

BH1750驱动(I2C设备)程序编写

BH1750驱动(I2C设备)程序编写

1.确定使用的IO口并准备硬件

2.修改设备树

2.1.设备树修改实操:

3.裁剪LINUX内核, 编译并烧写

3.1.裁剪内核

3.2.编译内核

3.3.烧写SD卡

4.BH1750驱动编写

4.1.BH1750基础知识

4.1.1.I2C

4.1.2.BH1750驱动流程

4.2.bh1750_drv.c逐帧解析

4.3.完整的bh1750.c

4.3.BH1750应用程序编写

5.现象测试

实现步骤:

1.确定使用的IO口并准备硬件

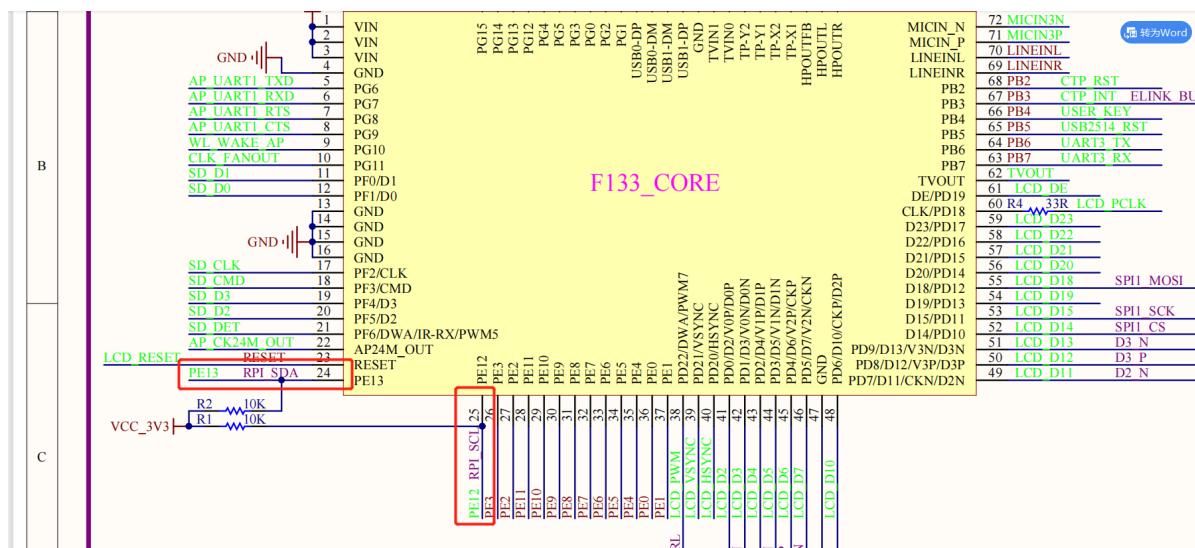
2.修改设备树

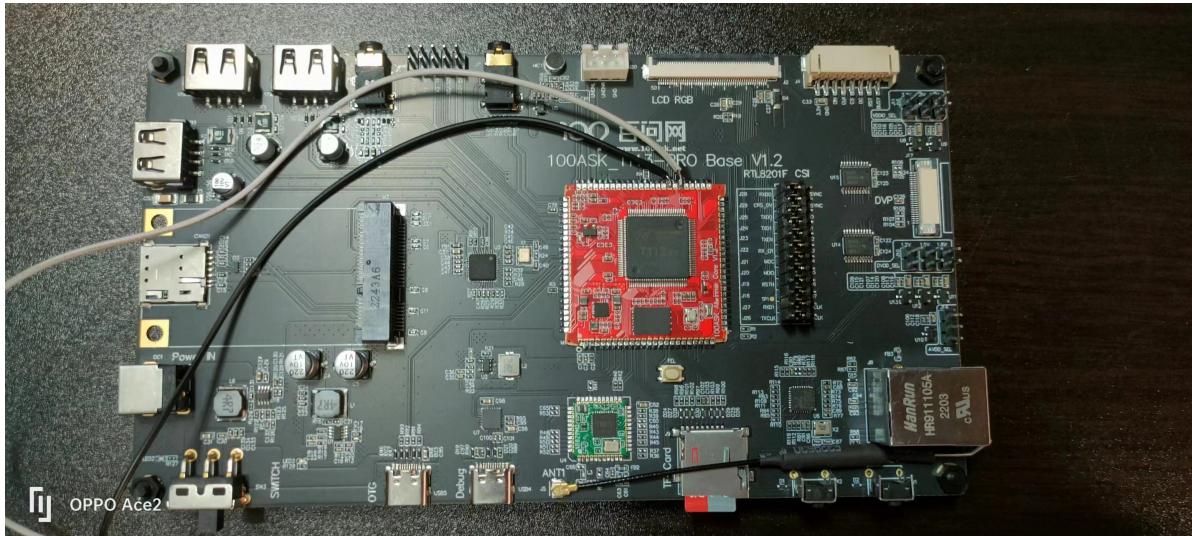
3.裁剪LINUX内核, 编译并烧写

4.BH1750驱动编写

1.确定使用的IO口并准备硬件

I2C接口的话, 开发板上已经准备好了引脚引出, 直接用就可以





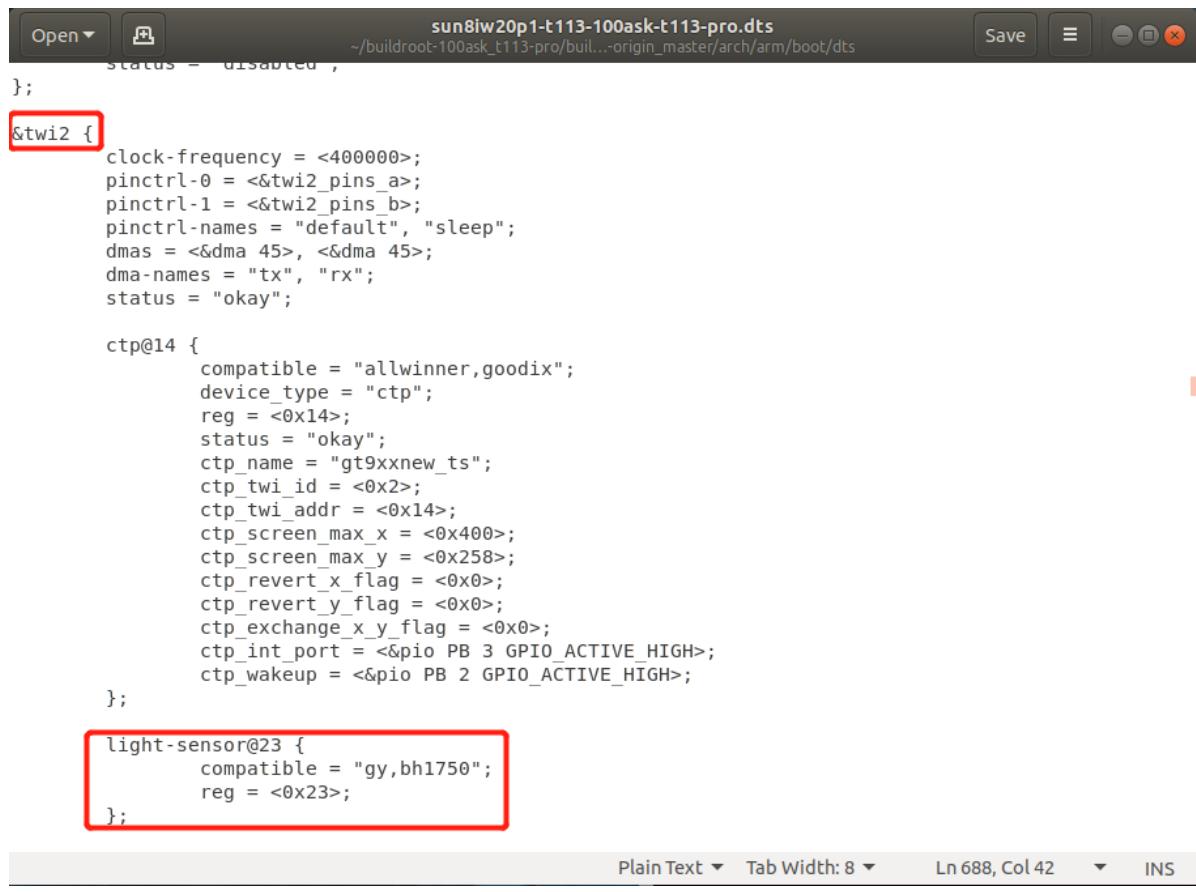
2.修改设备树

2.1.设备树修改实操:

```
cd /buildroot-100ask_t113-pro/buildroot/output/build/linux-  
origin_master/arch/arm/boot/dts  
  
sudo gedit sun8iw20p1-t113-100ask-t113-pro.dts
```

代码块1：挂载light-sensor@23设备节点到twi2设备节点上

```
light-sensor@23 {  
    compatible = "gy,bh1750";  
    reg = <0x23>;  
};
```



```

sun8iw20p1-t113-100ask-t113-pro.dts
-/buildroot-100ask_t113-pro/build..origin_master/arch/arm/boot/dts

