```
/*
 * imx219.c - imx219 sensor driver
 *
 * Copyright (c) 2014-2015, NVIDIA CORPORATION, All Rights Reserved.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * This program is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <linux/delay.h>
#include <linux/fs.h>
#include <linux/i2c.h>
#include <linux/clk.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/regulator/consumer.h>
#include <media/imx219.h>
#include <linux/gpio.h>
#include <linux/module.h>
#include <linux/sysedp.h>
#include <linux/kernel.h>
#include <linux/debugfs.h>
#include <linux/seq_file.h>

#include "nvc_utilities.h"

#include "imx219_tables.h"

struct imx219_info {
    struct miscdevice       miscdev_info;
    int             mode;
    struct imx219_power_rail    power;
    struct nvc_fuseid       fuse_id;
    struct i2c_client       *i2c_client;
    struct imx219_platform_data *pdata;
    struct clk          *mclk;
    struct mutex            imx219_camera_lock;
    struct dentry           *debugdir;
    atomic_t            in_use;
#ifdef CONFIG_DEBUG_FS
    struct dentry           *debugfs_root;
    u32             debug_i2c_offset;
#endif
    struct sysedp_consumer *sysedpc;
    /* AF data */
    u8 afdat[4];
    bool afdat_read;
    struct imx219_gain pre_gain;
    bool    pre_gain_delay;
};

static inline void
msleep_range(unsigned int delay_base)
{
    usleep_range(delay_base*1000, delay_base*1000+500);
}

static inline void
imx219_get_frame_length_regs(struct imx219_reg *regs, u32 frame_length)
{
```

```
    regs->addr = 0x0160;
    regs->val = (frame_length >> 8) & 0xff;
    (regs + 1)->addr = 0x0161;
    (regs + 1)->val = (frame_length) & 0xff;
}

static inline void
imx219_get_coarse_time_regs(struct imx219_reg *regs, u32 coarse_time)
{
    regs->addr = 0x15a;
    regs->val = (coarse_time >> 8) & 0xff;
    (regs + 1)->addr = 0x15b;
    (regs + 1)->val = (coarse_time) & 0xff;
}

static inline void
imx219_get_gain_reg(struct imx219_reg *regs, struct imx219_gain gain)
{
    regs->addr = 0x157;
    regs->val = gain.again;
    (regs+1)->addr = 0x158;
    (regs+1)->val = gain.dgain_upper;
    (regs+2)->addr = 0x159;
    (regs+2)->val = gain.dgain_lower;
}

static int
imx219_read_reg(struct i2c_client *client, u16 addr, u8 *val)
{
    int err;
    struct i2c_msg msg[2];
    unsigned char data[3];

    if (!client->adapter)
        return -ENODEV;

    msg[0].addr = client->addr;
    msg[0].flags = 0;
    msg[0].len = 2;
    msg[0].buf = data;

    /* high byte goes out first */
    data[0] = (u8) (addr >> 8);
    data[1] = (u8) (addr & 0xff);

    msg[1].addr = client->addr;
    msg[1].flags = I2C_M_RD;
    msg[1].len = 1;
    msg[1].buf = data + 2;

    err = i2c_transfer(client->adapter, msg, 2);
    if (err == 2) {
        *val = data[2];
        return 0;
    }

    pr_err("%s:i2c read failed, addr %x, err %d\n",
           __func__, addr, err);

    return err;
}

static int
imx219_write_reg(struct i2c_client *client, u16 addr, u8 val)
{
    int err;
    struct i2c_msg msg;
    unsigned char data[3];

    if (!client->adapter)
```

```
        return -ENODEV;

    data[0] = (u8) (addr >> 8);
    data[1] = (u8) (addr & 0xff);
    data[2] = (u8) (val & 0xff);

    msg.addr = client->addr;
    msg.flags = 0;
    msg.len = 3;
    msg.buf = data;

    err = i2c_transfer(client->adapter, &msg, 1);
    if (err == 1)
        return 0;

    pr_err("%s:i2c write failed, addr %x, val %x, err %d\n",
            __func__, addr, val, err);

    return err;
}

static int
imx219_write_table(struct i2c_client *client,
                const struct imx219_reg table[],
                const struct imx219_reg override_list[],
                int num_override_regs)
{
    int err;
    const struct imx219_reg *next;
    int i;
    u16 val;

    for (next = table; next->addr != IMX219_TABLE_END; next++) {
