/arm/boot/dts/rk312x.dtsi 定义了以下 PWM 节点，如下所示：

```
        pwm0: pwm@20050000 {
                compatible = "rockchip,rk-pwm";
                reg = <0x20050000 0x10>;
                #pwm-cells = <2>;
                pinctrl-names = "default";
                pinctrl-0 = <&pwm0_pin>;
                clocks = <&clk_gates7 10>;
                clock-names = "pclk_pwm";
                status = "disabled";
        };
```

```
pwm1: pwm@20050010 {
        compatible = "rockchip,rk-pwm";
        reg = <0x20050010 0x10>;
        #pwm-cells = <2>;
        pinctrl-names = "default";
        pinctrl-0 = <&pwm1_pin>;
        clocks = <&clk_gates7 10>;
        clock-names = "pclk_pwm";
        status = "disabled";
};

pwm2: pwm@20050020 {
        compatible = "rockchip,rk-pwm";
        reg = <0x20050020 0x10>;
        #pwm-cells = <2>;
        pinctrl-names = "default";
        pinctrl-0 = <&pwm2_pin>;
        clocks = <&clk_gates7 10>;
        clock-names = "pclk_pwm";
        status = "disabled";
};

pwm3: pwm@20050030 {
        compatible = "rockchip,rk-pwm";
        reg = <0x20050030 0x10>;
        #pwm-cells = <2>;
        pinctrl-names = "default";
        pinctrl-0 = <&pwm3_pin>;
        clocks = <&clk_gates7 10>;
        clock-names = "pclk_pwm";
        status = "disabled";
};

remotectl: pwm@20050030 {
        compatible = "rockchip,remotectl-pwm";
        reg = <0x20050030 0x10>;
        #pwm-cells = <2>;
        pinctrl-names = "default";
        pinctrl-0 = <&pwm3_pin>;
        clocks = <&clk_gates7 10>;
        clock-names = "pclk_pwm";
        remote_pwm_id = <3>;
        interrupts = <GIC_SPI 30 IRQ_TYPE_LEVEL_HIGH>;
```

```
            status = "okay";
        };
```

要使用 pwm0, 只需在 kernel/arch/arm/boot/dts/rk3128-fireprime.dts 加入：

```
&pwm0 {
        status = "okay";
};
```

1.4.2.2 配置 PWM 内核驱动

PWM 驱动位于文件 kernel/drivers/pwm/pwm-rockchip.c。

1.4.2.3 控制 PWM 设备

用户可在其它驱动文件中使用以上步骤生成的 PWM 节点。具体方法如下：

(1)、在要使用 PWM 控制的设备驱动文件中包含以下头文件：

```
#include <linux/pwm.h>
```

该头文件主要包含 PWM 的函数接口。

(2)、申请 PWM

使用

```
 struct pwm_device *pwm_request(int pwm_id, const char *label);
```

函数申请 PWM。例如：

```
struct pwm_device * pwm0 = NULL;
pwm0 = pwm_request(0, “backlight-pwm”);
```

参数 pwm_id 表示要申请 PWM 的通道，label 为该 PWM 所取的标签。

(3)、配置 PWM

使用

```
int pwm_config(struct pwm_device *pwm, int duty_ns, int period_ns);
```

配置 PWM 的占空比，例如：

```
pwm_config(pwm0, 500000, 1000000);
```

参数 pwm 为前一步骤申请的 pwm_device。duty_ns 为占空比激活的时长，单位为 ns。period_ns 为 PWM 周期，单位为 ns。

(4)、使能 PWM

函数

```
int pwm_enable(struct pwm_device *pwm);
```

用于使能 PWM，例如：

```
pwm_enable(pwm0);
```

参数 pwm 为要使能的 pwm_device。

1.4.2.4 控制 PWM 输出主要使用以下接口函数：

```
struct pwm_device *pwm_request(int pwm_id, const char *label);
```

功能：用于申请 pwm

参数：

pwm_id：要申请的 pwm 通道。

label: 为该申请的 pwm 所取的标签。

```
void pwm_free(struct pwm_device *pwm);
```

功能：用于释放所申请的 pwm

参数：

pwm：所要释放的 pwm 结构体

```
int pwm_config(struct pwm_device *pwm, int duty_ns, int
period_ns);
```

功能：用于配置 pwm 的占空比

参数：

pwm: 所要配置的 pwm

duty_ns：pwm 的占空比激活的时长，单位 ns

period_ns：pwm 占空比周期，单位 ns

```
int pwm_enable(struct pwm_device *pwm);
```

功能：使能 pwm

参数：

pwm：要使能的 pwm

```
void pwm_disable(struct pwm_device *pwm);
```

功能：禁止 pwm

参数：

pwm：要禁止的 pwm

1.5 IIC 使用

1.5.1 前言

FirePrime 开发板上有 4 个片上 I²C 控制器。本文主要描述如何在该开发板上配置 I²C。

配置 I²C 可分为两大步骤：

1. 定义和注册 I²C 设备
2. 定义和注册 I²C 驱动

下面以配置 lt8641ex 为例。

1.5.2 定义和注册 I2C 设备

在注册 I²C 设备时，需要结构体 i2c_client 来描述 I²C 设备。然而在标准 Linux 中，用户只需要提供相应的 I²C 设备信息，Linux 就会根据所提供的信息构造 i2c_client 结构体。

用户所提供的 I²C 设备信息以节点的形式写到 dts 文件中，如下所示：

```
&i2c0 {
      status = "okay";
     lt8641ex@3f {
            compatible = "firefly,lt8641ex";
            gpio-sw = <&gpio7 GPIO_B2 GPIO_ACTIVE_LOW>;
            reg = <0x3f>;
     };
```

```
};
```

1.5.3 定义和注册 I2C 驱动

1.5.3.1 定义 I²C 驱动

在定义 I²C 驱动之前，用户首先要定义变量 of_device_id 和 i2c_device_id 。

of_device_id 用于在驱动中调用 dts 文件中定义的设备信息，其定义如下所示：

```
static const struct of_device_id of_rk_lt8641ex_match[] = {
    { .compatible = "firefly,lt8641ex" },
    { /* Sentinel */ }
};
```

定义变量 i2c_device_id：

```
static const struct i2c_device_id lt8641ex_id[] = {
    { lt8641ex, 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, lt8641ex_id);
```

i2c_driver 如下所示：

```
static struct i2c_driver lt8641ex_device_driver = {
    .driver     = {
        .name   = "lt8641ex",
        .owner  = THIS_MODULE,
        .of_match_table = of_rk_lt8641ex_match,
    },
    .probe          = lt8641ex_probe,
    .remove         = lt8641ex_remove,
    .suspend        = lt8641ex_suspend,
    .resume         = lt8641ex_resume,
    .id_table       = lt8641ex_id,
};
```