status = "disabled";
};

&twi2 {
    clock-frequency = <400000>;
    pinctrl-0 = <&twi2_pins_a>;
    pinctrl-1 = <&twi2_pins_b>;
    pinctrl-names = "default", "sleep";
    dmas = <&dma 45>, <&dma 45>;
    dma-names = "tx", "rx";
    status = "okay";

    ctp@14 {
        compatible = "allwinner,goodix";
        device_type = "ctp";
        reg = <0x14>;
        status = "okay";
        ctp_name = "gt9xxnew_ts";
        ctp_twi_id = <0x2>;
        ctp_twi_addr = <0x14>;
        ctp_screen_max_x = <0x400>;
        ctp_screen_max_y = <0x258>;
        ctp_revert_x_flag = <0x0>;
        ctp_revert_y_flag = <0x0>;
        ctp_exchange_x_y_flag = <0x0>;
        ctp_int_port = <&pio PB 3 GPIO_ACTIVE_HIGH>;
        ctp_wakeup = <&pio PB 2 GPIO_ACTIVE_HIGH>;
    };

    light-sensor@23 {
        compatible = "gy,bh1750";
        reg = <0x23>;
    };
};

Plain Text ▾ Tab Width: 8 ▾ Ln 688, Col 42 ▾ INS

```

3.裁剪LINUX内核，编译并烧写

3.1.裁剪内核

1. I2C总线支持: Device Drivers --> I2C support --> <*> I2C Hardware Bus support
2. I2C设备驱动程序支持: Device Drivers --> I2C support --> <*> I2C device interface
3. SYSFS支持: File systems --> Pseudo filesystems --> <*> sysfs file system support
4. 工作队列支持: Kernel Hacking --> Kernel Debugging --> [*] Debug Workqueue usage
5. 电源管理支持: Power management and ACPI options --> [/] ACPI (Advanced Configuration and Power Interface) Support --> <> ACPI Support
6. 模块化支持: Loadable module support --> <*> Enable loadable module support

一般默认都添加上了，如果不放心可以检查一下

3.2.编译内核

进入到/buildroot目录下，输入以下两条命令即可在/output/images目录下找到生成的sd卡镜像，该命令用到了buildroot框架。

```

make linux-rebuild v=1
make v=1

```

3.3.烧写SD卡

不演示

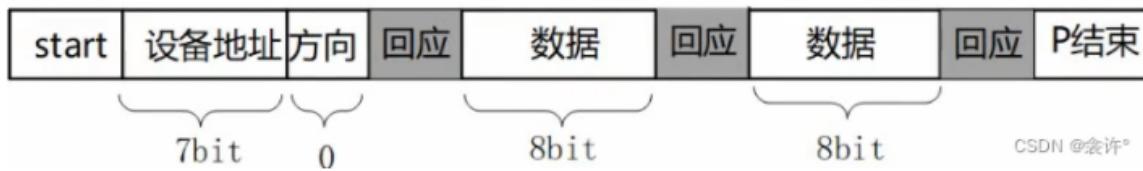
4.BH1750驱动编写

作为I2C设备，可以说学会了I2C驱动的编写，你才能说你大致学会了驱动开发的代码怎么写

4.1.BH1750基础知识

4.1.1.I2C

该不会还要我再说一遍I2C的基础知识吧？..