        if (next->addr == IMX219_TABLE_WAIT_MS) {
            msleep_range(next->val);
            continue;
        }

        val = next->val;

        /* When an override list is passed in, replace the reg */
        /* value to write if the reg is in the list            */
        if (override_list) {
            for (i = 0; i < num_override_regs; i++) {
                if (next->addr == override_list[i].addr) {
                    val = override_list[i].val;
                    break;
                }
            }
        }

        err = imx219_write_reg(client, next->addr, val);
        if (err) {
            pr_err("%s:imx219_write_table:%d", __func__, err);
            return err;
        }
    }
    return 0;
}

static int
imx219_set_mode(struct imx219_info *info, struct imx219_mode *mode)
{
    int sensor_mode;
    int err;
    struct imx219_reg reg_list[8];

    pr_info("%s:xres %u yres %u framelength %u coarsetime %u again %u dgain %u%u\n"
             , __func__, mode->xres, mode->yres, mode->frame_length,
             mode->coarse_time, mode->gain.again,
```

```
            mode->gain.dgain_upper, mode->gain.dgain_lower);

    if (mode->xres == 3280 && mode->yres == 2460) {
        sensor_mode = IMX219_MODE_3280x2460;
    } else if (mode->xres == 1640 && mode->yres == 1232) {
        sensor_mode = IMX219_MODE_1640x1232;
    } else if (mode->xres == 3280 && mode->yres == 1846) {
        sensor_mode = IMX219_MODE_3280x1846;
    } else if (mode->xres == 1280 && mode->yres == 720) {
        sensor_mode = IMX219_MODE_1280x720;
    } else {
        pr_err("%s: invalid resolution supplied to set mode %d %d\n",
             __func__, mode->xres, mode->yres);
        return -EINVAL;
    }

    /* get a list of override regs for the asking frame length, */
    /* coarse integration time, and gain.                       */
    imx219_get_frame_length_regs(reg_list, mode->frame_length);
    imx219_get_coarse_time_regs(reg_list + 2, mode->coarse_time);
    imx219_get_gain_reg(reg_list + 4, mode->gain);

    err = imx219_write_table(info->i2c_client,
                mode_table[sensor_mode],
                reg_list, 7);

    info->pre_gain = mode->gain;
    info->pre_gain_delay = false;

    info->mode = sensor_mode;
    pr_info("[IMX219]: stream on.\n");
    return 0;
}

static int
imx219_get_status(struct imx219_info *info, u8 *dev_status)
{
    *dev_status = 0;
    return 0;
}

static int
imx219_set_frame_length(struct imx219_info *info, u32 frame_length)
{
    struct imx219_reg reg_list[2];
    int i = 0;
    int ret;

    imx219_get_frame_length_regs(reg_list, frame_length);

    for (i = 0; i < 2; i++) {
        ret = imx219_write_reg(info->i2c_client, reg_list[i].addr,
             reg_list[i].val);
        if (ret)
            return ret;
    }

    return 0;
}

static int
imx219_set_coarse_time(struct imx219_info *info, u32 coarse_time)
{
    int ret;

    struct imx219_reg reg_list[2];
    int i = 0;

    imx219_get_coarse_time_regs(reg_list, coarse_time);
```

```
    for (i = 0; i < 2; i++) {
        ret = imx219_write_reg(info->i2c_client, reg_list[i].addr,
             reg_list[i].val);
        if (ret)
            return ret;
    }

    return 0;
}

static int
imx219_set_gain(struct imx219_info *info, struct imx219_gain gain)
{
    int i;
    int ret;
    struct imx219_reg reg_list[3];

    imx219_get_gain_reg(reg_list, gain);
    for (i = 0; i < 3; ++i) {
        ret = imx219_write_reg(info->i2c_client,
                    reg_list[i].addr,
                    reg_list[i].val);
        if (ret) {
            pr_err("%s: unable to write register: %d",
                        __func__, ret);
            return ret;
        }
    }

    return ret;
}

static int
imx219_set_group_hold(struct imx219_info *info, struct imx219_ae *ae)
{
    int ret = 0;
    struct imx219_reg reg_list[8];
