

//这些都是系统自带的头文件。

#include <linux/clk.h>

#include <linux/device.h>

#include <linux/delay.h>

#include <linux/gpio/consumer.h>

#include <linux/i2c.h>

#include <linux/module.h>

#include <linux/pm\_runtime.h>

#include <linux/of.h>

#include <linux/of\_graph.h>

#include <linux/regulator/consumer.h>

#include <linux/sysfs.h>

#include <linux/slab.h>

#include <linux/pinctrl/consumer.h>

#include <linux/version.h>

#include <linux/rk-camera-module.h>

//下面的都是关于media（v4l2）的一些头文件，红色警告不用管

#include <media/v4l2-async.h>

#include <media/media-entity.h>

#include <media/v4l2-common.h>

#include <media/v4l2-ctrls.h>

#include <media/v4l2-device.h>

#include <media/v4l2-event.h>

#include <media/v4l2-fwnode.h>

#include <media/v4l2-image-sizes.h>

#include <media/v4l2-mediabus.h>

#include <media/v4l2-subdev.h>

#define IMX415\_NAME "imx415"

#define to\_imx415(sd) container\_of(sd,struct imx415,subdev)

#define IMX415\_FLIP\_MIRROR\_REG 0x17

#define GC\_MIRROR\_BIT\_MASK BIT(0) //定义GC2053镜像位掩码

#define GC\_FLIP\_BIT\_MASK BIT(1) //定义GC2053翻转位

#define IMX415\_REG\_CTRL\_MODE 0x3E //定义控制模式寄存器地址

#define IMX415\_MODE\_SW\_STANDBY 0x11 //定义软件待机模式

#define IMX415\_MODE\_STREAMING 0x91 //定义流式传输模式

#define IMX415\_LANES 2 //定义通道数

#define REG\_NULL 0xFF

//对imx415\_write\_array形参：寄存器结构体进行定义：

struct regval {

u8 addr; //寄存器地址

u8 val; //寄存器写的值

};

//定义imx415\_g\_frame\_interval所需的imx415\_mode

struct imx415\_mode {

u32 width; //摄像头图像宽度

u32 height; //摄像头图像高度

struct v4l2\_fract max\_fps; //摄像头最大帧率

u32 hts\_def; //摄像头默认行总时间

u32 vts\_def; //摄像头默认场总时间

u32 exp\_def; //摄像头默认曝光时间

const struct regval \*reg\_list; //指向摄像头寄存器列表的指针

u32 hdr\_mode; //摄像头 HDR 模式

u32 vc[PAD\_MAX]; //摄像头虚拟通道

};

static int imx415\_s\_power(struct v4l2\_subdev \*sd, int on)

{

struct imx415 \*imx415 = to\_imx415(sd);

struct i2c\_client \*client = imx415->client;

int ret = 0;

mutex\_lock(&imx415->mutex);

/\* If the power state is not modified - no work to do. \*/

if (imx415->power\_on == !!on)

goto unlock\_and\_return;

if (on) {

//让sensor处于正常的电源工作状态

ret = pm\_runtime\_get\_sync(&client->dev);//同步获取设备的电源状态，确保设备被唤醒并处于活动状态

if (ret < 0) {

pm\_runtime\_put\_noidle(&client->dev);

goto unlock\_and\_return;

}

imx415->power\_on = true;

} else {

//低功耗状态

pm\_runtime\_put(&client->dev);

imx415->power\_on = false;

}

unlock\_and\_return:

mutex\_unlock(&imx415->mutex);

return ret;

}

/\* sensor register write \*/

static int imx415\_write\_reg(struct i2c\_client \*client, u8 reg, u8 val)

{

struct i2c\_msg msg;

u8 buf[2];

int ret;

buf[0] = reg & 0xFF;

buf[1] = val;

//传进来的寄存器地址和对应的值val

//怎么发出去呢，也就是说，我的子设备怎么知道呢？就需要I2C总线进行传输。

//对msg结构体进行填充

msg.addr = client->addr;

msg.flags = client->flags;

msg.buf = buf; //要传输的数据

msg.len = sizeof(buf);

//借助i2c\_transfer函数，开始进行i2c总线传输数据。

ret = i2c\_transfer(client->adapter, &msg, 1);

if (ret >= 0)

return 0;

dev\_err(&client->dev,

"gc2053 write reg(0x%x val:0x%x) failed !\n", reg, val);

return ret;

