```
/*
 * ov5647_v4l2.c - ov5647 sensor driver
 *
 * Copyright (c) 2013-2016, 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 <dt-bindings/gpio/tegra-gpio.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/gpio.h>
#include <linux/module.h>

#include <linux/seq_file.h>
#include <linux/of.h>
#include <linux/of_device.h>
#include <linux/of_gpio.h>

#include <media/v4l2-chip-ident.h>
#include <media/camera_common.h>
#include <media/ov5647.h>

#include "cam_dev/camera_gpio.h"

#include "ov5647_mode_tbls.h"

//#define OV5647_MAX_COARSE_DIFF                    7
#define OV5647_MAX_COARSE_DIFF                  8

#define OV5647_GAIN_SHIFT                       0
#define OV5647_MIN_GAIN                     0x00
#define OV5647_MAX_GAIN                     0x3FF
#define OV5647_MAX_UNREAL_GAIN              (0x0F80)
//#define OV5647_MIN_FRAME_LENGTH               (0x0)
#define OV5647_MIN_FRAME_LENGTH             (0)
#define OV5647_MAX_FRAME_LENGTH             (0x3fff)
#define OV5647_MIN_EXPOSURE_COARSE          16
//#define OV5647_MIN_EXPOSURE_COARSE            (0x0016)
#define OV5647_MAX_EXPOSURE_COARSE  \
    (OV5647_MAX_FRAME_LENGTH-OV5647_MAX_COARSE_DIFF)

#define OV5647_DEFAULT_GAIN                 OV5647_MIN_GAIN
//#define OV5647_DEFAULT_FRAME_LENGTH           (0x07B0)
//#define OV5647_DEFAULT_FRAME_LENGTH             (1104)
#define OV5647_DEFAULT_FRAME_LENGTH         (0x07C0)
//#define OV5647_DEFAULT_EXPOSURE_COARSE    OV5647_DEFAULT_FRAME_LENGTH-OV5647_MAX_COARSE_DIFF)
#define OV5647_DEFAULT_EXPOSURE_COARSE  100

#define OV5647_DEFAULT_MODE             OV5647_MODE_1920X1080
#define OV5647_DEFAULT_WIDTH              1920
#define OV5647_DEFAULT_HEIGHT             1080
#define OV5647_DEFAULT_DATAFMT          V4L2_MBUS_FMT_SRGGB10_1X10
#define OV5647_DEFAULT_CLK_FREQ         24000000
#define OV5647_DEFAULT_MAX_FPS          30

struct ov5647 {
    struct camera_common_power_rail power;
    int             numctrls;
    struct v4l2_ctrl_handler    ctrl_handler;
```

```
//  struct camera_common_eeprom_data eeprom[OV5647_EEPROM_NUM_BLOCKS];
//  u8              eeprom_buf[OV5647_EEPROM_SIZE];
    struct i2c_client       *i2c_client;
    struct v4l2_subdev      *subdev;
    struct media_pad        pad;

    int             reg_offset;

    s32             group_hold_prev;
    bool                group_hold_en;
    struct regmap           *regmap;
    struct camera_common_data   *s_data;
    struct camera_common_pdata  *pdata;
    struct v4l2_ctrl        *ctrls[];
};

static struct regmap_config ov5647_regmap_config = {
    .reg_bits = 16,
    .val_bits = 8,
};

static int ov5647_g_volatile_ctrl(struct v4l2_ctrl *ctrl);
static int ov5647_s_ctrl(struct v4l2_ctrl *ctrl);
static void ov5647_update_ctrl_range(struct ov5647 *priv, s32 frame_length);

static const struct v4l2_ctrl_ops ov5647_ctrl_ops = {
    .g_volatile_ctrl = ov5647_g_volatile_ctrl,
    .s_ctrl     = ov5647_s_ctrl,
};

static struct v4l2_ctrl_config ctrl_config_list[] = {
/* Do not change the name field for the controls! */
    {
        .ops = &ov5647_ctrl_ops,
        .id = V4L2_CID_GAIN,
        .name = "Gain",
        .type = V4L2_CTRL_TYPE_INTEGER,
        .flags = V4L2_CTRL_FLAG_SLIDER,
        .min = OV5647_MIN_GAIN,
        .max = OV5647_MAX_GAIN,
        .def = OV5647_DEFAULT_GAIN,
        .step = 1,
    },
    {
        .ops = &ov5647_ctrl_ops,
        .id = V4L2_CID_FRAME_LENGTH,
        .name = "Frame Length",
        .type = V4L2_CTRL_TYPE_INTEGER,
        .flags = V4L2_CTRL_FLAG_SLIDER,
        .min = OV5647_MIN_FRAME_LENGTH,
        .max = OV5647_MAX_FRAME_LENGTH,
        .def = OV5647_DEFAULT_FRAME_LENGTH,
        .step = 1,
    },
    {
        .ops = &ov5647_ctrl_ops,
        .id = V4L2_CID_COARSE_TIME,
        .name = "Coarse Time",
        .type = V4L2_CTRL_TYPE_INTEGER,
        .flags = V4L2_CTRL_FLAG_SLIDER,
        .min = OV5647_MIN_EXPOSURE_COARSE,
        .max = OV5647_MAX_EXPOSURE_COARSE,
        .def = OV5647_DEFAULT_EXPOSURE_COARSE,
        .step = 1,
    },
    {
        .ops = &ov5647_ctrl_ops,
        .id = V4L2_CID_COARSE_TIME_SHORT,
        .name = "Coarse Time Short",
        .type = V4L2_CTRL_TYPE_INTEGER,
```

```
        .flags = V4L2_CTRL_FLAG_SLIDER,
        .min = OV5647_MIN_EXPOSURE_COARSE,
        .max = OV5647_MAX_EXPOSURE_COARSE,
        .def = OV5647_DEFAULT_EXPOSURE_COARSE,
        .step = 1,
    },
    {
        .ops = &ov5647_ctrl_ops,
        .id = V4L2_CID_GROUP_HOLD,
        .name = "Group Hold",
        .type = V4L2_CTRL_TYPE_INTEGER_MENU,
        .min = 0,
        .max = ARRAY_SIZE(switch_ctrl_qmenu) - 1,
        .menu_skip_mask = 0,
        .def = 0,
        .qmenu_int = switch_ctrl_qmenu,
    },
    {
        .ops = &ov5647_ctrl_ops,
        .id = V4L2_CID_HDR_EN,
        .name = "HDR enable",
        .type = V4L2_CTRL_TYPE_INTEGER_MENU,
        .min = 0,
        .max = ARRAY_SIZE(switch_ctrl_qmenu) - 1,
        .menu_skip_mask = 0,
        .def = 0,
        .qmenu_int = switch_ctrl_qmenu,
    },
/*
    {
        .ops = &ov5647_ctrl_ops,
        .id = V4L2_CID_EEPROM_DATA,
        .name = "EEPROM Data",
        .type = V4L2_CTRL_TYPE_STRING,
        .flags = V4L2_CTRL_FLAG_VOLATILE,
        .min = 0,
        .max = OV5647_EEPROM_STR_SIZE,
        .step = 2,
    },
    {
        .ops = &ov5647_ctrl_ops,
        .id = V4L2_CID_OTP_DATA,
        .name = "OTP Data",
        .type = V4L2_CTRL_TYPE_STRING,
        .flags = V4L2_CTRL_FLAG_READ_ONLY,
        .min = 0,
        .max = OV5647_OTP_STR_SIZE,
        .step = 2,
    },
    {
        .ops = &ov5647_ctrl_ops,
        .id = V4L2_CID_FUSE_ID,
        .name = "Fuse ID",