注：变量 id_table 指示该驱动所支持的设备。

1.5.3.2 注册 I²C 驱动

使用 i2c_add_driver 函数注册 I²C 驱动。

```
    i2c_add_driver(&lt8641ex_device_driver);
```

在调用 i2c_add_driver 注册 I²C 驱动时，会遍历 I²C 设备，如果该驱动支持所遍历到的设备，则会调用该驱动的 probe 函数。

1.5.3.3 通过 I²C 收发数据

在注册好 I² C 驱动后，即可进行 I²C 通讯。

- 向从机发送信息

```
static int i2c_master_reg8_send(const struct i2c_client *client, const char reg,
const char *buf, int count, int scl_rate)
{
    struct i2c_adapter *adap=client->adapter;
    struct i2c_msg msg;
    int ret;
    char *tx_buf = (char *)kzalloc(count + 1, GFP_KERNEL);
    if(!tx_buf)
        return -ENOMEM;
    tx_buf[0] = reg;
    memcpy(tx_buf+1, buf, count);

    msg.addr = client->addr;
    msg.flags = client->flags;
    msg.len = count + 1;
    msg.buf = (char *)tx_buf;
    msg.scl_rate = scl_rate;

    ret = i2c_transfer(adap, &msg, 1);
    kfree(tx_buf);
    return (ret == 1) ? count : ret;

}
```

- 向从机读取信息

```
static int i2c_master_reg8_recv(const struct i2c_client *client, const char reg,
char *buf, int count, int scl_rate)
{
    struct i2c_adapter *adap=client->adapter;
    struct i2c_msg msgs[2];
    int ret;
    char reg_buf = reg;

    msgs[0].addr = client->addr;
    msgs[0].flags = client->flags;
```

```
    msgs[0].len = 1;
    msgs[0].buf = &reg_buf;
    msgs[0].scl_rate = scl_rate;

    msgs[1].addr = client->addr;
    msgs[1].flags = client->flags | I2C_M_RD;
    msgs[1].len = count;
    msgs[1].buf = (char *)buf;
    msgs[1].scl_rate = scl_rate;

    ret = i2c_transfer(adap, msgs, 2);

    return (ret == 2)? count : ret;
}
```

注:

1. msgs[0] 是要向从机发送的信息，告诉从机主机要读取信息。
2. msgs[1] 是主机向从机读取到的信息。

至此，主机可以使用函数 i2c_master_reg8_send 和 i2c_master_reg8_recv 和从机进行通讯。

- 实际通讯示例

例如主机和 LT8641EX 通讯，主机向 LT8641EX 发送信息，设置 LT8641EX 使用通道 1：

```
    int channel=1;
    i2c_master_reg8_send(g_lt8641ex->client, 0x00, &channel,1, 100000);
```

注：通道寄存器的地址为 0x00。

主机向从机 LT8641EX 读取当前使用的通道：

```
    u8 ch = 0xfe;
    i2c_master_reg8_recv(g_lt8641ex->client, 0x00, &ch,1, 100000);
```

注：ch 用于保存读取到的信息。

1.6 UART 使用

1.6.1 板载资源介绍

FirePrime 开发板内置 3 路 UART，分别为 uart0，uart1，uart2。

uart0 用于蓝牙数据传输，如果要使用 uart0，必须关掉蓝牙，才可以使用扩展槽上的 UART0 针脚。

uart1, 因为存在以下复用：

1. BT_HOST_WAKE/SPI_TXD/UART1_TX
2. BT_WAKE/SPI_RXD/UART1_RX
3. WIFI_REG_ON/SPI_CSN0/UART1_RTS

若要使用 uart1, 必须关掉蓝牙和 SPI 功能，这样才可以使用扩展槽上的 SPI_RX 和 SPI_TX 针作 UART1_RX 和 UART1_TX 使用。

uart2 一般用做调试串口，但同样存在复用，也就是说 TF 卡与调试串口不可以同时使用：

1. SDMMC_D0/UART2_TX
2. SDMMC_D1/UART2_RX

uart1 和 uart2 有 32 字节的 FIFO 收发缓冲区，uart0 则要有双 64 字节的 FIFO 用作蓝牙数据收发。所有 uart 均支持 5 位、6 位、7 位、8 位数据收发和 DMA 操作。

1.6.2 配置 DTS 节点

文件 kernel/arch/arm/boot/dts/rk312x.dtsi 中已经有 uart 相关节点定义，如下所示：

```
        uart0: serial@20060000 {
                compatible = "rockchip,serial";
                reg = <0x20060000 0x100>;
                interrupts = <GIC_SPI 20 IRQ_TYPE_LEVEL_HIGH>;
                clock-frequency = <24000000>;
                clocks = <&clk_uart0>, <&clk_gates8 0>;
                clock-names = "sclk_uart", "pclk_uart";
                reg-shift = <2>;
                reg-io-width = <4>;
                dmas = <&pdma 2>, <&pdma 3>;
                #dma-cells = <2>;
                pinctrl-names = "default";
                pinctrl-0 = <&uart0_xfer &uart0_cts &uart0_rts>;
                status = "disabled";
        };

        uart1: serial@20064000 {
                compatible = "rockchip,serial";
                reg = <0x20064000 0x100>;
                interrupts = <GIC_SPI 21 IRQ_TYPE_LEVEL_HIGH>;
```

```
                clock-frequency = <24000000>;
                clocks = <&clk_uart1>, <&clk_gates8 1>;
                clock-names = "sclk_uart", "pclk_uart";
                reg-shift = <2>;
                reg-io-width = <4>;
                dmas = <&pdma 4>, <&pdma 5>;
                #dma-cells = <2>;
                pinctrl-names = "default";
                pinctrl-0 = <&uart1_xfer &uart1_cts &uart1_rts>;
                status = "disabled";
        };
```

1.6.2.1 配置 uart0

用户只需在 kernel/arch/arm/boot/dts/rk3128-fireprime.dts 文件中打开 uart0 ，并关掉蓝牙，如下所示：

```
&uart0 {
        status = "okay";
        dma-names = "!tx", "!rx";
        pinctrl-0 = <&uart0_xfer &uart0_cts>;
};

        wireless-bluetooth {
                compatible = "bluetooth-platdata";
      ...
      status = "disabled";
   };
```