}

static long imx415\_ioctl(struct v4l2\_subdev \*sd, unsigned int cmd, void \*arg)

{

struct imx415 \*imx415 = to\_imx415(sd);

long ret = 0;

struct rkmodule\_hdr\_cfg \*hdr\_cfg;

u32 stream = 0;

switch (cmd) {

case RKMODULE\_GET\_HDR\_CFG:

hdr\_cfg = (struct rkmodule\_hdr\_cfg \*)arg;

hdr\_cfg->esp.mode = HDR\_NORMAL\_VC;

hdr\_cfg->hdr\_mode = imx415->cur\_mode->hdr\_mode;

break;

case RKMODULE\_SET\_HDR\_CFG:

case RKMODULE\_SET\_CONVERSION\_GAIN:

break;

/\*

ret = -EINVAL;

dev\_err(&imx415->client->dev, "imx415 not support hdr mode\n");

break;

\*/

case RKMODULE\_GET\_MODULE\_INFO:

imx415\_get\_module\_inf(imx415, (struct rkmodule\_inf \*)arg);

break;

case RKMODULE\_AWB\_CFG:

imx415\_set\_awb\_cfg(imx415,(struct rkmodule\_awb\_cfg \*)arg);

break;

case RKMODULE\_LSC\_CFG:

imx415\_set\_lsc\_cfg(imx415,(struct rkmodule\_lsc\_cfg \*)arg);

break;

case RKMODULE\_SET\_QUICK\_STREAM:

stream = \*((u32 \*)arg);

if (stream)

ret = imx415\_write\_reg(imx415->client, IMX415\_REG\_CTRL\_MODE,

IMX415\_MODE\_STREAMING);

else

ret = imx415\_write\_reg(imx415->client, IMX415\_REG\_CTRL\_MODE,

IMX415\_MODE\_SW\_STANDBY);

break;

default:

ret = -ENOTTY;

break;

}

return ret;

}

//对\_\_imx415\_start\_stream里面的imx415\_write\_array进行实现

static int imx415\_write\_array(struct i2c\_client \*client,const struct regval \*regs)

{

int i, ret = 0;

i = 0;

while (regs[i].addr != REG\_NULL) { //这一句是对结构体数组的每一个addr成员进行遍历，查看是否为REG\_NULL。

//也就是说当地址不是0xFF的时候就往寄存器写数据。

ret = imx415\_write\_reg(client, regs[i].addr, regs[i].val);

if (ret) {

dev\_err(&client->dev, "%s failed !\n", \_\_func\_\_);

break;

}

i++;

}

return ret;

}

//对imx415\_set\_flip里面的gc2053\_read\_reg进行实现

static int imx415\_read\_reg(struct i2c\_client \*client, u8 reg, u8 \*val)

{

struct i2c\_msg msg[2];

u8 buf[1];

int ret;

buf[0] = reg & 0xFF;

msg[0].addr = client->addr;

msg[0].flags = client->flags;

msg[0].buf = buf;

msg[0].len = sizeof(buf);

msg[1].addr = client->addr;

msg[1].flags = client->flags | I2C\_M\_RD;

msg[1].buf = buf;

msg[1].len = 1;

ret = i2c\_transfer(client->adapter, msg, 2);

if (ret >= 0) {

\*val = buf[0];

return 0;

}

dev\_err(&client->dev,

"imx415 read reg(0x%x val:0x%x) failed !\n", reg, \*val);

return ret;

}

//实现\_\_imx415\_start\_stream里面的imx415\_set\_flip

//设置镜像

static int imx415\_set\_flip(struct imx415 \*imx415, u8 mode)

{

u8 match\_reg = 0;

imx415\_read\_reg(imx415->client, IMX415\_FLIP\_MIRROR\_REG, &match\_reg);

if (mode == GC\_FLIP\_BIT\_MASK) {

match\_reg |= GC\_FLIP\_BIT\_MASK;

match\_reg &= ~GC\_MIRROR\_BIT\_MASK;

} else if (mode == GC\_MIRROR\_BIT\_MASK) {

match\_reg |= GC\_MIRROR\_BIT\_MASK;

match\_reg &= ~GC\_FLIP\_BIT\_MASK;