        .type = V4L2_CTRL_TYPE_STRING,
        .flags = V4L2_CTRL_FLAG_READ_ONLY,
        .min = 0,
        .max = OV5647_FUSE_ID_STR_SIZE,
        .step = 2,
    },
*/
};

static inline void ov5647_get_frame_length_regs(ov5647_reg *regs,
                u32 frame_length)
{
    regs->addr = OV5647_FRAME_LENGTH_ADDR_MSB;
    regs->val = (frame_length >> 8) & 0xff;
    (regs + 1)->addr = OV5647_FRAME_LENGTH_ADDR_LSB;
    (regs + 1)->val = (frame_length) & 0xff;
}
```

```
static inline void ov5647_get_coarse_time_regs(ov5647_reg *regs,
                u32 coarse_time)
{
    regs->addr = OV5647_COARSE_TIME_ADDR_1;
    regs->val = (coarse_time >> 12) & 0xff;
    (regs + 1)->addr = OV5647_COARSE_TIME_ADDR_2;
    (regs + 1)->val = (coarse_time >> 4) & 0xff;
    (regs + 2)->addr = OV5647_COARSE_TIME_ADDR_3;
    (regs + 2)->val = (coarse_time & 0xf) << 4;
  //(regs + 3)->addr = 0x3212;
  //(regs + 3)->val = 0x10;
  //(regs + 4)->addr = 0x3212;
  //(regs + 4)->val = 0xA0;

}

static inline void ov5647_get_coarse_time_short_regs(ov5647_reg *regs,
                u32 coarse_time)
{
    regs->addr = OV5647_COARSE_TIME_SHORT_ADDR_1;
    regs->val = (coarse_time >> 12) & 0xff;
    (regs + 1)->addr = OV5647_COARSE_TIME_SHORT_ADDR_2;
    (regs + 1)->val = (coarse_time >> 4) & 0xff;
    (regs + 2)->addr = OV5647_COARSE_TIME_SHORT_ADDR_3;
    (regs + 2)->val = (coarse_time & 0xf) << 4;
}

static inline void ov5647_get_gain_regs(ov5647_reg *regs,
                u16 gain)
{
    regs->addr = OV5647_GAIN_ADDR_MSB;
    regs->val = (gain >> 8) & 0xff;

    (regs + 1)->addr = OV5647_GAIN_ADDR_LSB;
    (regs + 1)->val = (gain) & 0xff;

  //(regs + 2)->addr = 0x3212;
    //(regs + 2)->val = 0x00;
}

static int test_mode;
module_param(test_mode, int, 0644);

static inline int ov5647_read_reg(struct camera_common_data *s_data,
                u16 addr, u8 *val)
{
    struct ov5647 *priv = (struct ov5647 *)s_data->priv;

    return regmap_read(priv->regmap, addr, (unsigned int *) val);
}

static int ov5647_write_reg(struct camera_common_data *s_data, u16 addr, u8 val)
{
    int err;
    struct ov5647 *priv = (struct ov5647 *)s_data->priv;

    err = regmap_write(priv->regmap, addr, val);
    if (err)
        pr_err("%s:i2c write failed, %x = %x\n",
            __func__, addr, val);

    return err;
}

static int ov5647_write_table(struct ov5647 *priv,
                 const ov5647_reg table[])
{
    return regmap_util_write_table_8(priv->regmap,
```

```
                    table,
                    NULL, 0,
                    OV5647_TABLE_WAIT_MS,
                    OV5647_TABLE_END);
}

/*
static void ov5647_gpio_set(struct ov5647 *priv,
                unsigned int gpio, int val)
{
    if (priv->pdata->use_cam_gpio)
  {
      dev_info(&priv->i2c_client->dev, "%s: Cam GPIO set gpio value: %d\n", __func__, val);
        cam_gpio_ctrl(priv->i2c_client, gpio, val, 1);
  }
    else {
      dev_info(&priv->i2c_client->dev, "%s: Not Cam GPIO set gpio value: %d\n", __func__, val);
        if (gpio_cansleep(gpio))
            gpio_set_value_cansleep(gpio, val);
        else
            gpio_set_value(gpio, val);
    }
}
*/

static int ov5647_power_on(struct camera_common_data *s_data)
{
    int err = 0;
    struct ov5647 *priv = (struct ov5647 *)s_data->priv;
    struct camera_common_power_rail *pw = &priv->power;

    dev_info(&priv->i2c_client->dev, "%s: power on\n", __func__);

    if (priv->pdata && priv->pdata->power_on) {
        err = priv->pdata->power_on(pw);
        if (err)
            pr_err("%s failed.\n", __func__);
        else
            pw->state = SWITCH_ON;
        return err;
    }

  if (gpio_cansleep(pw->reset_gpio))
    gpio_set_value_cansleep(pw->reset_gpio, 0);
  else
    gpio_set_value(pw->reset_gpio, 0);
  usleep_range(10, 20);

  usleep_range(1, 2);
  if (gpio_cansleep(pw->reset_gpio))
    gpio_set_value_cansleep(pw->reset_gpio, 1);
  else
    gpio_set_value(pw->reset_gpio, 1);

  usleep_range(1, 2);
  clk_set_rate(pw->mclk, OV5647_DEFAULT_CLK_FREQ);
  clk_prepare_enable(pw->mclk);

    /* datasheet fig 2-9: t3 */
    //usleep_range(1350, 1360);
  usleep_range(7400, 7410);

    pw->state = SWITCH_ON;
    return 0;

}

static int ov5647_power_off(struct camera_common_data *s_data)
{
    //int err = 0;
```

```
    struct ov5647 *priv = (struct ov5647 *)s_data->priv;
    struct camera_common_power_rail *pw = &priv->power;

    dev_info(&priv->i2c_client->dev, "%s: power off\n", __func__);


  /*
  usleep_range(1, 2);
  if (gpio_cansleep(pw->reset_gpio))
    gpio_set_value_cansleep(pw->reset_gpio, 0);
  else
    gpio_set_value(pw->reset_gpio, 0);
  usleep_range(1, 2);
  */


    /* datasheet 2.9: reset requires ~2ms settling time*/
    usleep_range(2000, 2010);
  clk_disable_unprepare(pw->mclk);
  pw->state = SWITCH_OFF;

    return 0;
}

static int ov5647_power_put(struct ov5647 *priv)
{
    struct camera_common_power_rail *pw = &priv->power;

    if (unlikely(!pw))
        return -EFAULT;

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

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

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

    /*
    if (priv->pdata->use_cam_gpio)
        cam_gpio_deregister(priv->i2c_client, pw->pwdn_gpio);
    else {
    */
        //gpio_free(pw->pwdn_gpio);
        gpio_free(pw->reset_gpio);
    //}

    return 0;
}

static int ov5647_power_get(struct ov5647 *priv)
{
    struct camera_common_power_rail *pw = &priv->power;
    struct camera_common_pdata *pdata = priv->pdata;
    const char *mclk_name;
    const char *parentclk_name;
    struct clk *parent;
    int err = 0;

    mclk_name = priv->pdata->mclk_name ?