1.6.2.2 配置 uart1

用户只需在 kernel/arch/arm/boot/dts/rk3128-fireprime.dts 文件中打开 uart1 ,并关掉蓝牙和 SPI，如下所示：

```
//...
        wireless-bluetooth {
                compatible = "bluetooth-platdata";
      ...
      status = "disabled";
   };
//...
&spi0 {
     status = "disabled";
};
&uart1 {
        status = "okay";
        dma-names = "!tx", "!rx";
```

```
        pinctrl-0 = <&uart1_xfer &uart1_cts>;
};
```

1.6.3 编译并烧写内核

将串口驱动编译到内核中，在 kernel 目录下执行如下命令：

```
make rk3128-fireprime.img
```

把 kernel 目录下生成的 kernel.img 和 resource.img 烧录到开发板中即可。

1.6.4 测试串口通讯

配置好串口后，用户可以通过主机的 USB 转串口适配器向开发板的串口收发数据，以 uart0 为例，步骤如下：

(1) 连接硬件

将开发板 uart0 的 TX、RX、GND 引脚分别和主机串口适配器的 RX、TX、GND 引脚相连。

注意：如果是使用 Firefly 的串口适配器，则是 TX 对 TX，RX 对 RX 连接。

(2) 打开主机的串口终端

在终端打开 kermit，并设置波特率：

```
$ sudo kermit
C-Kermit> set line /dev/ttyUSB0
C-Kermit> set speed 9600
C-Kermit> set flow-control none
C-Kermit> connect
```

- /dev/ttyUSB0 为 USB 转串口适配器的设备文件
- uart0 的波特率默认为 9600

(3) 发送数据

uart0 的设备文件为 /dev/ttyS0。在设备上运行下列命令：

```
echo firefly uart test... > /dev/ttyS0
```

主机中的串口终端即可接收到字符串“firefly uart test...”

(4) 接收数据

首先在设备上运行下列命令：

```
cat /dev/ttyS0
```

然后在主机的串口终端输入字符串 “Firefly uart test...” ，设备端即可见到相同的字符串。

要改变 uart0 的波特率，例如 115200，可以运行以下命令：

```
stty -F /dev/ttyS0 115200
```

1.7 IR 使用

1.7.1 红外遥控配置

FirePrime 开发板上使用红外收发传感器 IR (在 USB OTG 接口和音频接口之间)实现遥控功能。本文主要描述在开发板上如何配置红外遥控器。

其配置步骤可分为两个部分：

1. 修改内核驱动：内核空间修改，Linux 和 Android 都要修改这部分的内容。
2. 修改键值映射：用户空间修改（仅限 Android 系统）。

1.7.2 内核驱动

在 Linux 内核中，IR 驱动仅支持 NEC 编码格式。以下是在内核中配置红外遥控的方法。

所涉及到的文件：

- dts 配置文件：kernel/arch/arm/boot/dts/rk3128-fireprime.dts
- 驱动源文件： kernel/drivers/input/remotectl/rk_pwm_remotectl.c

1.7.2.1 添加 IR 及对应键值表

```
&remotectl {
        handle_cpu_id = <1>;
        ir_key1{
             rockchip,usercode = <0xff00>;
             rockchip,key_table =
                   <0xeb   KEY_POWER>,
                   <0xa3   250>,
                   <0xec   KEY_MENU>,
                   <0xfc   KEY_UP>,
                   <0xfd   KEY_DOWN>,
                   <0xf1   KEY_LEFT>,
                   <0xe5   KEY_RIGHT>,
                   <0xf8   KEY_REPLY>,
                   <0xb7   KEY_HOME>,
                   <0xfe   KEY_BACK>,
                   <0xa7   KEY_VOLUMEDOWN>,
                   <0xf4   KEY_VOLUMEUP>;
        };
        ir_key2{
                rockchip,usercode = <0xff00>;
                rockchip,key_table =
                        <0xf9    KEY_HOME>,
                        <0xbf    KEY_BACK>,
                        <0xfb    KEY_MENU>,
                        <0xaa    KEY_REPLY>,
                        <0xb9    KEY_UP>,
                        <0xe9    KEY_DOWN>,
                        <0xb8    KEY_LEFT>,
                        <0xea    KEY_RIGHT>,
                        <0xeb    KEY_VOLUMEDOWN>,
                        <0xef    KEY_VOLUMEUP>,
                        <0xf7    KEY_MUTE>,
                        <0xe7    KEY_POWER>,
                        <0xfc    KEY_POWER>,
                        <0xa9    KEY_VOLUMEDOWN>,
                        <0xa8    KEY_VOLUMEDOWN>,
                        <0xe0    KEY_VOLUMEDOWN>,
                        <0xa5    KEY_VOLUMEDOWN>,
                        <0xab    183>,
                        <0xb7    388>,
                        <0xf8    184>,
                        <0xaf    185>,
                        <0xed    KEY_VOLUMEDOWN>,
```

```
                        <0xee    186>,
                        <0xb3    KEY_VOLUMEDOWN>,
                        <0xf1    KEY_VOLUMEDOWN>,
                        <0xf2    KEY_VOLUMEDOWN>,
                        <0xf3    KEY_SEARCH>,
                        <0xb4    KEY_VOLUMEDOWN>,
                        <0xbe    KEY_SEARCH>;
        };
        ir_key3{
                rockchip,usercode = <0x1dcc>;
                rockchip,key_table =
                        <0xee    KEY_REPLY>,
                        <0xf0    KEY_BACK>,
                        <0xf8    KEY_UP>,
                        <0xbb    KEY_DOWN>,
                        <0xef    KEY_LEFT>,
                        <0xed    KEY_RIGHT>,
                        <0xfc    KEY_HOME>,
                        <0xf1    KEY_VOLUMEUP>,
                        <0xfd    KEY_VOLUMEDOWN>,
                        <0xb7    KEY_SEARCH>,
                        <0xff    KEY_POWER>,
                        <0xf3    KEY_MUTE>,
                        <0xbf    KEY_MENU>,
                        <0xf9    0x191>,
                        <0xf5    0x192>,
                        <0xb3    388>,
                        <0xbe    KEY_1>,
                        <0xba    KEY_2>,
                        <0xb2    KEY_3>,
                        <0xbd    KEY_4>,
                        <0xf9    KEY_5>,
                        <0xb1    KEY_6>,
                        <0xfc    KEY_7>,
                        <0xf8    KEY_8>,
                        <0xb0    KEY_9>,
                        <0xb6    KEY_0>,
                        <0xb5    KEY_BACKSPACE>;
        };
};
```