数据写入write

流程如下：

1. 主芯片(开发板内置)发出一个START信号到从设备
2. 主芯片发出设备地址 (用于选中I2C总线上的I2C设备) 以及方向 (0表示写, 1表示读)
3. 从设备回应 (用于确定设备是否存在)
4. 主设备发送一个字节的数据给从设备，并等待回应，当接受到从设备的回应后才继续发送下一个字节的数据
5. 数据发送结束后，主设备送出一个停止信号给从设备。

数据读出read

流程如下：

1. 主芯片(开发板内置)发出一个START信号到从设备
2. 主芯片发出设备地址 (用于选中I2C总线上的I2C设备) 以及方向 (0表示写, 1表示读)
3. 从设备回应 (用于确定设备是否存在)
4. 从设备发送一个字节的数据给主设备，并等待回应，当接受到主设备的回应后才继续发送下一个字节的数据
5. 数据发送结束后，主设备送出一个停止信号给从设备。

说白了I2C就是读和写，而读操作和写操作的区别也就是，第二步的设备地址0/1的区别，第四步谁发给谁的区别。

4.1.2.BH1750驱动流程

BH1750的手册写的有点迷，直接上操作流程不行吗？

第1步：发送上电命令即(0X01)

BH1750FVI Technical Note

指令集合结构

指令	功能代码	注释
断电	0000_0000	无激活状态。
通电	0000_0001	等待测量指令。
重置	0000_0111	重置数字寄存器值， 重置指令在断电模式下不起作用。
连续 H 分辨率模式	0001_0000	在 1lx 分辨率下开始测量。 测量时间一般为 120ms。
连续 H 分辨率模式 2	0001_0001	在 0.5lx 分辨率下开始测量。 测量时间一般为 120ms。
连续 L 分辨率模式	0001_0011	在 41lx 分辨率下开始测量。 测量时间一般为 16ms。
一次 H 分辨率模式	0010_0000	在 1lx 分辨率下开始测量。 测量时间一般为 120ms。 测量后自动设置为断电模式。
		在 0.5lx 分辨率下开始测量。

第2步：发送测量命令，用于配置BH1750的测量模式，它推荐我用H分辨率模式，我这里用H分辨率模式2即(0x11)

BH1750FVI Technical Note

测量模式说明

测量模式	测量时间	分辨率
H-分辨率模式 2	典型时间： 120ms.	0.5lx
H-分辨率模式	典型时间： 120ms.	1lx.
L-分辨率模式	典型时间： 16ms.	4lx.

我们建议您使用 H 分辨率模式。
H 分辨率模式下足够长的测量时间（积分时间）能够抑制一些噪声（包括 50Hz/60Hz）。
同时，H 分辨率模式的分辨率在 1lx 下，适用于黑暗场合下（少于 10 lx）。H 分辨率模式 2 同样适用于黑暗场合下的检测。
解释异步复位和重置的命令 “0000_0111”

1. 异步重置
电源供应时序基础上将所有寄存器复位序列。请参考本页的 **VCC 和 DVI 电源供应时序** 图供电序列。在 **DVI = “0”** 是电源掉电模式。
2. 重置命令
重置命令仅对光强度数据寄存器起作用（指令值为 “0”）电源供应时序对其无影响。它的作用是原来清除之前的测量结果。这个命令不能在断电模式，所以在输入该指令前要设置为通电模式。
VCC 和 DVI 电源供应时序图