            priv->pdata->mclk_name : "cam_mclk1";
    pw->mclk = devm_clk_get(&priv->i2c_client->dev, mclk_name);
    if (IS_ERR(pw->mclk)) {
        dev_err(&priv->i2c_client->dev,
            "unable to get clock %s\n", mclk_name);
        return PTR_ERR(pw->mclk);
```

```
    }

    parentclk_name = priv->pdata->parentclk_name;
    if (parentclk_name) {
        parent = devm_clk_get(&priv->i2c_client->dev, parentclk_name);
        if (IS_ERR(parent))
            dev_err(&priv->i2c_client->dev,
                "unable to get parent clcok %s",
                parentclk_name);
        else
            clk_set_parent(pw->mclk, parent);
    }


    /* analog 2.8v */
    //err |= camera_common_regulator_get(priv->i2c_client,
    //  &pw->avdd, pdata->regulators.avdd);
    /* IO 1.8v */
    //err |= camera_common_regulator_get(priv->i2c_client,
    //      &pw->iovdd, pdata->regulators.iovdd);

    if (!err) {
        pw->reset_gpio = pdata->reset_gpio;
        //pw->pwdn_gpio = pdata->pwdn_gpio;
    }

/*
    if (priv->pdata->use_cam_gpio) {
        err = cam_gpio_register(priv->i2c_client, pw->pwdn_gpio);
        if (err)
            dev_err(&priv->i2c_client->dev,
                "%s ERR can't register cam gpio %u!\n",
                 __func__, pw->pwdn_gpio);
    } else {
*/
//      gpio_request(pw->pwdn_gpio, "cam_pwdn_gpio");
        gpio_request(pw->reset_gpio, "cam_reset_gpio");
/*
    }
*/


    pw->state = SWITCH_OFF;
    return err;
}


static int ov5647_reset_camera(struct ov5647 *priv)
{
    struct camera_common_power_rail *pw = &priv->power;
    dev_info(&priv->i2c_client->dev, "%s: Reset\n", __func__);
    if (pw->reset_gpio)
  {
      dev_info(&priv->i2c_client->dev, "%s: Reset Low\n", __func__);
        //ov5647_gpio_set(priv, pw->reset_gpio, 0);
        if (gpio_cansleep(pw->reset_gpio))
            gpio_set_value_cansleep(pw->reset_gpio, 0);
        else
            gpio_set_value(pw->reset_gpio, 0);

  }
    usleep_range(2000, 2010);
    if (pw->reset_gpio)
  {
      dev_info(&priv->i2c_client->dev, "%s: Reset High\n", __func__);
        //ov5647_gpio_set(priv, pw->reset_gpio, 1);
        if (gpio_cansleep(pw->reset_gpio))
            gpio_set_value_cansleep(pw->reset_gpio, 1);
        else
            gpio_set_value(pw->reset_gpio, 1);
```

```
  }
  usleep_range(7400, 7410);

  //return ov5647_write_table(priv, ov5647_reset);
  return 0;
}

static int ov5647_set_gain(struct ov5647 *priv, s32 val);
static int ov5647_set_frame_length(struct ov5647 *priv, s32 val);
static int ov5647_set_coarse_time(struct ov5647 *priv, s32 val);
static int ov5647_set_coarse_time_short(struct ov5647 *priv, s32 val);

static int ov5647_s_stream(struct v4l2_subdev *sd, int enable)
{
    struct i2c_client *client = v4l2_get_subdevdata(sd);
    struct camera_common_data *s_data = to_camera_common_data(client);
    struct ov5647 *priv = (struct ov5647 *)s_data->priv;
    struct v4l2_control control;
    int err;

    dev_info(&client->dev, "%s++\n", __func__);

    if (!enable) {
        ov5647_update_ctrl_range(priv, OV5647_MAX_FRAME_LENGTH);

        return ov5647_write_table(priv,
            mode_table[OV5647_MODE_STOP_STREAM]);
    }

    err = ov5647_write_table(priv, mode_table[s_data->mode]);
    if (err)
        goto exit;

    /* write list of override regs for the asking frame length,
     * coarse integration time, and gain. Failures to write
     * overrides are non-fatal */
    control.id = V4L2_CID_GAIN;
    err = v4l2_g_ctrl(&priv->ctrl_handler, &control);
    err |= ov5647_set_gain(priv, control.value);
    if (err)
        dev_info(&client->dev, "%s: warning gain override failed\n",
            __func__);

    control.id = V4L2_CID_FRAME_LENGTH;
    err = v4l2_g_ctrl(&priv->ctrl_handler, &control);
    err |= ov5647_set_frame_length(priv, control.value);
    if (err)
        dev_info(&client->dev,
            "%s: warning frame length override failed\n",
            __func__);

    control.id = V4L2_CID_COARSE_TIME;
    err = v4l2_g_ctrl(&priv->ctrl_handler, &control);
    err |= ov5647_set_coarse_time(priv, control.value);
    if (err)
        dev_info(&client->dev,
            "%s: warning coarse time override failed\n",
            __func__);

  /*
    control.id = V4L2_CID_COARSE_TIME_SHORT;
    err = v4l2_g_ctrl(&priv->ctrl_handler, &control);
    err |= ov5647_set_coarse_time_short(priv, control.value);
    if (err)
        dev_info(&client->dev,
            "%s: warning coarse time short override failed\n",
            __func__);
  */
```

```
    err = ov5647_write_table(priv, mode_table[OV5647_MODE_START_STREAM]);
    if (err)
        goto exit;

    if (test_mode)
        err = ov5647_write_table(priv,
            mode_table[OV5647_MODE_TEST_PATTERN]);

    dev_info(&client->dev, "%s--\n", __func__);
    return 0;
exit:
    dev_info(&client->dev, "%s: error setting stream\n", __func__);
    return err;
}

static int ov5647_g_input_status(struct v4l2_subdev *sd, u32 *status)
{
    struct i2c_client *client = v4l2_get_subdevdata(sd);
    struct camera_common_data *s_data = to_camera_common_data(client);
    struct ov5647 *priv = (struct ov5647 *)s_data->priv;
    struct camera_common_power_rail *pw = &priv->power;