注：

1. usercode：用户码，每个 IR 都有一个对应的用户码；

2. key_table：IR 按键的扫描码和按键代码映射表。

1.7.2.2 如何获取用户码和 IR 键值

在 remotectl_do_something 函数中获取用户码和键值：

```
        case RMC_USERCODE: {
                if ((RK_PWM_TIME_BIT1_MIN < ddata->period) &&
                    (ddata->period < RK_PWM_TIME_BIT1_MAX))
                        ddata->scandata |= (0x01 << ddata->count);
                ddata->count++;
                if (ddata->count == 0x10) {
                        DBG_CODE("USERCODE=0x%x\n", ddata->scandata);
                        if (remotectl_keybd_num_lookup(ddata)) {
                                ddata->state = RMC_GETDATA;
                                ddata->scandata = 0;
                                ddata->count = 0;
                        } else {
                                if (rk_remote_print_code){
                                        ddata->state = RMC_GETDATA;
                                        ddata->scandata = 0;
                                        ddata->count = 0;
                                } else
                                        ddata->state = RMC_PRELOAD;
                        }
                }
        }
```

注：用户可以使用 DBG_CODE() 函数打印用户码。

向 remotectl_button 数组添加用户码和键值：

```
        case RMC_GETDATA: {
          #ifdef CONFIG_FIREFLY_POWER_LED
          mod_timer(&timer_led,jiffies + msecs_to_jiffies(50));
          remotectl_led_ctrl(0);
          #endif

              if(!get_state_remotectl() && (ddata->keycode != KEY_POWER))
              {
                     ledtrig_ir_activity();
              }
                  if ((RK_PWM_TIME_BIT1_MIN < ddata->period) &&
                      (ddata->period < RK_PWM_TIME_BIT1_MAX))
                           ddata->scandata |= (0x01<<ddata->count);
                  ddata->count++;
```

```
                if (ddata->count < 0x10)
                        return;
                DBG_CODE("RMC_GETDATA=%x\n", (ddata->scandata>>8));
                if ((ddata->scandata&0x0ff) ==
                    ((~ddata->scandata >> 8) & 0x0ff)) {
                        if (remotectl_keycode_lookup(ddata)) {
                                ddata->press = 1;
                                input_event(ddata->input, EV_KEY,
                                            ddata->keycode, 1);
                                input_sync(ddata->input);
                                ddata->state = RMC_SEQUENCE;
                        } else {
                                ddata->state = RMC_PRELOAD;
                        }
                } else {
                        ddata->state = RMC_PRELOAD;
                }
        }
        break;
```

注：用户可以使用 DBG_CODE() 函数打印键值。

1.7.2.3 将 IR 驱动编译进内核

检查内核配置，确保以下选项选上：

```
CONFIG_ROCKCHIP_REMOTECTL=y
CONFIG_ROCKCHIP_REMOTECTL_PWM=y
```

否则，在 kernel 路径下使用 make menuconfig ，按照如下方法将 IR 驱动选中，并保存退出。

```
Device Drivers
  --->Input device support
  -----> [*]   rkxx remotectl
  -------->[*]   rkxx remoctrl pwm0 capture.
```

执行 make 命令即可将该驱动编进内核。

1.7.3 Android 键值映射

文件 /system/usr/keylayout/20050030_pwm.kl 用于将 Linux 层获取的键值映射到 Android 上对应的键值。用户可以添加或者修改该文件的内容以实现不同的键值映射。

该文件内容如下所示：

```
key 28   ENTER
key 116   POWER           WAKE
key 158   BACK
key 139   MfENU
key 217   SEARCH
key 232   DPAD_CENTER
key 108   DPAD_DOWN
key 103   DPAD_UP
key 102   HOME
key 105   DPAD_LEFT
key 106   DPAD_RIGHT
key 115   VOLUME_UP
key 114   VOLUME_DOWN
key 143   NOTIFICATION     WAKE
key 113   VOLUME_MUTE
key 388   TV_KEYMOUSE_MODE_SWITCH
key 400   TV_MEDIA_MULT_BACKWARD
key 401   TV_MEDIA_MULT_FORWARD
key 402   TV_MEDIA_PLAY_PAUSE
key 64    TV_MEDIA_PLAY
key 65    TV_MEDIA_PAUSE
key 66    TV_MEDIA_STOP
key 67    TV_MEDIA_REWIND
key 68    TV_MEDIA_FAST_FORWARD
key 87    TV_MEDIA_PREVIOUS
key 88    TV_MEDIA_NEXT
key 250   FIREFLY_RECENT
```

注：通过 adb 修改该文件重启后即可生效。

1.8 Camera 使用

1.8.1 前言

FirePrime 开发板上有 CIF 接口，支持 CIF 双摄像头。

在 kernel/drivers/media/video 目录里有以下 CIF 摄像头的驱动，：

- gc0307
- gc0308

- gc0309
- gc0328
- gc0329
- gc2015
- gc2035
- gt2005
- hm2057
- hm5065
- mt9p111
- nt99160
- nt99240
- ov2659
- ov5640
- sp0838
- sp2518