} else if (mode == (GC\_MIRROR\_BIT\_MASK |

GC\_FLIP\_BIT\_MASK)) {

match\_reg |= GC\_FLIP\_BIT\_MASK;

match\_reg |= GC\_MIRROR\_BIT\_MASK;

} else {

match\_reg &= ~GC\_FLIP\_BIT\_MASK;

match\_reg &= ~GC\_MIRROR\_BIT\_MASK;

}

return imx415\_write\_reg(imx415->client, IMX415\_FLIP\_MIRROR\_REG, match\_reg);

}

//实现imx415\_s\_stream的start和stop\_stream:

//开启和关闭流需要对寄存器进行操作，所以需要写寄存器，同时会有v4l2控制器的处理以及镜像翻转，所以我们在开启流的时候可以做一些操作操作，而这些操作就需要对寄存器进行写操作。

static int \_\_imx415\_start\_stream(struct imx415 \*imx415)

{

int ret;

//首先对imx415 sensor 寄存器 填写值

ret = imx415\_write\_array(imx415->client, imx415->cur\_mode->reg\_list);

if (ret)

return ret;

/\* In case these controls are set before streaming \*/

mutex\_unlock(&imx415->mutex);

//然后对v4l2控制器做设置操作，

v4l2\_ctrl\_handler\_setup(&imx415->ctrl\_handler);

mutex\_lock(&imx415->mutex);

//例如作镜像翻转等

ret = imx415\_set\_flip(imx415, imx415->flip);

if (ret)

return ret;

//最后开启流

return imx415\_write\_reg(imx415->client, IMX415\_REG\_CTRL\_MODE,IMX415\_MODE\_STREAMING);

}

static int \_\_imx415\_stop\_stream(struct imx415 \*imx415)

{

return imx415\_write\_reg(imx415->client, IMX415\_REG\_CTRL\_MODE,

IMX415\_MODE\_SW\_STANDBY);

}

//实现v4l2\_subdev\_video\_ops结构体的imx415\_s\_stream

static int imx415\_s\_stream(struct v4l2\_subdev \*sd, int on)

{

struct imx415 \*imx415 = to\_imx415(sd);

struct i2c\_client \*client = imx415->client;

int ret = 0;

mutex\_lock(&imx415->mutex); //上锁是因为不能并行操作

on = !!on;

if (on == imx415->streaming)

goto unlock\_and\_return;

if (on) { //如果on为1就开启流

ret = pm\_runtime\_get\_sync(&client->dev);

if (ret < 0) {

pm\_runtime\_put\_noidle(&client->dev);

goto unlock\_and\_return;

}

ret = \_\_imx415\_start\_stream(imx415);

if (ret) {

v4l2\_err(sd, "start stream failed while write regs\n");

pm\_runtime\_put(&client->dev);

goto unlock\_and\_return;

}

} else {

\_\_imx415\_stop\_stream(imx415);

pm\_runtime\_put(&client->dev);

}

imx415->streaming = on;

unlock\_and\_return:

mutex\_unlock(&imx415->mutex);

return 0;

}

//填充imx415\_core\_ops：

static const struct v4l2\_subdev\_core\_ops imx415\_core\_ops = {

.s\_power = imx415\_s\_power, //对sensor电源进行操作

.ioctl = imx415\_ioctl,

};

//实现v4l2\_subdev\_video\_ops所需的imx415\_g\_frame\_interval帧间隔：

static int imx415\_g\_frame\_interval(struct v4l2\_subdev \*sd,struct v4l2\_subdev\_frame\_interval \*fi)

{

struct imx415 \*imx415 = to\_imx415(sd);

const struct imx415\_mode \*mode = imx415->cur\_mode;

mutex\_lock(&imx415->mutex);

fi->interval = mode->max\_fps;

mutex\_unlock(&imx415->mutex);

return 0;

}

//实现v4l2\_subdev\_video\_ops所需的imx415\_g\_mbus\_config

//v4l2\_mbus\_config定义在 include/media/v4l2\_mediabus.h文件里。

static int imx415\_g\_mbus\_config(struct v4l2\_subdev \*sd,struct v4l2\_mbus\_config \*config)