    *status = pw->state == SWITCH_ON;
    return 0;
}

static struct v4l2_subdev_video_ops ov5647_subdev_video_ops = {
    .s_stream   = ov5647_s_stream,
    .s_mbus_fmt = camera_common_s_fmt,
    .g_mbus_fmt = camera_common_g_fmt,
    .try_mbus_fmt   = camera_common_try_fmt,
    .enum_mbus_fmt  = camera_common_enum_fmt,
    .g_mbus_config  = camera_common_g_mbus_config,
    .g_input_status = ov5647_g_input_status,
    .enum_framesizes    = camera_common_enum_framesizes,
    .enum_frameintervals    = camera_common_enum_frameintervals,
};

static struct v4l2_subdev_core_ops ov5647_subdev_core_ops = {
    .s_power    = camera_common_s_power,
};

static int ov5647_get_fmt(struct v4l2_subdev *sd,
        struct v4l2_subdev_fh *fh,
        struct v4l2_subdev_format *format)
{
    return camera_common_g_fmt(sd, &format->format);
}

static int ov5647_set_fmt(struct v4l2_subdev *sd,
        struct v4l2_subdev_fh *fh,
    struct v4l2_subdev_format *format)
{
    int ret;

    if (format->which == V4L2_SUBDEV_FORMAT_TRY)
        ret = camera_common_try_fmt(sd, &format->format);
    else
        ret = camera_common_s_fmt(sd, &format->format);

    return ret;
}

static struct v4l2_subdev_pad_ops ov5647_subdev_pad_ops = {
    .enum_mbus_code = camera_common_enum_mbus_code,
    .set_fmt = ov5647_set_fmt,
    .get_fmt = ov5647_get_fmt,
};

static struct v4l2_subdev_ops ov5647_subdev_ops = {
```

```
    .core   = &ov5647_subdev_core_ops,
    .video  = &ov5647_subdev_video_ops,
    .pad    = &ov5647_subdev_pad_ops,
};

static struct of_device_id ov5647_of_match[] = {
    { .compatible = "nvidia,ov5647", },
    { },
};

static struct camera_common_sensor_ops ov5647_common_ops = {
    .power_on = ov5647_power_on,
    .power_off = ov5647_power_off,
    .write_reg = ov5647_write_reg,
    .read_reg = ov5647_read_reg,
};

static int ov5647_set_group_hold(struct ov5647 *priv)
{
    int err;
    int gh_prev = switch_ctrl_qmenu[priv->group_hold_prev];

    if (priv->group_hold_en == true && gh_prev == SWITCH_OFF) {
        /* enter group hold */
        err = ov5647_write_reg(priv->s_data,
                       OV5647_GROUP_HOLD_ADDR, 0x00);
        if (err)
            goto fail;

        priv->group_hold_prev = 1;

        dev_info(&priv->i2c_client->dev,
             "%s: enter group hold\n", __func__);
    } else if (priv->group_hold_en == false && gh_prev == SWITCH_ON) {
        /* leave group hold */
        err = ov5647_write_reg(priv->s_data,
                       OV5647_GROUP_HOLD_ADDR, 0x10);
        if (err)
            goto fail;

        err = ov5647_write_reg(priv->s_data,
                       OV5647_GROUP_HOLD_ADDR, 0xA0);
        if (err)
            goto fail;

        priv->group_hold_prev = 0;

        dev_info(&priv->i2c_client->dev,
             "%s: leave group hold\n", __func__);
    }
    return 0;

fail:
    dev_info(&priv->i2c_client->dev,
         "%s: Group hold control error\n", __func__);
    return err;
}

/*
static u16 ov5647_to_real_gain(u32 rep, int shift)
{
    u16 gain;
    int gain_int;
    int gain_dec;
    int min_int = (1 << shift);

    if (rep < OV5647_MIN_GAIN)
        rep = OV5647_MIN_GAIN;
    else if (rep > OV5647_MAX_GAIN)
        rep = OV5647_MAX_GAIN;
```

```
    gain_int = (int)(rep >> shift);
    gain_dec = (int)(rep & ~(0xffff << shift));

    gain = ((gain_int * min_int + gain_dec) * 32 + min_int) / (2 * min_int);

    return gain;
}
*/

#define GAIN_WRITE_LENGTH 2
static int ov5647_set_gain(struct ov5647 *priv, s32 val)
{
    //ov5647_reg reg_list[2];
    ov5647_reg reg_list[GAIN_WRITE_LENGTH];
    int err;
    u16 gain;
    int i;

    if (!priv->group_hold_prev)
        ov5647_set_group_hold(priv);

    /* translate value */
    //gain = ov5647_to_real_gain((u32)val, OV5647_GAIN_SHIFT);
  gain = (u32)val;

    ov5647_get_gain_regs(reg_list, gain);
    dev_info(&priv->i2c_client->dev,
         "%s: gain %04x val: %04x\n", __func__, val, gain);

    for (i = 0; i < GAIN_WRITE_LENGTH; i++) {
        err = ov5647_write_reg(priv->s_data, reg_list[i].addr,
             reg_list[i].val);
        if (err)
            goto fail;
    }

    return 0;

fail:
    dev_info(&priv->i2c_client->dev,
         "%s: GAIN control error\n", __func__);
    return err;
}

static void ov5647_update_ctrl_range(struct ov5647 *priv, s32 frame_length)
{
    struct v4l2_ctrl *ctrl = NULL;
    int ctrl_ids[2] = {V4L2_CID_COARSE_TIME, V4L2_CID_COARSE_TIME_SHORT};
    //int ctrl_ids[1] = {V4L2_CID_COARSE_TIME};
    s32 max, min, def;
    int i, j;

    for (i = 0; i < ARRAY_SIZE(ctrl_ids); i++) {
        for (j = 0; j < priv->numctrls; j++) {
            if (priv->ctrls[j]->id == ctrl_ids[i]) {
                ctrl = priv->ctrls[j];
                break;
            }
        }

        if (j == priv->numctrls) {
            dev_err(&priv->i2c_client->dev,
                "could not find ctrl: %x\n",
                ctrl_ids[i]);
            continue;
        }

        max = frame_length - OV5647_MAX_COARSE_DIFF;
        /* clamp the value in case above is negative */
```

```
        max = clamp_val(max, OV5647_MIN_EXPOSURE_COARSE,
            OV5647_MAX_EXPOSURE_COARSE);
        min = OV5647_MIN_EXPOSURE_COARSE;
        def = clamp_val(OV5647_DEFAULT_EXPOSURE_COARSE, min, max);
        if (__v4l2_ctrl_modify_range(ctrl, min, max, 1, def))
            dev_err(&priv->i2c_client->dev,
                "ctrl %x: range update failed\n",
                ctrl_ids[i]);
    }

}

static int ov5647_set_frame_length(struct ov5647 *priv, s32 val)
{
    ov5647_reg reg_list[2];
    int err;
    u32 frame_length;
    int i;

    if (!priv->group_hold_prev)
        ov5647_set_group_hold(priv);

    frame_length = (u32)val;

    ov5647_get_frame_length_regs(reg_list, frame_length);
    dev_info(&priv->i2c_client->dev,
         "%s: val: %d\n", __func__, frame_length);

    for (i = 0; i < 2; i++) {
        err = ov5647_write_reg(priv->s_data, reg_list[i].addr,
             reg_list[i].val);
        if (err)
            goto fail;
    }

    ov5647_update_ctrl_range(priv, val);
    return 0;

fail:
    dev_info(&priv->i2c_client->dev,
         "%s: FRAME_LENGTH control error\n", __func__);
    return err;
}

#define COARSE_TIME_LENGTH 3
static int ov5647_set_coarse_time(struct ov5647 *priv, s32 val)
{
    ov5647_reg reg_list[COARSE_TIME_LENGTH];
    int err;
    u32 coarse_time;
    int i;

    if (!priv->group_hold_prev)
        ov5647_set_group_hold(priv);

    coarse_time = (u32)val;

    ov5647_get_coarse_time_regs(reg_list, coarse_time);
    dev_info(&priv->i2c_client->dev,
         "%s: val: %d\n", __func__, coarse_time);

    for (i = 0; i < COARSE_TIME_LENGTH; i++) {
        err = ov5647_write_reg(priv->s_data, reg_list[i].addr,
             reg_list[i].val);
        if (err)
            goto fail;
    }

    return 0;
```

```
fail:
    dev_info(&priv->i2c_client->dev,
         "%s: COARSE_TIME control error\n", __func__);
    return err;
}

#define COARSE_TIME_SHORT_LENGTH 3
static int ov5647_set_coarse_time_short(struct ov5647 *priv, s32 val)
{
    ov5647_reg reg_list[COARSE_TIME_SHORT_LENGTH];
    int err;
    //struct v4l2_control hdr_control;
    //int hdr_en;
    u32 coarse_time_short;
    int i;

    if (!priv->group_hold_prev)
        ov5647_set_group_hold(priv);

    // check hdr enable ctrl
  /*
    hdr_control.id = V4L2_CID_HDR_EN;

    err = camera_common_g_ctrl(priv->s_data, &hdr_control);
    if (err < 0) {
        dev_err(&priv->i2c_client->dev,
            "%s: could not find device ctrl.\n", __func__);
        return err;
    }

    hdr_en = switch_ctrl_qmenu[hdr_control.value];
    if (hdr_en == SWITCH_OFF)
        return 0;
  */

    coarse_time_short = (u32)val;

    ov5647_get_coarse_time_short_regs(reg_list, coarse_time_short);
    dev_info(&priv->i2c_client->dev,
         "%s: val: %d\n", __func__, coarse_time_short);

    for (i = 0; i < COARSE_TIME_SHORT_LENGTH; i++) {
        err = ov5647_write_reg(priv->s_data, reg_list[i].addr,
             reg_list[i].val);
        if (err)
            goto fail;
    }

    return 0;

fail:
    dev_info(&priv->i2c_client->dev,
         "%s: COARSE_TIME_SHORT control error\n", __func__);
    return err;
}

/*
static int ov5647_eeprom_device_release(struct ov5647 *priv)
{
    int i;

    for (i = 0; i < OV5647_EEPROM_NUM_BLOCKS; i++) {
        if (priv->eeprom[i].i2c_client != NULL) {
            i2c_unregister_device(priv->eeprom[i].i2c_client);
            priv->eeprom[i].i2c_client = NULL;
        }
    }

    return 0;
}
```

```
static int ov5647_eeprom_device_init(struct ov5647 *priv)
{
    char *dev_name = "eeprom_ov5647";
    static struct regmap_config eeprom_regmap_config = {
        .reg_bits = 8,
        .val_bits = 8,
    };
    int i;
    int err;

    if (!priv->pdata->has_eeprom)
        return -EINVAL;

    for (i = 0; i < OV5647_EEPROM_NUM_BLOCKS; i++) {
        priv->eeprom[i].adap = i2c_get_adapter(
                priv->i2c_client->adapter->nr);
        memset(&priv->eeprom[i].brd, 0, sizeof(priv->eeprom[i].brd));
        strncpy(priv->eeprom[i].brd.type, dev_name,
                sizeof(priv->eeprom[i].brd.type));
        priv->eeprom[i].brd.addr = OV5647_EEPROM_ADDRESS + i;
        priv->eeprom[i].i2c_client = i2c_new_device(
                priv->eeprom[i].adap, &priv->eeprom[i].brd);

        priv->eeprom[i].regmap = devm_regmap_init_i2c(
            priv->eeprom[i].i2c_client, &eeprom_regmap_config);
        if (IS_ERR(priv->eeprom[i].regmap)) {
            err = PTR_ERR(priv->eeprom[i].regmap);
            ov5647_eeprom_device_release(priv);
            return err;
        }
    }

    return 0;
}

static int ov5647_read_eeprom(struct ov5647 *priv,
                struct v4l2_ctrl *ctrl)
{
    int err, i;

    for (i = 0; i < OV5647_EEPROM_NUM_BLOCKS; i++) {
        err = regmap_bulk_read(priv->eeprom[i].regmap, 0,
            &priv->eeprom_buf[i * OV5647_EEPROM_BLOCK_SIZE],
            OV5647_EEPROM_BLOCK_SIZE);
        if (err)
            return err;
    }

    for (i = 0; i < OV5647_EEPROM_SIZE; i++)
        sprintf(&ctrl->string[i*2], "%02x",
            priv->eeprom_buf[i]);
    return 0;
}

static int ov5647_write_eeprom(struct ov5647 *priv,
                char *string)
{
    int err;
    int i;
    u8 curr[3];
    unsigned long data;

    for (i = 0; i < OV5647_EEPROM_SIZE; i++) {
        curr[0] = string[i*2];
        curr[1] = string[i*2+1];
        curr[2] = '\0';

        err = kstrtol(curr, 16, &data);
        if (err) {