本章主要描述如何配置 CIF。

1.8.2 配置步骤

1.8.2.1 加入传感器的连接信息

在 kernel/arch/arm/boot/dts/rk3128-cif-sensor.dtsi 定义摄像头列表：

```
        rk3128_cif_sensor: rk3128_cif_sensor{
                        compatible = "rockchip,sensor";
                        status = "disabled";
                        CONFIG_SENSOR_POWER_IOCTL_USR              = <1>;
                        CONFIG_SENSOR_RESET_IOCTL_USR              = <0>;
                        CONFIG_SENSOR_POWERDOWN_IOCTL_USR = <0>;
                        CONFIG_SENSOR_FLASH_IOCTL_USR              = <0>;
                        CONFIG_SENSOR_AF_IOCTL_USR
        = <0>;
// ... skip some modules
gc0329{
                        is_front = <1>;
                        rockchip,powerdown = <&gpio3 GPIO_D7
GPIO_ACTIVE_HIGH>;
                        pwdn_active = <gc0329_PWRDN_ACTIVE>;
                        #rockchip,power = <>;
                        pwr_active = <PWR_ACTIVE_HIGH>;
```

```
                        #rockchip,reset = <>;
                        #rst_active = <>;
                        #rockchip,flash = <>;
                        #rockchip,af = <>;
                        mir = <0>;
                        flash_attach = <0>;
                        resolution = <gc0329_FULL_RESOLUTION>;
                        powerup_sequence = <gc0329_PWRSEQ>;
                        orientation = <0>;
                        i2c_add = <gc0329_I2C_ADDR>;
                        i2c_rata = <100000>;
                        i2c_chl = <0>;
                        cif_chl = <0>;
                        mclk_rate = <24>;
                };
                gc0329_b {
                        is_front = <0>;
                        rockchip,powerdown = <&gpio3 GPIO_B3
GPIO_ACTIVE_HIGH>;
                        pwdn_active = <gc0329_PWRDN_ACTIVE>;
                        #rockchip,power = <>;
                        pwr_active = <PWR_ACTIVE_HIGH>;
                        #rockchip,reset = <>;
                        #rst_active = <>;
                        #rockchip,flash = <>;
                        #rockchip,af = <>;
                        mir = <0>;
                        flash_attach = <0>;
                        resolution = <gc0329_FULL_RESOLUTION>;
                        powerup_sequence = <gc0329_PWRSEQ>;
                        orientation = <0>;
                        i2c_add = <gc0329_I2C_ADDR>;
                        i2c_rata = <100000>;
                        i2c_chl = <0>; // <0>;
                        cif_chl = <0>;
                        mclk_rate = <24>;
                };
        };
```

这里列出 gc0329 双摄像头模组的配置，其属性解释如下：

- is_front: 0: 后置摄像头，1: 前置摄像头
- rockchip,powerdown: 定义该摄像头对应的 powerdown gpio
- pwdn_active: powerdown 的有效电平

- mir: 0: 不支持镜像，1: 支持镜像，
- resolution: 最大的分辨率
- orientation: 旋转角度，0: 0 度， 90: 90 度， 180: 180 度， 270: 270 度
- i2c_add: 摄像头的 I²C 地址
- i2c_rata: I²C 频率，单位 Hz
- i2c_chl: I²C 通道号
- cif_chl: cif 控制器信息，rk312x 仅有 cif0
- mclk_rate: 传感器时钟频率，单位：MHz

其中的常量值都在 kernel/arch/arm/mach-rockchip/rk_camera_sensor_info.h 中定义：

```
#define gc0329_FULL_RESOLUTION     0x30000          // 0.3 megapixel
#define gc0329_I2C_ADDR           0x62
#define gc0329_PWRDN_ACTIVE           0x01
#define gc0329_PWRSEQ                sensor_PWRSEQ_DEFAULT
//Sensor power  active level define
#define PWR_ACTIVE_HIGH               0x01
#define PWR_ACTIVE_LOW                                    0x0
```

分辨率对应的图像长宽，在 kernel/drivers/media/video/generic_sensor.h 的 sensor_get_full_width_height 函数里定义：

```
static inline int sensor_get_full_width_height(int full_resolution, unsigned
short *w, unsigned short *h)
{
    switch (full_resolution)
    {
        case 0x30000:
        {
            *w = 640;
            *h = 480;
            break;
        }

        case 0x100000:
        {
            *w = 1024;
            *h = 768;
            break;
        }
        //...
    }
}
```

1.8.2.2 启用 CIF 传感器设备

要启用 CIF ，只需在 kernel/arch/arm/boot/dts/rk3128-fireprime.dts 加入：

```
&rk3128_cif_sensor{
        status = "okay";
};
```

1.9 SPI 使用

1.9.1 SPI 工作方式

SPI 以主从方式工作，这种模式通常有一个主设备和一个或多个从设备，需要至少 4 根线，分别是：

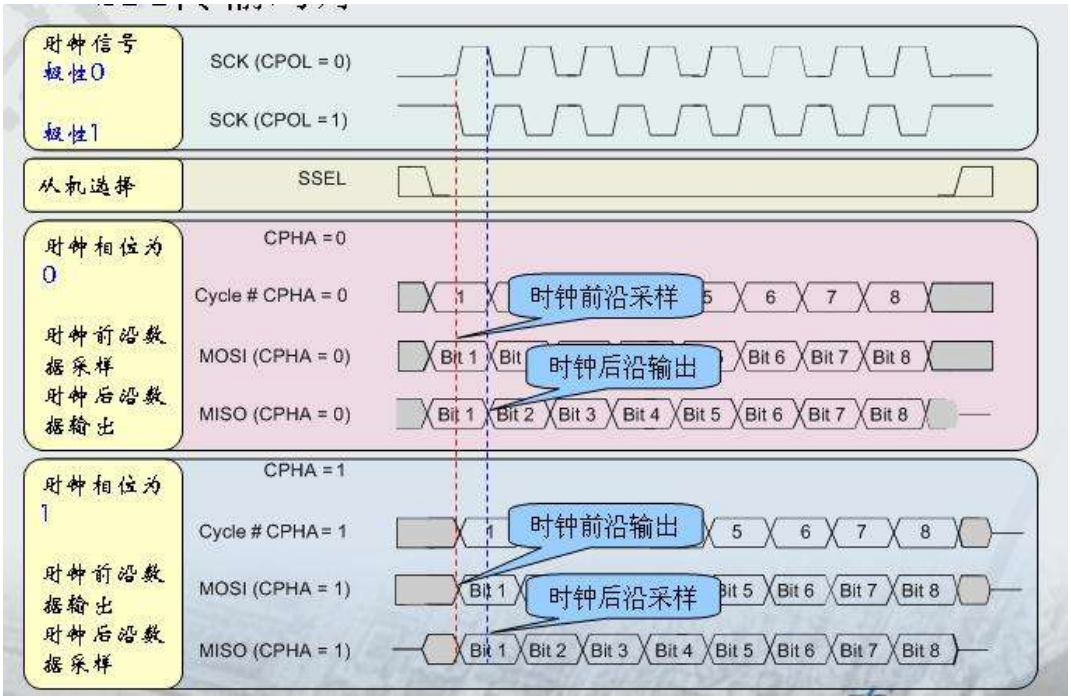
CS	片选信号
SCLK	时钟信号
MOSI	主设备数据输出、从设备数据输入
MISO	主设备数据输入，从设备数据输出