BH1750FVI Technical Note

指令集合结构

指令	功能代码	注释
断电	0000_0000	无激活状态。
通电	0000_0001	等待测量指令。
重置	0000_0111	重置数字寄存器值， 重置指令在断电模式下不起作用。
连续 H 分辨率模式	0001_0000	在 1lx 分辨率下开始测量。 测量时间一般为 120ms。
连续 H 分辨率模式 2	0001_0001	在 0.5lx 分辨率下开始测量。 测量时间一般为 120ms。
连续 L 分辨率模式	0001_0011	在 41lx 分辨率下开始测量。 测量时间一般为 16ms。
一次 H 分辨率模式	0010_0000	在 1lx 分辨率下开始测量。 测量时间一般为 120ms。 测量后自动设置为断电模式。
一次 H 分辨率模式 2	0010_0001	在 0.5lx 分辨率下开始测量。 测量时间一般为 120ms。

第3步：等待测量结束。

测量的时间手册上面有写，我这里就不列出来了，高分辨率连续测量需要等待的时间最长，手册上面写的是平均120ms，最大值180ms，所以为了保证每次读取到的数据都是最新测量的，程序上面可以延时200ms以上，当然也不用太长，浪费时间。如果你用别的测量模式，等待时间都比这个模式要短。

第4步：读取数据。

先是“起始信号(ST)”，接着是“器件地址+读写位”，然后是应答位，紧接着接收1个字节的数据（单片机在这个时候要把SDA引脚从输出改成输入了），然后给BH1750发送应答，继续接收1个字节数据，然后不应答（因为我们接收的数据只有2个字节，收完就可以结束通讯了），最后是“结束信号(SP)”。

第5步：计算结果。

接收完两个字节还不算完成，因为这个数据还不是测量出来的光照强度值，我们还需要进行计算，计算公式是：光照强度 = (寄存器值[15:0] * 分辨率) / 1.2 (单位：勒克斯lx)

因为我们从BH1750寄存器读出来的是2个字节的数据，先接收的是高8位[15:8]，后接收的是低8位[7:0]，所以我们需要先把这2个字节合成一个数，然后乘上分辨率，再除以1.2即可得到光照值。

例如：我们读出来的第1个字节是0x12(0001 0010)，第2个字节是0x53(0101 0011)，那么合并之后就是0x1253(0001 0010 0101 0011)，换算成十进制也就是4691，乘上分辨率（我用的分辨率是1），再除以1.2，最后等于3909.17 lx。

明白了就直接上编程实践：

还是这七步：

```
/* 1. 确定主设备号*/  
/* 2. 定义file_operations结构体*/  
/* 3. 实现对应的open/read/write等函数，填入file_operations结构体 */  
/* 4. 把file_operations结构体告诉内核：注册驱动程序*/  
/* 5. 谁来注册驱动程序啊？得有一个入口函数：安装驱动程序时就会去调用这个入口函数*/  
/* 6. 有入口函数就应该有出口函数：卸载驱动程序时，就会去调这个出口函数*/  
/* 7. 其他完善：提供设备信息，自动创建设备节点*/
```

4.2.bh1750_drv.c逐帧解析

前情提要：i2c_client、i2c_msg、结构体，详见：https://blog.csdn.net/qq_33001335/article/details/129476743

```
struct i2c_client {  
    unsigned short flags;          /* div., see below      */  
#define I2C_CLIENT_PEC      0x04    /* Use Packet Error Checking */  
#define I2C_CLIENT_TEN      0x10    /* we have a ten bit chip address */  
                                /* Must equal I2C_M_TEN below */  
#define I2C_CLIENT_SLAVE     0x20    /* we are the slave */  
#define I2C_CLIENT_HOST_NOTIFY 0x40    /* We want to use I2C host notify */  
#define I2C_CLIENT_WAKE      0x80    /* for board_info; true iff can wake */  
#define I2C_CLIENT_SCCB      0x9000  /* Use Omniprecision SCCB protocol */  
                                /* Must match I2C_M_STOP|IGNORE_NAK */  
  
    unsigned short addr;          /* chip address - NOTE: 7bit      */  
                                /* addresses are stored in the */  
                                /* _LOWER_ 7 bits      */  
    char name[I2C_NAME_SIZE];  
    struct i2c_adapter *adapter;  /* the adapter we sit on      */  
    struct device dev;           /* the device structure      */
```

```

int init_irq;           /* irq set at initialization */
int irq;                /* irq issued by device */
struct list_head detected;

#if IS_ENABLED(CONFIG_I2C_SLAVE)
    i2c_slave_cb_t slave_cb; /* callback for slave mode */
#endif
};

struct i2c_msg {/**//这个结构体一看就知道作用了吧？设备地址，数据长度，数据缓冲区。
    __u16 addr; /* slave address */
    __u16 flags;
#define I2C_M_RD      0x0001 /* read data, from slave to master */
                           /* I2C_M_RD is guaranteed to be 0x0001! */
#define I2C_M_TEN     0x0010 /* this is a ten bit chip address */
#define I2C_M_DMA_SAFE 0x0200 /* the buffer of this message is DMA safe */
                           /* makes only sense in kernelspace */
                           /* userspace buffers are copied anyway */
#define I2C_M_RECV_LEN 0x0400 /* length will be first received byte */
#define I2C_M_NO_RD_ACK 0x0800 /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_IGNORE_NAK 0x1000 /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_REV_DIR_ADDR 0x2000 /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_NOSTART   0x4000 /* if I2C_FUNC_NOSTART */
#define I2C_M_STOP      0x8000 /* if I2C_FUNC_PROTOCOL_MANGLING */
    __u16 len;      /* msg length */
    __u8 *buf;      /* pointer to msg data */
};

/*unsigned short addr;//这个addr成员用于记录I2C设备地址
struct i2c_adapter *adapter;//这个成员点进去发现里面装了一些如nr等总线信息(设备挂在哪条I2C
总线上)

//剩下的就比较常规，和一般的设备驱动结构体差不多，只不过Linux内核的代码帮我们封装好了这些变量成
员
*/

```

/* 1. 确定主设备号*/

```

struct class *class; /* 类 */
int major;           /* 主设备号 */
struct i2c_client *gy_sensor_client; /*这个i2c_client结构体里面定义了设备地址，设备挂载
在哪条总线上等信息。*/

```

/* 2. 定义file_operations结构体*/

```

/* bh1750 操作函数 */
static const struct file_operations gy_sensor_fops = {
    .owner = THIS_MODULE,
    .open = gy_sensor_open,
    .read = gy_sensor_read,
};

```

/* 3. 实现对应的open/read/write等函数，填入file_operations结构体 */

gy_sensor_write_reg(unsigned char addr) (I2C写操作函数)

```
// 构造i2c_msg通过这个client调用i2c_transfer来读写
static int gy_sensor_write_reg(unsigned char addr)
{
    int ret = -1;
    struct i2c_msg msgs;

    printk("gy_sensor_client -> addr=%d\n", gy_sensor_client->addr);
    msgs.addr = gy_sensor_client->addr; // GY302_ADDR, 直接封装于i2c_msg
    msgs.buf = &addr;
    msgs.len = 1; //长度1 byte
    msgs.flags = 0; //表示写

    ret = i2c_transfer(gy_sensor_client->adapter, &msgs, 1);
    //这里都封装好了, 本来根据I2C协议写数据需要先写入器件写地址, 然后才能读
    if (ret < 0)
    {
        printk("i2c_transfer write err\n");
        return -1;
    }
    return 0;
}

/*这段代码主要是完成了*/
```

gy_sensor_read_reg(unsigned char *buf) (I2C读操作)

```
static int gy_sensor_read_reg(unsigned char *buf)
{
    int ret = -1;
    struct i2c_msg msg;
    msg.addr = gy_sensor_client->addr; // GY30_ADDR
    msg.buf = buf;
    msg.len = 2; //长度1 byte
    msg.flags = I2C_M_RD; //表示读
    ret = i2c_transfer(gy_sensor_client->adapter, &msg, 1); //这里都封装好了, 本来根据I2C协议读数据需要先写入读地址, 然后才能读
    if (ret < 0)
    {
        printk("i2c_transfer write err\n");
        return -1;
    }
    return 0;
}
```

open函数

```
// 初始化光线传感器
int gy_sensor_open(struct inode *inode, struct file *file)
{
    printk("open gy_sensor\n");
    gy_sensor_write_reg(0x01); //通电
    gy_sensor_write_reg(0x11); //H测量模式2
    return 0;
}
```

read函数

```
// 读出传感器的两个字节
static ssize_t gy_sensor_read(struct file *file, char __user *buf, size_t count,
loff_t *off)
{
    int ret;
    //unsigned char addr = 0;
    unsigned char data[2];
    gy_sensor_read_reg(data);
    ret = copy_to_user(buf, data, 2);
    return 1;
}
```