```

```
            dev_err(&priv->i2c_client->dev,
                "invalid eeprom string\n");
            return -EINVAL;
        }

        priv->eeprom_buf[i] = (u8)data;
        err = regmap_write(priv->eeprom[i >> 8].regmap,
                    i & 0xFF, (u8)data);
        if (err)
            return err;
        msleep(20);
    }
    return 0;
}
*/

/*
static int ov5647_read_otp_bank(struct ov5647 *priv,
                u8 *buf, int bank, u16 addr, int size)
{
    int err;

    // sleeps calls in the sequence below are for internal device
    // signal propagation as specified by sensor vendor

    usleep_range(10000, 11000);
    err = ov5647_write_table(priv, mode_table[OV5647_MODE_START_STREAM]);
    if (err)
        return err;

    err = ov5647_write_reg(priv->s_data, OV5647_OTP_BANK_SELECT_ADDR,
                0xC0 | bank);
    if (err)
        return err;

    err = ov5647_write_reg(priv->s_data, OV5647_OTP_LOAD_CTRL_ADDR, 0x01);
    if (err)
        return err;

    usleep_range(10000, 11000);
    err = regmap_bulk_read(priv->regmap, addr, buf, size);
    if (err)
        return err;

    err = ov5647_write_table(priv,
            mode_table[OV5647_MODE_STOP_STREAM]);
    if (err)
        return err;

    return 0;
}
*/

/*
static int ov5647_otp_setup(struct ov5647 *priv)
{
    int err;
    int i;
    struct v4l2_ctrl *ctrl;
    u8 otp_buf[OV5647_OTP_SIZE];

    err = camera_common_s_power(priv->subdev, true);
    if (err)
        return -ENODEV;

    for (i = 0; i < OV5647_OTP_NUM_BANKS; i++) {
        err = ov5647_read_otp_bank(priv,
                    &otp_buf[i * OV5647_OTP_BANK_SIZE],
                    i,
                    OV5647_OTP_BANK_START_ADDR,
```

```
                    OV5647_OTP_BANK_SIZE);
        if (err)
            return -ENODEV;
    }

    ctrl = v4l2_ctrl_find(&priv->ctrl_handler, V4L2_CID_OTP_DATA);
    if (!ctrl) {
        dev_err(&priv->i2c_client->dev,
            "could not find device ctrl.\n");
        return -EINVAL;
    }

    for (i = 0; i < OV5647_OTP_SIZE; i++)
        sprintf(&ctrl->string[i*2], "%02x",
            otp_buf[i]);
    ctrl->cur.string = ctrl->string;

    err = camera_common_s_power(priv->subdev, false);
    if (err)
        return -ENODEV;

    return 0;
}
*/

/*
static int ov5647_fuse_id_setup(struct ov5647 *priv)
{
    int err;
    int i;
    struct v4l2_ctrl *ctrl;
    u8 fuse_id[OV5647_FUSE_ID_SIZE];

    err = camera_common_s_power(priv->subdev, true);
    if (err)
        return -ENODEV;

    err = ov5647_read_otp_bank(priv,
                &fuse_id[0],
                OV5647_FUSE_ID_OTP_BANK,
                OV5647_FUSE_ID_OTP_START_ADDR,
                OV5647_FUSE_ID_SIZE);
    if (err)
        return -ENODEV;

    ctrl = v4l2_ctrl_find(&priv->ctrl_handler, V4L2_CID_FUSE_ID);
    if (!ctrl) {
        dev_err(&priv->i2c_client->dev,
            "could not find device ctrl.\n");
        return -EINVAL;
    }

    for (i = 0; i < OV5647_FUSE_ID_SIZE; i++)
        sprintf(&ctrl->string[i*2], "%02x",
            fuse_id[i]);
    ctrl->cur.string = ctrl->string;

    err = camera_common_s_power(priv->subdev, false);
    if (err)
        return -ENODEV;

    return 0;
}
*/

static int ov5647_g_volatile_ctrl(struct v4l2_ctrl *ctrl)
{
    struct ov5647 *priv =
        container_of(ctrl->handler, struct ov5647, ctrl_handler);
    int err = 0;
```

```
    if (priv->power.state == SWITCH_OFF)
        return 0;

    switch (ctrl->id) {
/*
    case V4L2_CID_EEPROM_DATA:
        err = ov5647_read_eeprom(priv, ctrl);
        if (err)
            return err;
        break;
*/
    default:
            pr_err("%s: unknown ctrl id.\n", __func__);
            return -EINVAL;
    }

    return err;
}

static int ov5647_s_ctrl(struct v4l2_ctrl *ctrl)
{
    struct ov5647 *priv =
        container_of(ctrl->handler, struct ov5647, ctrl_handler);
    int err = 0;

    if (priv->power.state == SWITCH_OFF)
        return 0;

    switch (ctrl->id) {
    case V4L2_CID_GAIN:
        err = ov5647_set_gain(priv, ctrl->val);
        break;
    case V4L2_CID_FRAME_LENGTH:
        err = ov5647_set_frame_length(priv, ctrl->val);
        break;
    case V4L2_CID_COARSE_TIME:
        err = ov5647_set_coarse_time(priv, ctrl->val);
        break;
    case V4L2_CID_COARSE_TIME_SHORT:
        err = ov5647_set_coarse_time_short(priv, ctrl->val);
        break;
    case V4L2_CID_GROUP_HOLD:
        if (switch_ctrl_qmenu[ctrl->val] == SWITCH_ON) {
            priv->group_hold_en = true;
        } else {
            priv->group_hold_en = false;
            err = ov5647_set_group_hold(priv);
        }
        break;
    case V4L2_CID_EEPROM_DATA:
    break;
    case V4L2_CID_OTP_DATA:
    break;
    case V4L2_CID_FUSE_ID:
    break;
    case V4L2_CID_HDR_EN:
        break;
    default:
        pr_info("%s: unknown ctrl id.\n", __func__);
        return -EINVAL;
    }

    return err;
}

static int ov5647_ctrls_init(struct ov5647 *priv, bool eeprom_ctrl)
{
    struct i2c_client *client = priv->i2c_client;
    //struct camera_common_data *common_data = priv->s_data;
```

```
    struct v4l2_ctrl *ctrl;
    int numctrls;
    int err;
    int i;

    dev_info(&client->dev, "%s++\n", __func__);