Linux 内核用 CPOL 和 CPHA 的组合来表示当前 SPI 的四种工作模式：

CPOL＝0，CPHA＝0	SPI_MODE_0
CPOL＝0，CPHA＝1	SPI_MODE_1
CPOL＝1，CPHA＝0	SPI_MODE_2
CPOL＝1，CPHA＝1	SPI_MODE_3

CPOL：表示时钟信号的初始电平的状态， 0 为低电平， 1 为高电平。
CPHA：表示在哪个时钟沿采样， 0 为第一个时钟沿采样， 1 为第二个时钟沿采样。

SPI 的四种工作模式波形图如下：



1.9.2 在内核添加自己的驱动文件

在内核源码目录 kernel/drivers/spi/中创建新的驱动文件，如：spi-rockchip-firefly.c 在驱动文件所在目录下的 Kconfig 文件添加对应的驱动文件配置，如：

```
@@ -525,6 +525,10 @@ config SPI_ROCKCHIP_TEST
      bool "ROCKCHIP spi test code"
      depends on SPI_ROCKCHIP

+config SPI_ROCKCHIP_FIREFLY
+      bool "ROCKCHIP spi firefly code"
+      depends on SPI_ROCKCHIP
+
 #
 # There are lots of SPI device types, with sensors and memory
 # being probably the most widely used ones.
```

在驱动文件所在目录下的 Makefile 文件添加对应的驱动文件名，如：

```
+obj-$(CONFIG_SPI_ROCKCHIP_FIREFLY) +=
spi-rockchip-firefly.o
```

用 make menuconfig 在内核选项中选中所添加的驱动文件，如：

```
There is no help available for this option.
 │ Symbol: SPI_ROCKCHIP_FIREFLY [=y]
 │ Type  : boolean
 │ Prompt: ROCKCHIP spi firefly code
 │   Location:
 │     -> Device Drivers
 │       -> SPI support (SPI [=y])
 │         -> ROCKCHIP SPI controller core support
(SPI_ROCKCHIP_CORE [=y])
 │           -> ROCKCHIP SPI interface driver (SPI_ROCKCHIP
[=y])
 │   Defined at drivers/spi/Kconfig:528
 │   Depends on: SPI [=y] && SPI_MASTER [=y] && SPI_ROCKCHIP
[=y]
```

1.9.3 定义和注册 SPI 设备

在 DTS 中添加 SPI 驱动结点描述，如下所示： kernel/arch/arm/boot/dts/rk3128-fireprime.dts

```
&spi0 {
        status = "okay";
        max-freq = <24000000>;
        spidev@00 {
               compatible = "rockchip,spi_firefly";
               reg = <0x00>;
               spi-max-frequency = <14000000>;
               spi-cpha = <1>;
               //spi-cpol = <1>;
        };
};
```

status:如果要启用 SPI，则设为 okay，如不启用，设为 disable。
spidev@00:由于本例子使用的是 SPI0，且使用 CS0，故此处设为 00，如果使用 CS1，则设为 01。
compatible:这里的属性必须与驱动中的结构体：of_device_id 中的成员 compatible 保持一致。
reg:此处与 spidev@00 保持一致，本例设为：0x00；
spi-max-frequency：此处设置 spi 使用的最高频率。
spi-cpha，spi-cpol：SPI 的工作模式在此设置，本例所用的模块 SPI 工作模式为 SPI_MODE_1，故设：spi-cpha = <1>，如果您所用设备工作模式为 SPI_MODE0，则需在此把这两个注释掉，如果用 SPI_MODE3，则设：spi-cpha = <1>;spi-cpol = <1>。

1.9.4 定义和注册 SPI 驱动

1.9.4.1 定义 SPI 驱动

在定义 SPI 驱动之前，用户首先要定义变量 of_device_id 。 of_device_id 用于在驱动中调用 dts 文件中定义的设备信息，其定义如下所示：

```
static const struct of_device_id spidev_dt_ids[] = {
       { .compatible = "rockchip,spi_firefly" },
       {},
};
```

此处的 compatible 与 DTS 文件中的保持一致。

定义 spi_driver 如下所示：

```
static struct spi_driver spidev_spi_driver = {
        .driver = {
             .name =        "silead_fp",
             .owner =       THIS_MODULE,
             .of_match_table =
of_match_ptr(spidev_dt_ids),
       },
       .probe =       spi_gsl_probe,
       .remove =      spi_gsl_remove,
};
```

1.9.4.2 注册 SPI 驱动

在初始化函数 static int __init spidev_init(void)中创建一个字符设备： alloc_chrdev_region(&devno, 0,255, "sileadfp");
向内核添加该设备：

```
spidev_major = MAJOR(devno);
cdev_init(&spicdev, &spidev_fops);
spicdev.owner = THIS_MODULE;
status = cdev_add(&spicdev,MKDEV(spidev_major,
0),N_SPI_MINORS);
```

创建设备类：

```
class_create(THIS_MODULE, "spidev");
```

向内核注册 SPI 驱动：

```
spi_register_driver(&spidev_spi_driver);
```

如果内核启动时匹配成功，则调用该驱动的 probe 函数。 probe 函数如下所示：

```
static int spi_gsl_probe(struct spi_device *spi)
{
	struct spidev_data *spidev;
	int                                status;
	unsigned long            minor;
	struct gsl_fp_data  *fp_data;

	printk("===============spi_gsl_probe
==============\n");
	if(!spi)
		return -ENOMEM;

	/* Allocate driver data */
	spidev = kzalloc(sizeof(*spidev), GFP_KERNEL);
	if (!spidev)
		return -ENOMEM;

	/* Initialize the driver data */
	spidev->spi = spi;
	spin_lock_init(&spidev->spi_lock);//初始化自旋锁

	mutex_init(&spidev->buf_lock);//初始化互斥锁

	INIT_LIST_HEAD(&spidev->device_entry);//初始化设备链
表

	//init fp_data
	fp_data = kzalloc(sizeof(struct gsl_fp_data),
GFP_KERNEL);
	if(fp_data == NULL){
		status = -ENOMEM;
		return status;
```

```
	}
	//set fp_data struct value
	fp_data->spidev = spidev;


	mutex_lock(&device_list_lock);//上互斥锁

	minor = find_first_zero_bit(minors, N_SPI_MINORS);//
在内存区中查找第一个值为 0 的位

	if (minor < N_SPI_MINORS) {
		struct device *dev;
		spidev->devt = MKDEV(spidev_major, minor);
		dev = device_create(spidev_class, &spi->dev,
spidev->devt, spidev, "silead_fp_dev");创建/dev/下设备结点

		status = IS_ERR(dev) ? PTR_ERR(dev) : 0;
	} else {
		dev_dbg(&spi->dev, "no minor number
available!\n");
		status = -ENODEV;
	}
	if (status == 0) {
		set_bit(minor, minors);
		list_add(&spidev->device_entry,
&device_list);//添加进设备链表

	}

	mutex_unlock(&device_list_lock);//解互斥锁

	if (status == 0)
		spi_set_drvdata(spi, spidev);
	else
		kfree(spidev);


	printk("%s:name=%s,bus_num=%d,cs=%d,mode=%d,speed=%
d\n",__func__,spi->modalias, spi->master->bus_num,
spi->chip_select, spi->mode,

spi->max_speed_hz);//打印 SPI 信息

	return status;
}
```