    int offset = 0;

    if (ae->frame_length_enable) {
        imx219_get_frame_length_regs(reg_list + offset,
                        ae->frame_length);
        offset += 2;
    }
    if (ae->coarse_time_enable) {
        imx219_get_coarse_time_regs(reg_list + offset, ae->coarse_time);
        offset += 2;
    }
    if (ae->gain_enable) {
        imx219_get_gain_reg(reg_list + offset, ae->gain);
        offset += 3;
    }

    reg_list[offset].addr = IMX219_TABLE_END;

    ret = imx219_write_table(info->i2c_client,
                reg_list, NULL, 0);
    return ret;
}

static int imx219_get_sensor_id(struct imx219_info *info)
{
    int ret = 0;
    int i;
    u8 bak = 0;

    pr_info("%s\n", __func__);
    if (info->fuse_id.size)
        return 0;
```

```
    /* Note 1: If the sensor does not have power at this point
    Need to supply the power, e.g. by calling power on function */

    for (i = 0; i < 4; i++) {
        ret |= imx219_read_reg(info->i2c_client, 0x0004 + i, &bak);
        pr_info("chip unique id 0x%x = 0x%02x\n", i, bak);
        info->fuse_id.data[i] = bak;
    }

    for (i = 0; i < 2; i++) {
        ret |= imx219_read_reg(info->i2c_client, 0x000d + i , &bak);
        pr_info("chip unique id 0x%x = 0x%02x\n", i + 4, bak);
        info->fuse_id.data[i + 4] = bak;
    }

    if (!ret)
        info->fuse_id.size = 6;

    /* Note 2: Need to clean up any action carried out in Note 1 */

    return ret;
}

static int imx219_get_af_data(struct imx219_info *info)
{
    int ret = 0;
    int i;
    u8 bak = 0;
    u8 *dat = (u8 *)info->afdat;

    pr_info("%s\n", __func__);
    if (info->afdat_read)
        return 0;

    imx219_write_reg(info->i2c_client, 0x0100, 0); /* SW-Stanby */
    msleep_range(33); /* wait one frame */

    imx219_write_reg(info->i2c_client, 0x012A, 0x18); /* 24Mhz input */
    imx219_write_reg(info->i2c_client, 0x012B, 0x00);

    imx219_write_reg(info->i2c_client, 0x3302, 0x02); /* clock setting */
    imx219_write_reg(info->i2c_client, 0x3303, 0x58); /* clock setting */
    imx219_write_reg(info->i2c_client, 0x3300, 0); /* ECC ON */
    imx219_write_reg(info->i2c_client, 0x3200, 1); /* set 'Read' */

    imx219_write_reg(info->i2c_client, 0x3202, 1); /* page 1 */

    for (i = 0; i < 4; i++) {
        ret |= imx219_read_reg(info->i2c_client, 0x3204 + i, &bak);
        *(dat+3-i) = bak;
        pr_info("[%d] x%x ", i, bak);
    }
    info->afdat_read = true;

    return ret;
}

static void imx219_mclk_disable(struct imx219_info *info)
{
    dev_dbg(&info->i2c_client->dev, "%s: disable MCLK\n", __func__);
    clk_disable_unprepare(info->mclk);
}

static int imx219_mclk_enable(struct imx219_info *info)
{
    int err;
    unsigned long mclk_init_rate = 24000000;

    dev_dbg(&info->i2c_client->dev, "%s: enable MCLK with %lu Hz\n",
        __func__, mclk_init_rate);
```

```
    err = clk_set_rate(info->mclk, mclk_init_rate);
    if (!err)
        err = clk_prepare_enable(info->mclk);
    return err;
}

static long
imx219_ioctl(struct file *file,
             unsigned int cmd, unsigned long arg)
{
    int err = 0;
    struct imx219_info *info = file->private_data;

    switch (cmd) {
    case IMX219_IOCTL_SET_POWER:
        if (!info->pdata)
            break;
        if (arg && info->pdata->power_on) {
            err = imx219_mclk_enable(info);
            if (!err) {
                sysedp_set_state(info->sysedpc, 1);
                err = info->pdata->power_on(&info->power);
            }
        if (err < 0)
            imx219_mclk_disable(info);
        }
        if (!arg && info->pdata->power_off) {
            info->pdata->power_off(&info->power);
            imx219_mclk_disable(info);
            sysedp_set_state(info->sysedpc, 0);
        }
        break;
    case IMX219_IOCTL_SET_MODE:
    {
        struct imx219_mode mode;