    numctrls = ARRAY_SIZE(ctrl_config_list);
    v4l2_ctrl_handler_init(&priv->ctrl_handler, numctrls);

    for (i = 0; i < numctrls; i++) {
        /* Skip control 'V4L2_CID_EEPROM_DATA' if eeprom inint err */
/*
        if (ctrl_config_list[i].id == V4L2_CID_EEPROM_DATA) {
            if (!eeprom_ctrl) {
                common_data->numctrls -= 1;
                continue;
            }
        }
*/

        ctrl = v4l2_ctrl_new_custom(&priv->ctrl_handler,
            &ctrl_config_list[i], NULL);
        if (ctrl == NULL) {
            dev_err(&client->dev, "Failed to init %s ctrl\n",
                ctrl_config_list[i].name);
            continue;
        }

        if (ctrl_config_list[i].type == V4L2_CTRL_TYPE_STRING &&
            ctrl_config_list[i].flags & V4L2_CTRL_FLAG_READ_ONLY) {
            ctrl->string = devm_kzalloc(&client->dev,
                ctrl_config_list[i].max + 1, GFP_KERNEL);
            if (!ctrl->string)
                return -ENOMEM;
        }
        priv->ctrls[i] = ctrl;
    }

    priv->numctrls = numctrls;
    priv->subdev->ctrl_handler = &priv->ctrl_handler;
    if (priv->ctrl_handler.error) {
        dev_err(&client->dev, "Error %d adding controls\n",
            priv->ctrl_handler.error);
        err = priv->ctrl_handler.error;
        goto error;
    }

    err = v4l2_ctrl_handler_setup(&priv->ctrl_handler);
    if (err) {
        dev_err(&client->dev,
            "Error %d setting default controls\n", err);
        goto error;
    }

/*
    err = ov5647_otp_setup(priv);
    if (err) {
        dev_err(&client->dev,
            "Error %d reading otp data\n", err);
        goto error;
    }
*/

/*
    err = ov5647_fuse_id_setup(priv);
    if (err) {
        dev_err(&client->dev,
            "Error %d reading fuse id data\n", err);
        goto error;
```

```
    }
*/

    return 0;

error:
    v4l2_ctrl_handler_free(&priv->ctrl_handler);
    return err;
}

MODULE_DEVICE_TABLE(of, ov5647_of_match);

static struct camera_common_pdata *ov5647_parse_dt(struct i2c_client *client)
{
    struct device_node *node = client->dev.of_node;
    struct camera_common_pdata *board_priv_pdata;
    const struct of_device_id *match;
    int gpio;
    int err;

    if (!node)
        return NULL;

    match = of_match_device(ov5647_of_match, &client->dev);
    if (!match) {
        dev_err(&client->dev, "Failed to find matching dt id\n");
        return NULL;
    }

    board_priv_pdata = devm_kzalloc(&client->dev,
              sizeof(*board_priv_pdata), GFP_KERNEL);
    if (!board_priv_pdata)
        return NULL;

    err = camera_common_parse_clocks(client, board_priv_pdata);
    if (err) {
        dev_err(&client->dev, "Failed to find clocks\n");
        goto error;
    }

/*
    gpio = of_get_named_gpio(node, "pwdn-gpios", 0);
    if (gpio < 0) {
        dev_err(&client->dev, "pwdn gpios not in DT\n");
        goto error;
    }
    board_priv_pdata->pwdn_gpio = (unsigned int)gpio;
*/

    gpio = of_get_named_gpio(node, "reset-gpios", 0);
  dev_info(&client->dev, "%s: OV5647 Reset GPIO: %d\n", __func__, gpio);
    if (gpio < 0) {
        /* reset-gpio is not absoluctly needed */
        dev_info(&client->dev, "reset gpios not in DT\n");
        gpio = 0;
    }
    board_priv_pdata->reset_gpio = (unsigned int)gpio;

    board_priv_pdata->use_cam_gpio =
        of_property_read_bool(node, "cam,use-cam-gpio");

/*
    err = of_property_read_string(node, "avdd-reg",
            &board_priv_pdata->regulators.avdd);
    if (err) {
        dev_err(&client->dev, "avdd-reg not in DT\n");
        goto error;
    }
    err = of_property_read_string(node, "iovdd-reg",
            &board_priv_pdata->regulators.iovdd);
```

```
    if (err) {
        dev_err(&client->dev, "iovdd-reg not in DT\n");
        goto error;
    }
*/

/*
    board_priv_pdata->has_eeprom =
        of_property_read_bool(node, "has-eeprom");
*/

    return board_priv_pdata;

error:
    devm_kfree(&client->dev, board_priv_pdata);
    return NULL;
}

static int ov5647_open(struct v4l2_subdev *sd, struct v4l2_subdev_fh *fh)
{
    int err;
    struct i2c_client *client = v4l2_get_subdevdata(sd);
    struct camera_common_data *s_data = to_camera_common_data(client);
    struct ov5647 *priv = (struct ov5647 *)s_data->priv;


    dev_info(&client->dev, "%s:\n", __func__);

  err = ov5647_reset_camera(priv);
    if (err)
    dev_info(&client->dev, "%s: Error resetting camera\n", __func__);

    return 0;
}

static const struct v4l2_subdev_internal_ops ov5647_subdev_internal_ops = {
    .open = ov5647_open,
};

static const struct media_entity_operations ov5647_media_ops = {
    .link_validate = v4l2_subdev_link_validate,
};

static int ov5647_probe(struct i2c_client *client,
            const struct i2c_device_id *id)
{
    struct camera_common_data *common_data;
    struct device_node *node = client->dev.of_node;
    struct ov5647 *priv;
  struct soc_camera_link *ov5647_iclink;
    char debugfs_name[10];
    int err;

    pr_info("[OV5647]: probing v4l2 sensor.\n");

    if (!IS_ENABLED(CONFIG_OF) || !node)
        return -EINVAL;

    common_data = devm_kzalloc(&client->dev,
                sizeof(struct camera_common_data), GFP_KERNEL);
    if (!common_data)
        return -ENOMEM;

    priv = devm_kzalloc(&client->dev,
                sizeof(struct ov5647) + sizeof(struct v4l2_ctrl *) *
                ARRAY_SIZE(ctrl_config_list),
                GFP_KERNEL);
    if (!priv)
        return -ENOMEM;
```

```
    priv->regmap = devm_regmap_init_i2c(client, &ov5647_regmap_config);
    if (IS_ERR(priv->regmap)) {
        dev_err(&client->dev,
            "regmap init failed: %ld\n", PTR_ERR(priv->regmap));
        return -ENODEV;
    }

  //sprintf(node_name, "ov5647_%c", 'a' + client->adapter->nr - 30);
  //dev_info("Node Name: %s" % node_name);
  dev_info(&client->dev, "%s: Node Name: ov5647_a\n", __func__);
  //client->dev.of_node = of_find_node_by_name(NULL, node_name);
  client->dev.of_node = of_find_node_by_name(NULL, "ov5647_a");

  if (client->dev.of_node) {
      priv->pdata = ov5647_parse_dt(client);
    dev_info(&client->dev, "Found and parsed of_node in DT\n");
  }
  else {
    ov5647_iclink = client->dev.platform_data;
    priv->pdata = ov5647_iclink->dev_priv;
    dev_info(&client->dev, "Did not find of_node in DT\n");
  }

    if (!priv->pdata) {
        dev_err(&client->dev, "%s: unable to get platform data\n", __func__);
        return -EFAULT;
    }

    common_data->ops          = &ov5647_common_ops;
    common_data->ctrl_handler = &priv->ctrl_handler;
    common_data->i2c_client   = client;
    common_data->frmfmt       = ov5647_frmfmt;
    common_data->colorfmt     = camera_common_find_datafmt(
                      OV5647_DEFAULT_DATAFMT);
    common_data->power        = &priv->power;
    common_data->ctrls        = priv->ctrls;
  common_data->ident        = V4L2_IDENT_OV5647;
    common_data->priv         = (void *)priv;
    common_data->numctrls     = ARRAY_SIZE(ctrl_config_list);
    common_data->numfmts      = ARRAY_SIZE(ov5647_frmfmt);
    common_data->def_mode     = OV5647_DEFAULT_MODE;
    common_data->def_width    = OV5647_DEFAULT_WIDTH;
    common_data->def_height   = OV5647_DEFAULT_HEIGHT;
    common_data->fmt_width    = common_data->def_width;
    common_data->fmt_height   = common_data->def_height;
    common_data->def_clk_freq = OV5647_DEFAULT_CLK_FREQ;
    common_data->fmt_maxfps     = OV5647_DEFAULT_MAX_FPS;

    priv->i2c_client          = client;
    priv->s_data              = common_data;
    priv->subdev              = &common_data->subdev;
    priv->subdev->dev         = &client->dev;
    priv->s_data->dev         = &client->dev;

    err = ov5647_power_get(priv);
    if (err)
        return err;

    err = camera_common_parse_ports(client, common_data);
    if (err) {
        dev_err(&client->dev, "Failed to find port info\n");
        return err;
    }
    sprintf(debugfs_name, "ov5647_%c", common_data->csi_port + 'a');
    dev_info(&client->dev, "%s: name %s\n", __func__, debugfs_name);
    camera_common_create_debugfs(common_data, debugfs_name);

    v4l2_i2c_subdev_init(priv->subdev, client, &ov5647_subdev_ops);
  err = ov5647_reset_camera(priv);
    if (err)
```

```
        dev_err(&client->dev,
            "Failed to reset camera... continuing: %d\n", err);


    /* eeprom interface */
/*
    err = ov5647_eeprom_device_init(priv);
    if (err)
        dev_err(&client->dev,
            "Failed to allocate eeprom reg map: %d\n", err);
*/

    //err = ov5647_ctrls_init(priv, !err);
  dev_info(&client->dev, "%s: Initialize Controls\n", __func__);
    err = ov5647_ctrls_init(priv, false);
    if (err)
        return err;

    priv->subdev->internal_ops = &ov5647_subdev_internal_ops;
    priv->subdev->flags |= V4L2_SUBDEV_FL_HAS_DEVNODE |
                V4L2_SUBDEV_FL_HAS_EVENTS;

#if defined(CONFIG_MEDIA_CONTROLLER)
    dev_info(&client->dev, "%s: Initialize media control\n", __func__);
    priv->pad.flags = MEDIA_PAD_FL_SOURCE;
    priv->subdev->entity.type = MEDIA_ENT_T_V4L2_SUBDEV_SENSOR;
    priv->subdev->entity.ops = &ov5647_media_ops;
    err = media_entity_init(&priv->subdev->entity, 1, &priv->pad, 0);
    if (err < 0) {
        dev_err(&client->dev, "unable to init media entity\n");
        return err;
    }
#endif

    err = v4l2_async_register_subdev(priv->subdev);
    if (err)
        return err;

    dev_info(&client->dev, "Detected OV5647 sensor\n");
    return 0;
}

static int
ov5647_remove(struct i2c_client *client)
{
    struct camera_common_data *s_data = to_camera_common_data(client);
    struct ov5647 *priv = (struct ov5647 *)s_data->priv;

    v4l2_async_unregister_subdev(priv->subdev);
#if defined(CONFIG_MEDIA_CONTROLLER)
    media_entity_cleanup(&priv->subdev->entity);
#endif

    v4l2_ctrl_handler_free(&priv->ctrl_handler);
    ov5647_power_put(priv);
    camera_common_remove_debugfs(s_data);

    return 0;
}

static const struct i2c_device_id ov5647_id[] = {
    { "ov5647", 0 },
    { }
};

MODULE_DEVICE_TABLE(i2c, ov5647_id);

static struct i2c_driver ov5647_i2c_driver = {
    .driver = {
        .name = "ov5647",
```

```
        .owner = THIS_MODULE,
        .of_match_table = of_match_ptr(ov5647_of_match),
    },
    .probe = ov5647_probe,
    .remove = ov5647_remove,
    .id_table = ov5647_id,
};

module_i2c_driver(ov5647_i2c_driver);

MODULE_DESCRIPTION("SoC Camera driver for Sony OV5647");
MODULE_AUTHOR("David Wang <davidw@nvidia.com>");
MODULE_LICENSE("GPL v2");
```