如果注册 SPI 驱动成功，你可以在/dev/目录下面看你到注册的驱动名称，可以在 sys/class/下面看到你注册的驱动设备类。

1.9.5 SPI 读写数据过程

1.9.5.1 SPI 写数据

```
static ssize_t spidev_write(struct file *filp, const char
__user *buf, size_t count, loff_t *f_pos)
{
	struct spidev_data	*spidev;
	ssize_t			status = 0;
	unsigned long		missing;
	if (count > bufsiz)
		return -EMSGSIZE;

	spidev = filp->private_data;

	mutex_lock(&spidev->buf_lock);
	missing = copy_from_user(spidev->buffer, buf,
count);//把数据从用户空间传到内核空间
	if (missing == 0) {
		status = spidev_sync_write(spidev, count);//
调用写同步函数
	} else
		status = -EFAULT;
	mutex_unlock(&spidev->buf_lock);

	return status;
}
```

写同步函数：

```
spidev_sync_write(struct spidev_data *spidev, size_t len)
{
	struct spi_transfer	t = {
			.tx_buf		= spidev->buffer,//发送
缓冲区
			.len		= len,//发送数据长度
		};
```

```
        struct spi_message      m;
        spi_message_init(&m);//初始化 spi_message
        spi_message_add_tail(&t, &m);//将新的 spi_transfer 添
加到 spi_message 队列尾部
        return spidev_sync(spidev, &m);//同步读写
}
```

1.9.5.2 SPI 读数据

在本例所用的模块中，读数据的过程为：

1. 主机向模块写寄存器的地址及读的指令（如：地址为 0xf0,读指令为 0x00）
2. 模块收到读的指令后，数据以页的形式发送
3. 主机设置读的模式
4. 主机读取一页数据并存储

```
static ssize_t
spidev_read(struct file *filp, char __user *buf, size_t
count, loff_t *f_pos)
{
        struct spidev_data *spidev;
        int                status = 0;
        int          i = 0;

        spidev = filp->private_data;
        mutex_lock(&spidev->buf_lock);


        gsl_fp_write(spidev, 0x00, 0xf0);//读之前先向模块写读的
指令及寄存器地址
        while(1){
                for(i=0;i <= 110*118/128/read_pages;i++){
                        status =
gsl_fp_getOneFrame(spidev,0x00);//读 1 页数据
                }
                pos = 0;
                break;
        }
```

```
	if(status > 0){
		printk("gsl read data success!!!\n");
	}else{
		printk("gsl read data failed!!!");
	}

	mutex_unlock(&spidev->buf_lock);
	return status;
}
static inline unsigned int
gsl_fp_getOneFrame(struct spidev_data *spidev,unsigned char
reg_8b)
{
	int status,i;
	unsigned char buf_d[128*1+3] = {0x00,0x00};
	struct spi_transfer t;
	t.tx_buf = buf_d;
	t.rx_buf = buf_d;
	t.len = 131;

	status = gsl_spidev_sync_read(spidev, &t);

	if (status > 0){

		for(i=0;i<128*read_pages;i++)
			kmalloc_area[pos++] = buf_d[i+3];
	}

	return status;
}
static inline ssize_t
gsl_spidev_sync_read(struct spidev_data *spidev,struct
spi_transfer *t)
{
	struct spi_message m;
	spi_message_init(&m);


	t->bits_per_word = 8;//每次读的数据长度为 8 位

	t->delay_usecs = 1;//每次读完延时

	t->speed_hz = 14*1000*1000;//读的速率
```

```
        t->cs_change = 1;//CS 引脚电平变化

        spi_message_add_tail(t, &m);
        return spidev_sync(spidev, &m);
}
```

注：Firefly 的 SPI 驱动是 Linux 下通用的驱动，可以参考源码：kernel/drivers/spi/spidev.c

1.10 MIPI DSI 使用

1.10.1 Config 配置

在 arch/arm/configs/firefly-rk3128_defconfig 添加配置：

```
CONFIG_LCD_MIPI=y
CONFIG_MIPI_DSI=y
CONFIG_RK31_MIPI_DSI=y
```

1.10.2 LCD 引脚配置

引脚配置放在 arch/arm/boot/dts/firefly-rk3128.dts 中的 lcdc0 子节点 power_ctr 中 ,分别有电源使能引脚 lcd_en、片选引脚 lcd_cs，复位引脚 lcd_rst，可以根据显示屏做修改和删减。 如：

```
673     power_ctr: power_ctr {
674         rockchip,debug = <0>;
675         lcd_en:lcd_en {
676             rockchip,power type = <GPIO>;
677             gpios = <&gpio7 GPIO_A3 GPIO_ACTIVE_HIGH>;
678             rockchip,delay = <10>;
679         };
680
681         lcd cs:lcd cs {
682             rockchip,power_type = <GPIO>;
683             gpios = <&gpio7 GPIO A4 GPIO ACTIVE HIGH>;
684             rockchip,delay = <10>;
685         };
686
687         /*lcd_rst:lcd_rst {
```

```
688           rockchip,power_type = <GPIO>;
689           gpios = <&gpio3 GPIO D6 GPIO ACTIVE HIGH>;
690           rockchip,delay = <5>;
691       };*/
```

1.10.3 驱动配置

1.10.3.1 新建 DTS 配置文件

在 arch/arm/boot/dts/目录中新建 dst 配置文件，如 lcd-xxx-mipi.dtsi。

1.10.3.2 添加 DTS 文件

在 arch/arm/boot/dts/firefly-rk3128.dts 中添加#include "lcd-xxx-mipi.dtsi"，如果原来 include 了其他屏的 DTS 配置，注释掉它们。