        if (copy_from_user(&mode, (const void __user *)arg,
            sizeof(struct imx219_mode))) {
            pr_err("%s:Failed to get mode from user.\n", __func__);
            return -EFAULT;
        }
        return imx219_set_mode(info, &mode);
    }
    case IMX219_IOCTL_SET_FRAME_LENGTH:
        err = imx219_set_frame_length(info, (u32)arg);
        break;
    case IMX219_IOCTL_SET_COARSE_TIME:
        err = imx219_set_coarse_time(info, (u32)arg);
        break;
    case IMX219_IOCTL_SET_GAIN:
    {
        struct imx219_gain gain;
        if (copy_from_user(&gain, (const void __user *)arg,
            sizeof(struct imx219_gain))) {
            pr_err("%s:Failed to get gain from user\n", __func__);
            return -EFAULT;
        }
        err = imx219_set_gain(info, gain);
        break;
    }
    case IMX219_IOCTL_GET_STATUS:
    {
        u8 status;

        err = imx219_get_status(info, &status);
        if (err)
            return err;
        if (copy_to_user((void __user *)arg, &status, 1)) {
            pr_err("%s:Failed to copy status to user\n", __func__);
            return -EFAULT;
```

```
        }
        return 0;
    }
    case IMX219_IOCTL_GET_FUSEID:
    {
        err = imx219_get_sensor_id(info);

        if (err) {
            pr_err("%s:Failed to get fuse id info.\n", __func__);
            return err;
        }
        if (copy_to_user((void __user *)arg, &info->fuse_id,
                sizeof(struct nvc_fuseid))) {
            pr_info("%s:Failed to copy fuseid\n", __func__);
            return -EFAULT;
        }
        return 0;
    }
    case IMX219_IOCTL_SET_GROUP_HOLD:
    {
        struct imx219_ae ae;
        if (copy_from_user(&ae, (const void __user *)arg,
            sizeof(struct imx219_ae))) {
            pr_info("%s:fail group hold\n", __func__);
            return -EFAULT;
        }
        return imx219_set_group_hold(info, &ae);
    }
    case IMX219_IOCTL_GET_AFDAT:
    {
        err = imx219_get_af_data(info);

        if (err) {
            pr_err("%s:Failed to get af data.\n", __func__);
            return err;
        }
        if (copy_to_user((void __user *)arg, info->afdat, 4)) {
            pr_err("%s:Failed to copy status to user\n", __func__);
            return -EFAULT;
        }
        return 0;
    }
    case IMX219_IOCTL_SET_FLASH_MODE:
    {
        dev_dbg(&info->i2c_client->dev,
            "IMX219_IOCTL_SET_FLASH_MODE not used\n");
        return -ENODEV;/* not support on sensor strobe */
    }
    case IMX219_IOCTL_GET_FLASH_CAP:
        return -ENODEV;/* not support on sensor strobe */
    default:
        pr_err("%s:unknown cmd: %u\n", __func__, cmd);
        err = -EINVAL;
    }

    return err;
}

static int
imx219_open(struct inode *inode, struct file *file)
{
    struct miscdevice   *miscdev = file->private_data;
    struct imx219_info *info;

    info = container_of(miscdev, struct imx219_info, miscdev_info);
    /* check if the device is in use */
    if (atomic_xchg(&info->in_use, 1)) {
        pr_info("%s:BUSY!\n", __func__);
        return -EBUSY;
    }
```

```
    file->private_data = info;

    return 0;
}

static int
imx219_release(struct inode *inode, struct file *file)
{
    struct imx219_info *info = file->private_data;

    file->private_data = NULL;

    /* warn if device is already released */
    WARN_ON(!atomic_xchg(&info->in_use, 0));
    return 0;
}

static int imx219_regulator_get(struct imx219_info *info,
                struct regulator **vreg, char vreg_name[])
{
    struct regulator *reg = NULL;
    int err = 0;

    reg = regulator_get(&info->i2c_client->dev, vreg_name);
    if (unlikely(IS_ERR(reg))) {
        dev_err(&info->i2c_client->dev, "%s %s ERR: %d\n",
            __func__, vreg_name, (int)reg);
        err = PTR_ERR(reg);
        reg = NULL;
    } else