创建platform_driver，对于platform_driver，我感觉其实也就是出口函数和入口函数的进一步封装，也是一种框架，这里直接归在fileoperation结构体的函数定义填充，我感觉也是因为两个框架其实很像

```
/* 构造一个platform_driver,
其中的of_match_table字段需要与 light-sensor@23 节点的compatible属性值一致,
当匹配时则调用platform_driver的probe函数 */
static const struct of_device_id ids[] =
{
    { .compatible = "gy,bh1750" },
    {}
};
```

gy_sensor_probe(struct i2c_client *client, const struct i2c_device_id *id)

```
// 在i2c_driver的probe函数中得到在总线驱动程序中解析得到的i2c_client,
// 并为该光线传感器注册一个字符设备
static int gy_sensor_probe(struct i2c_client *client,
                           const struct i2c_device_id *id)
{
    gy_sensor_client = client;

    printk("%s %s %d\n", __FILE__, __FUNCTION__, __LINE__);
    major = register_chrdev(0, "gy_sensor", &gy_sensor_fops); //注册字符设备
    class = class_create(THIS_MODULE, "gy_sensor"); //创建类
    device_create(class, NULL, MKDEV(major, 0), NULL, "gy_sensor"); /* /dev/gy_sensor */ //创建设备
    return 0;
}
```

```

gy_sensor_remove(struct i2c_client *client)

// 在platform_driver的remove函数中，注销该字符设备
static int gy_sensor_remove(struct i2c_client *client)
{
    device_destroy(class, MKDEV(major, 0));
    class_destroy(class);
    unregister_chrdev(major, "gy_sensor");

    return 0;
}

```

platform_driver结构体

```

/* 分配/设置i2c_driver */
static struct i2c_driver gy_sensor_driver = {
    .driver = {
        .name = "bh1750",
        .owner = THIS_MODULE,
        .of_match_table = ids,
    },
    .probe = gy_sensor_probe,
    .remove = gy_sensor_remove,
};

```

/* 4. 把file_operations结构体告诉内核：注册驱动程序*/

/* 5. 谁来注册驱动程序啊？得有一个入口函数：安装驱动程序时就会去调用这个入口函数*/

```

static int __init bh1750_init(void)
{
    int ret = 0;]

    ret = i2c_add_driver(&gy_sensor_driver);
    return ret;
}

```

/* 6. 有入口函数就应该有出口函数：卸载驱动程序时，就会去调这个出口函数*/

```

static void __exit bh1750_exit(void)
{
    i2c_del_driver(&gy_sensor_driver);
}

```

/* 7. 其他完善：提供设备信息，自动创建设备节点*/

```

/* module_i2c_driver(bh1750_driver) */
module_init(bh1750_init);
module_exit(bh1750_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("zhu");

```

4.3.完整的bh1750.c

```
#include <linux/ide.h>
#include <linux/module.h>
#include <linux/i2c.h>

struct class *class; /* 类 */
int major; /* 主设备号 */
struct i2c_client *gy_sensor_client;

// 构造i2c_msg通过这个client调用i2c_transfer来读写
static int gy_sensor_write_reg(unsigned char addr)
{
    int ret = -1;
    struct i2c_msg msgs;

    printk("gy_sensor_client -> addr=%d\n", gy_sensor_client->addr);
    msgs.addr = gy_sensor_client->addr; // GY302_ADDR, 直接封装于i2c_msg
    msgs.buf = &addr;
    msgs.len = 1; //长度1 byte
    msgs.flags = 0; //表示写

    ret = i2c_transfer(gy_sensor_client->adapter, &msgs, 1); //i2c_transfer帮你完成了START信号STOP信号等操作, 只需要你对几个结构体填充一下变量成员传进去就能实现功能
    if (ret < 0)
    {
        printk("i2c_transfer write err\n");
        return -1;
    }
    return 0;
}

static int gy_sensor_read_reg(unsigned char *buf)
{
    int ret = -1;
    struct i2c_msg msg;
    msg.addr = gy_sensor_client->addr; // GY30_ADDR
    msg.buf = buf;
    msg.len = 2; //长度1 byte
    msg.flags = I2C_M_RD; //表示读
    ret = i2c_transfer(gy_sensor_client->adapter, &msg, 1); //这里都封装好了, 本来根据i2c协议读数据需要先写入读地址, 然后才能读
    if (ret < 0)
    {
        printk("i2c_transfer write err\n");
        return -1;
    }
    return 0;
}

// 初始化光线传感器
int gy_sensor_open(struct inode *inode, struct file *file)
{
    printk("open gy_sensor\n");
```

```

gy_sensor_write_reg(0x01); // power up
gy_sensor_write_reg(0x11);
return 0;
}

// 读出传感器的两个字节
static ssize_t gy_sensor_read(struct file *file, char __user *buf, size_t count,
loff_t *off)
{
    int ret;
    //unsigned char addr = 0;
    unsigned char data[2];
    gy_sensor_read_reg(data);
    ret = copy_to_user(buf, data, 2);
    return 1;
}

/* bh1750 操作函数 */
static const struct file_operations gy_sensor_fops = {
    .owner = THIS_MODULE,
    .open = gy_sensor_open,
    .read = gy_sensor_read,
};

/* 构造一个platform_driver,
其中的of_match_table字段需要与 light-sensor@23 节点的compatible属性值一致,
当匹配时则调用platform_driver的probe函数 */
static const struct of_device_id ids[] =
{
    {.compatible = "gy,bh1750"},
    {}
};

// 在i2c_driver的probe函数中得到在总线驱动程序中解析得到的i2c_client,
// 并为该光线传感器注册一个字符设备
static int gy_sensor_probe(struct i2c_client *client,
                           const struct i2c_device_id *id)
{
    gy_sensor_client = client;

    printk("%s %s %d\n", __FILE__, __FUNCTION__, __LINE__);
    major = register_chrdev(0, "gy_sensor", &gy_sensor_fops);
    class = class_create(THIS_MODULE, "gy_sensor");
    device_create(class, NULL, MKDEV(major, 0), NULL, "gy_sensor"); /* /dev/gy_sensor */
    return 0;
}

// 在platform_driver的remove函数中, 注销该字符设备
static int gy_sensor_remove(struct i2c_client *client)
{
    device_destroy(class, MKDEV(major, 0));
    class_destroy(class);
    unregister_chrdev(major, "gy_sensor");
}

```

```

    return 0;

}

/* 分配/设置i2c_driver */
static struct i2c_driver gy_sensor_driver = {
    .driver = {
        .name = "bh1750",
        .owner = THIS_MODULE,
        .of_match_table = ids,
    },
    .probe = gy_sensor_probe,
    .remove = gy_sensor_remove,
};

/*
 * @description : 驱动入口函数
 */
static int __init bh1750_init(void)
{
    int ret = 0;

    ret = i2c_add_driver(&gy_sensor_driver);
    return ret;
}

/*
 * @description : 驱动出口函数
 */
static void __exit bh1750_exit(void)
{
    i2c_del_driver(&gy_sensor_driver);
}

/* module_i2c_driver(bh1750_driver) */
module_init(bh1750_init);
module_exit(bh1750_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("zhu");

```

4.3.BH1750应用程序编写

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

```

```
int main(int argc, char **argv)
{
    int fd;
    char val;
    unsigned char buf[3];
    float flight;
    fd = open("/dev/gy_sensor", O_RDWR);
    if (fd < 0)
    {
        printf("can't open /dev/gy_sensor\n");
        return -1;
    }

    usleep(200000);
    while(1)
    {
        if(read(fd,&buf,3)){
            flight=(buf[0]*256+buf[1])*0.5/1.2;
            printf("light: %6.2f\n",flight);
        }
        else{
            printf("read err!\n");
        }
        sleep(4);
    }

    return 0;
}
```

5.现象测试