1.10.3.3 添加背光节点信息

在 lcd-xxx-mipi.dtsi 中添加背光节点信息。

```
  backlight {
      compatible = "pwm-backlight";
      pwms = <&pwm1 0 10000>;
      rockchip,pwm_id= <1>;
      /* | dark(255-221) | light scale(220-0) | , scale_div=255*/
      brightness-levels = </*255 254 253 252 251 250 249 248 247 246 245 244 243 242 241
240 239 238 237 236 235 234 233 232 231 230 229 228 227 226 225
224 223 222 221 */220 219 218 217 216 215 214 213 212 211 210 209 208 207 206 205 204 203
202 201 200 199 198 197 196 195 194 193 192 191 190 189 188 187
186 185 184 183 182 181 180 179 178 177 176 175 174 173 172 171 170 169 168 167 166 165
164 163 162 161 160 159 158 157 156 155 154 153 152 151 150 149
148 147 146 145 144 143 142 141 140 139 138 137 136 135 134 133 132 131 130 129 128 127
126 125 124 123 122 121 120 119 118 117 116 115 114 113 112 111
110 109 108 107 106 105 104 103 102 101 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85
84 83 82 81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34
33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13
12 11 10 9 8 7 6 5 4 3 2 1 0>;
      default-brightness-level = <128>;
      enable-gpios = <&gpio8 GPIO_A6 GPIO_ACTIVE_HIGH>;
  };
```

pwms 属性：配置 PWM，Firefly-RK3128 使用 pwm1，范例中的 10000 是 PWM 频率。
brightness-levels 属性：配置背光亮度数组，最大值为 255，配置暗区和亮区，并把亮区数组做 255 的比例调节。比如范例中暗区是 255-221，亮区是 220-0。
default-brightness-level 属性：开机时默认背光亮度，范围为 0-255。

enable-gpios 属性：配置背光使能引脚。

具体请参考 kernel 中的说明文档：

Documentation/devicetree/bindings/video/backlight/pwm-backlight.txt

1.10.3.4 配置 MIPI 相关信息

```
disp mipi init: mipi dsi init{
          compatible = "rockchip,mipi_dsi_init";
          rockchip,screen_init    = <1>;
          rockchip,dsi lane      = <4>;
          rockchip,dsi_hs_clk     = <1000>;
          rockchip,mipi_dsi_num   = <1>;
};
```

rockchip,screen_init 属性：0 表示不需要特殊指令初始化显示屏，1,表示需要初始化指令。

rockchip,dsi_lane 属性：数据 lane 的数量。

rockchip,dsi_hs_clk 属性：配置 hsclk。

rockchip,mipi_dsi_num：配置只用 DSI 接口的数量，即单通道 MIPI 屏为 1，双通道 MIPI 屏为 2。

具体请参考 kernel 中的说明文档：

Documentation/devicetree/bindings/video/rockchip_mipidsi_lcd.txt

1.10.3.5 配置初始化命令

当 rockchip,screen_init 为 1 时需要配置显示屏的初始化命令，初始化命令在节点 disp_mipi_init_cmds 中配置。

rockchip,cmd_debug 属性：打开可输出指令调试信息。

rockchip,on-cmdsXX 子节点：配置每条指令的信息。

rockchip,cmd_type：数据传输模式，LPDT 或 HSDT。

rockchip,dsi_id：指令传输的 DSI 接口，0 为向 DSI0（双通道 MIPI 屏时为左半屏）发送指令，1 为向 DSI1（双通道 MIPI 屏时为右半屏）发送指令,2 为同时向两个 DSI 发送数据。

rockchip,cmd：指令序列。其中第一个字节为 DSI 数据类型，第二个字节为 REG，后面的字节为指令内容。

rockchip,cmd_delay：发送指令后的延时，单位为 ms。

1.10.3.6 配置显示时序

时序的在节点 disp_timings 配置。

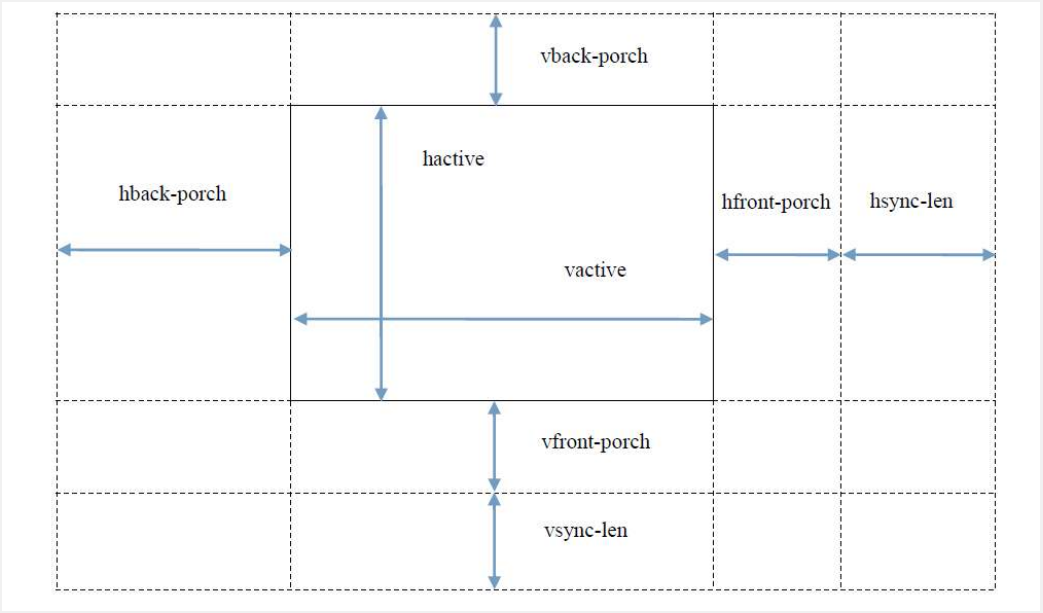
screen-type 属性：显示屏类型，单通道 MIPI 屏时为 SCREEN_MIPI，双通道 MIPI 屏时为 SCREEN_DUAL_MIPI。

lvds-format 属性：无关选项。

out-face 属性：配置颜色，可为 OUT_P888（24 位）、OUT_P666（18 位）或者 OUT_P565（16 位）。

clock-frequency 属性：屏时钟，单位 Hz。

其他的时序属性参考下图：



1.10.3.7 dsihost 配置

如果是双 MIPI 屏，需要使能 dsihost0 和 dsihost1，如：

```
&dsihost0 {
   status = "okay";
};
&dsihost1 {
   status = "okay";
};
```

如果是单 MIPI 屏，只需使能 dsihost0，如：

```
&dsihost0 {
   status = "okay";
};
```

1.11 待续……