        dev_dbg(&info->i2c_client->dev, "%s: %s\n",
            __func__, vreg_name);

    *vreg = reg;
    return err;
}

static int imx219_power_get(struct imx219_info *info)
{
    struct imx219_power_rail *pw = &info->power;
    int err = 0;

    err |= imx219_regulator_get(info, &pw->avdd, "vana"); /* ananlog 2.7v */
    err |= imx219_regulator_get(info, &pw->dvdd, "vdig"); /* digital 1.2v */
    err |= imx219_regulator_get(info, &pw->iovdd, "dovdd"); /* IO 1.8v */
    err |= imx219_regulator_get(info, &pw->vdd_af, "vdd_af1"); /* IO 1.8v */

    return err;
}

static int imx219_power_put(struct imx219_power_rail *pw)
{
    if (unlikely(!pw))
        return -EFAULT;

    if (likely(pw->avdd))
        regulator_put(pw->avdd);

    if (likely(pw->iovdd))
        regulator_put(pw->iovdd);

    if (likely(pw->dvdd))
        regulator_put(pw->dvdd);

    pw->avdd = NULL;
    pw->iovdd = NULL;
    pw->dvdd = NULL;

    return 0;
```

```
}

static const struct file_operations imx219_fileops = {
    .owner = THIS_MODULE,
    .open = imx219_open,
    .unlocked_ioctl = imx219_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl = imx219_ioctl,
#endif
    .release = imx219_release,
};

static struct miscdevice imx219_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "imx219",
    .fops = &imx219_fileops,
};

#ifdef CONFIG_DEBUG_FS
static int imx219_stats_show(struct seq_file *s, void *data)
{
    static struct imx219_info *info;

    seq_printf(s, "%-20s : %-20s\n", "Name", "imx219-debugfs-testing");
    seq_printf(s, "%-20s : 0x%X\n", "Current i2c-offset Addr",
            info->debug_i2c_offset);
    seq_printf(s, "%-20s : 0x%X\n", "DC BLC Enabled",
            info->debug_i2c_offset);
    return 0;
}

static int imx219_stats_open(struct inode *inode, struct file *file)
{
    return single_open(file, imx219_stats_show, inode->i_private);
}

static const struct file_operations imx219_stats_fops = {
    .open       = imx219_stats_open,
    .read       = seq_read,
    .llseek     = seq_lseek,
    .release    = single_release,
};

static int debug_i2c_offset_w(void *data, u64 val)
{
    struct imx219_info *info = (struct imx219_info *)(data);
    dev_info(&info->i2c_client->dev,
            "imx219:%s setting i2c offset to 0x%X\n",
            __func__, (u32)val);
    info->debug_i2c_offset = (u32)val;
    dev_info(&info->i2c_client->dev,
            "imx219:%s new i2c offset is 0x%X\n", __func__,
            info->debug_i2c_offset);
    return 0;
}

static int debug_i2c_offset_r(void *data, u64 *val)
{
    struct imx219_info *info = (struct imx219_info *)(data);
    *val = (u64)info->debug_i2c_offset;
    dev_info(&info->i2c_client->dev,
            "imx219:%s reading i2c offset is 0x%X\n", __func__,
            info->debug_i2c_offset);
    return 0;
}

static int debug_i2c_read(void *data, u64 *val)
{
    struct imx219_info *info = (struct imx219_info *)(data);
    u8 temp1 = 0;
```

```
    u8 temp2 = 0;
    dev_info(&info->i2c_client->dev,
            "imx219:%s reading offset 0x%X\n", __func__,
            info->debug_i2c_offset);
    if (imx219_read_reg(info->i2c_client,
                info->debug_i2c_offset, &temp1)
        || imx219_read_reg(info->i2c_client,
            info->debug_i2c_offset+1, &temp2)) {
        dev_err(&info->i2c_client->dev,
                "imx219:%s failed\n", __func__);
        return -EIO;
    }
    dev_info(&info->i2c_client->dev,
            "imx219:%s read value is 0x%X\n", __func__,
            temp1<<8 | temp2);
    *val = (u64)(temp1<<8 | temp2);
    return 0;
}

static int debug_i2c_write(void *data, u64 val)
{
    struct imx219_info *info = (struct imx219_info *)(data);
    dev_info(&info->i2c_client->dev,
            "imx219:%s writing 0x%X to offset 0x%X\n", __func__,
            (u8)val, info->debug_i2c_offset);
    if (imx219_write_reg(info->i2c_client,