{

#if USE\_DVP\_INTERFACE

config->type = V4L2\_MBUS\_PARALLEL;

config->flags = V4L2\_MBUS\_HSYNC\_ACTIVE\_HIGH |

V4L2\_MBUS\_VSYNC\_ACTIVE\_LOW |

V4L2\_MBUS\_PCLK\_SAMPLE\_RISING;

#else

struct imx415 \*imx415 = to\_imx415(sd);

const struct imx415\_mode \*mode = imx415->cur\_mode;

u32 val = 0;

if (mode->hdr\_mode == NO\_HDR)

val = 1 << (IMX415\_LANES - 1) |

//下面这两个定义在：include/media/v4l2\_mediabus.h文件里面

V4L2\_MBUS\_CSI2\_CHANNEL\_0 |

V4L2\_MBUS\_CSI2\_CONTINUOUS\_CLOCK;

config->type = V4L2\_MBUS\_CSI2;

config->flags = val;

#endif

return 0;

}

//填充imx415\_video\_ops

//v4l2\_subdev\_video\_ops结构体定义在：include/media/v4l2\_subdev.h文件里面

static const struct v4l2\_subdev\_video\_ops imx415\_video\_ops = {

.s\_stream = imx415\_s\_stream,

.g\_frame\_interval = imx415\_g\_frame\_interval,

.g\_mbus\_config = imx415\_g\_mbus\_config,

};

//填充imx415\_pad\_ops

static const struct v4l2\_subdev\_pad\_ops imx415\_pad\_ops = {

.enum\_mbus\_code = imx415\_enum\_mbus\_code,

.enum\_frame\_size = imx415\_enum\_frame\_sizes,

.enum\_frame\_interval = imx415\_enum\_frame\_interval,

.get\_format = imx415\_get\_format, //获取摄像头格式

.set\_format = imx415\_set\_format, //设置摄像头格式

};

static const struct v4l2\_subdev\_ops imx415\_subdev\_ops =

{

.core = &imx415\_core\_ops,

.video = &imx415\_video\_ops,

.pad = &imx415\_pad\_ops,

};

//实现imx415\_pm\_ops：

static const struct dev\_pm\_ops imx415\_pm\_ops = {

SET\_RUNTIME\_PM\_OPS(imx415\_runtime\_suspend,

imx415\_runtime\_resume, NULL)

};

static int imx415\_pm\_resume(struct device \*dev)

//3.实现prob函数

static int imx415\_probe(struct i2c\_client \*client,const struct i2c\_device\_id \*id)

{

//1.定义相关结构体

struct device \*dev = &client->dev;

struct device\_node \*node = dev->of\_node;

struct imx415 \*imx415;

struct v4l2\_subdev \*sd;

int ret;

//2.分配内存为imx415结构体，获取其设备树里面的信息

imx415 = devm\_kzalloc(dev,sizeof(\*imx415),GFP\_KERNEL);

imx415->client = client;

//3.开始解析设备信息。

//解析模块索引值

ret = of\_property\_read\_u32(node,RKMODULE\_CAMERA\_MODULE\_INDEX,&imx415->module\_index);

//解析摄像头是前向还是后置。

ret |= of\_property\_read\_string(node, RKMODULE\_CAMERA\_MODULE\_FACING,

&imx415->module\_facing);

//解析模块名称

ret |= of\_property\_read\_string(node, RKMODULE\_CAMERA\_MODULE\_NAME,

&imx415->module\_name);

//解析镜头名称

ret |= of\_property\_read\_string(node, RKMODULE\_CAMERA\_LENS\_NAME,

&imx415->len\_name);

if (ret) {

dev\_err(dev,

"could not get module information!\n");

return -EINVAL;

}

imx415->xvclk = devm\_clk\_get(&client->dev, "xvclk");

if (IS\_ERR(imx415->xvclk)) {

dev\_err(&client->dev, "Failed to get xvclk\n");

return -EINVAL;