                info->debug_i2c_offset, (u8)val)) {
        dev_err(&info->i2c_client->dev, "imx219:%s failed\n", __func__);
        return -EIO;
    }
    return 0;
}

DEFINE_SIMPLE_ATTRIBUTE(i2c_offset_fops, debug_i2c_offset_r,
        debug_i2c_offset_w, "0x%llx\n");
DEFINE_SIMPLE_ATTRIBUTE(i2c_read_fops, debug_i2c_read,
        /*debug_i2c_dummy_w*/ NULL, "0x%llx\n");
DEFINE_SIMPLE_ATTRIBUTE(i2c_write_fops, /*debug_i2c_dummy_r*/NULL,
        debug_i2c_write, "0x%llx\n");

static int imx219_debug_init(struct imx219_info *info)
{
    dev_dbg(&info->i2c_client->dev, "%s", __func__);

    info->debugfs_root = debugfs_create_dir(imx219_device.name, NULL);

    if (!info->debugfs_root)
        goto err_out;

    if (!debugfs_create_file("stats", S_IRUGO,
            info->debugfs_root, info, &imx219_stats_fops))
        goto err_out;

    if (!debugfs_create_file("offset", S_IRUGO | S_IWUSR,
            info->debugfs_root, info, &i2c_offset_fops))
        goto err_out;

    if (!debugfs_create_file("read", S_IRUGO,
            info->debugfs_root, info, &i2c_read_fops))
        goto err_out;

    if (!debugfs_create_file("write", S_IWUSR,
            info->debugfs_root, info, &i2c_write_fops))
        goto err_out;

    return 0;

err_out:
    dev_err(&info->i2c_client->dev, "ERROR:%s failed", __func__);
    debugfs_remove_recursive(info->debugfs_root);
```

```
    return -ENOMEM;
}
#endif

static int
imx219_probe(struct i2c_client *client,
             const struct i2c_device_id *id)
{
    struct imx219_info *info;
    int err;
    const char *mclk_name;

    pr_info("[IMX219]: probing sensor.\n");

    info = devm_kzalloc(&client->dev,
            sizeof(struct imx219_info), GFP_KERNEL);
    if (!info) {
        pr_err("%s:Unable to allocate memory!\n", __func__);
        return -ENOMEM;
    }

    info->pdata = client->dev.platform_data;
    info->i2c_client = client;
    atomic_set(&info->in_use, 0);
    info->mode = -1;
    info->afdat_read = false;

    mclk_name = info->pdata->mclk_name ?
            info->pdata->mclk_name : "default_mclk";
    info->mclk = devm_clk_get(&client->dev, mclk_name);
    if (IS_ERR(info->mclk)) {
        dev_err(&client->dev, "%s: unable to get clock %s\n",
            __func__, mclk_name);
        return PTR_ERR(info->mclk);
    }

    imx219_power_get(info);

    memcpy(&info->miscdev_info,
        &imx219_device,
        sizeof(struct miscdevice));

    err = misc_register(&info->miscdev_info);
    if (err) {
        pr_err("%s:Unable to register misc device!\n", __func__);
        goto imx219_probe_fail;
    }

    i2c_set_clientdata(client, info);
    /* create debugfs interface */
#ifdef CONFIG_DEBUG_FS
    imx219_debug_init(info);
#endif
    info->sysedpc = sysedp_create_consumer("imx219", "imx219");
    return 0;

imx219_probe_fail:
    imx219_power_put(&info->power);

    return err;
}

static int
imx219_remove(struct i2c_client *client)
{
    struct imx219_info *info;
    info = i2c_get_clientdata(client);
    misc_deregister(&imx219_device);

    imx219_power_put(&info->power);
```

```
#ifdef CONFIG_DEBUG_FS
    debugfs_remove_recursive(info->debugfs_root);
#endif
    sysedp_free_consumer(info->sysedpc);
    return 0;
}

static const struct i2c_device_id imx219_id[] = {
    { "imx219", 0 },
    { }
};

MODULE_DEVICE_TABLE(i2c, imx219_id);

static struct i2c_driver imx219_i2c_driver = {
    .driver = {
        .name = "imx219",
        .owner = THIS_MODULE,
    },
    .probe = imx219_probe,
    .remove = imx219_remove,
    .id_table = imx219_id,
};

static int __init imx219_init(void)
{
    pr_info("[IMX219] sensor driver loading\n");
    return i2c_add_driver(&imx219_i2c_driver);
}

static void __exit imx219_exit(void)
{
    i2c_del_driver(&imx219_i2c_driver);
}

module_init(imx219_init);
module_exit(imx219_exit);
```