}

imx415->reset\_gpio = devm\_gpiod\_get(dev, "reset", GPIOD\_OUT\_LOW);

if (IS\_ERR(imx415->reset\_gpio))

dev\_warn(dev, "Failed to get reset-gpios\n");

imx415->pwdn\_gpio = devm\_gpiod\_get(dev, "pwdn", GPIOD\_OUT\_LOW);

if (IS\_ERR(imx415->pwdn\_gpio))

dev\_info(dev, "Failed to get pwdn-gpios, maybe no used\n");

imx415->power\_gpio = devm\_gpiod\_get(dev, "power", GPIOD\_OUT\_LOW);

if (IS\_ERR(imx415->power\_gpio))

dev\_warn(dev, "Failed to get power-gpios\n");

ret = imx415\_configure\_regulators(imx415);

if (ret) {

dev\_err(dev, "Failed to get power regulators\n");

return ret;

}

ret = imx415\_parse\_of\_mipi(imx415);

if (ret != 0)

return -EINVAL;

imx415->pinctrl = devm\_pinctrl\_get(dev);

if (!IS\_ERR(imx415->pinctrl)) {

imx415->pins\_default =

pinctrl\_lookup\_state(imx415->pinctrl,

OF\_CAMERA\_PINCTRL\_STATE\_DEFAULT);

if (IS\_ERR(imx415->pins\_default))

dev\_err(dev, "could not get default pinstate\n");

imx415->pins\_sleep =

pinctrl\_lookup\_state(imx415->pinctrl,

OF\_CAMERA\_PINCTRL\_STATE\_SLEEP);

if (IS\_ERR(imx415->pins\_sleep))

dev\_err(dev, "could not get sleep pinstate\n");

} else {

dev\_err(dev, "no pinctrl\n");

}

mutex\_init(&imx415->mutex);

sd = &imx415->subdev;

//4. 初始化v4l2\_subdev结构体

v4l2\_i2c\_subdev\_init(sd, client, &imx415\_subdev\_ops);

ret = imx415\_initialize\_controls(imx415);

if (ret)

goto err\_destroy\_mutex;

ret = \_\_imx415\_power\_on(imx415);

if (ret)

goto err\_free\_handler;

ret = imx415\_check\_sensor\_id(imx415, client);

if (ret)

goto err\_power\_off;

return 0;

}

//4.

static int imx415\_remove(struct i2c\_client \*client)

{

return 0;

}

//5. 定义匹配表

static const struct i2c\_device\_id imx415\_i2c\_driver[] =

{

{"imx415",0},

{ },

};

#if IS\_ENABLE(CONFIG\_OF)

static const struct of\_device\_id imx415\_of\_match[] = {

{.compatible = "fsl,imx415", },

{ },

};

MODULE\_DEVICE\_TABLE(of, imx415\_of\_match);

#endif

//1.定义和设置i2c\_drv取得结构体

/\*

driver：Device driver model driver主要包含驱动名称和与DTS注册设备进行匹配的

of\_match\_table。当of\_match\_table中的compatible域 和 dts文件的compatible域匹配时，·probe函数才会被调用

id\_table:List of I2C devices supported by this driver如果kernel没有使用of\_match\_table和

dts注册设备进行进行匹配，则kernel使用该table进行匹配

probe:Callback for device binding

remove:Callback for device unbinding

\*/

static struct i2c\_driver imx415\_i2c\_driver = {

.driver = {

.name = IMX415\_NAME,

.pm = &imx415\_pm\_ops,

.of\_match\_table = of\_match\_ptr(imx415\_of\_match),

},

.probe = &imx415\_probe,

.remove = &imx415\_remove,

.id\_table = imx415\_match\_id,

};

//2.注册imx415\_i2c\_driver结构体

static int \_\_init sensor\_mod\_init(void)

{

return i2c\_add\_driver(&imx415\_i2c\_driver);

}

static void \_\_exit sensor\_mod\_exit(void)

{

i2c\_del\_driver(&imx415\_i2c\_driver);

}

device\_initcall\_sync(sensor\_mod\_init);

module\_exit(sensor\_mod\_exit);

MODULE\_DESCRIPTION("IMX415 CMOS Image Sensor driver");

MODULE\_LICENSE("GPL v2");

//这就是一个sensor的驱动框架。

总结：

今天主要就是完成了v4l2\_subdev\_video\_ops里面的这三个回调函数。

一个是开启流，获得摄像头镜像

一个是设置帧间隔，设置一帧的时间

最后是配置MIPI相关的总线的类型。

后面再去实现端口号的操作：v4l2\_subdev\_pad\_ops: