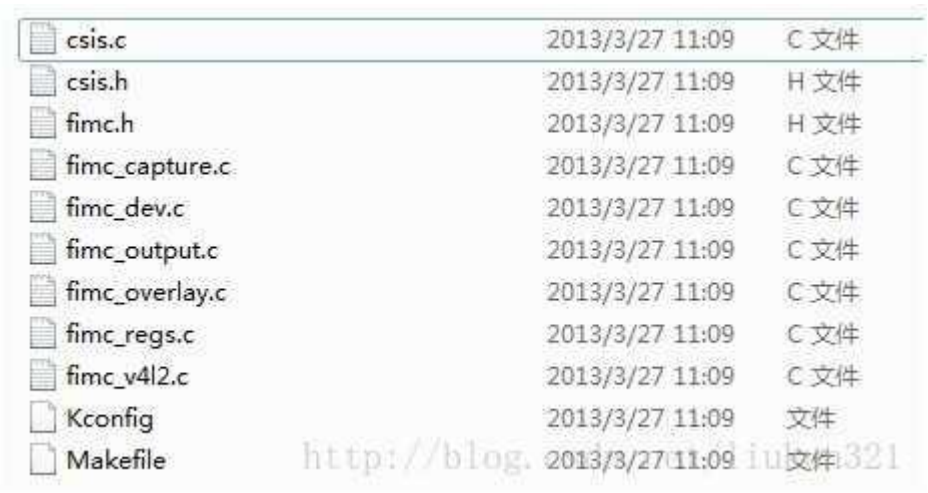
基于 Linux 3.0.8 Samsung FIMC（S5PV210） 的摄像头驱动框架解读

FIMC 这个名字应该是从 S5P100 开始出现的，在 s5pv210 里面的定义是摄像头接口，但是它同样具有图像数据颜色空间转换的作用。而 exynos4412 对它的定义看起来更清晰些，摄像头接口被定义为 FIMC-LITE 。颜色空间转换的硬件结构被定义为 FIMC-IS。不多说了，我们先来看看 Linux3.0.8 三星的 BSP 包中与 fimc 驱动相关的文件。



上面的源码文件组成了整个 fimc 的驱动框架。通过.c 文件的命名也大致可以猜测到 FIMC 的几个用途：

1、**Capture** ，Camera Interface 用于控制 Camera，及 m2m 操作

2、**Output**，这个用途可以简单看成：只使用了 FIMC 的 m2m 功能，这里 fimc 实际上就成了一个带有颜色空间转换功能的高速 DMA。

3、**Overlay**，比如 Android 的 Overlay 就依赖了 FIMC 的这个功能，可以简单把它看作是个 m2fb，当然实质上还是 m2m。

清楚 FIMC 的大致用途了。再来说说，每个 C 文件在 FIMC 驱动框架中扮演了何种角色：

csis.c 文件，用于 MIPI 接口的摄像头设备，这里不多说什么了。

fimc_dev.c 是驱动中对 FIMC 硬件设备最高层的抽象，这在后面会详细介绍。

fimc_v4l2.c linux 驱动中 ，将 fimc 设备的功能操作接口（**Capture，output，Overlay**），用 v4l2 框架封装。在应用层用过摄像头设备，或在应用层使用 FMIC 设备完成过 m2m 操作的朋友应该都清楚，fimc 经层层封装后最终暴露给用户空间的是 v4l2 标准接口的设备文件 videoX。 这里面也引出了一个我们应该关注的问题：Fimc 设备在软件层上是如何同摄像头设备关联的。

fimc_capture.c 实现对 camera Interface 的控制操作，它实现的基础依赖硬件相关的摄像头驱动（eg.ov965X.c / ov5642.c 等）。

并且提供以下函数接口，由 fimc v4l2.c 文件进一步封装:

```
int fimc_g_parm(struct file *file, void *fh, struct v4l2_streamparm    *a)
int fimc_s_parm(struct file *file, void *fh, struct v4l2_streamparm    *a)
int fimc_queryctrl(struct file *file, void    *fh, struct v4l2_queryctrl    *qc)
int fimc_querymenu(struct file *file, void    *fh, struct v4l2_querymenu    *qm)
int fimc_enum_input(struct file *file, void    *fh, struct v4l2_input    *inp)
int fimc_g_input(struct file *file, void    *fh, unsigned int    *i)
int fimc_release_subdev(struct fimc_control    *ctrl)
int fimc_s_input(struct file *file, void    *fh, unsigned int    i)
int fimc_enum_fmt_vid_capture(struct file *file, void *fh,struct v4l2_fmtdesc *f)
int fimc_g_fmt_vid_capture(struct file *file, void *fh, struct v4l2_format *f)
int fimc_s_fmt_vid_capture(struct file *file, void *fh, struct v4l2_format *f)
int fimc_try_fmt_vid_capture(struct file *file, void *fh, struct v4l2_format *f)
int fimc_reqbufs_capture(void *fh, struct v4l2_requestbuffers *b)
```

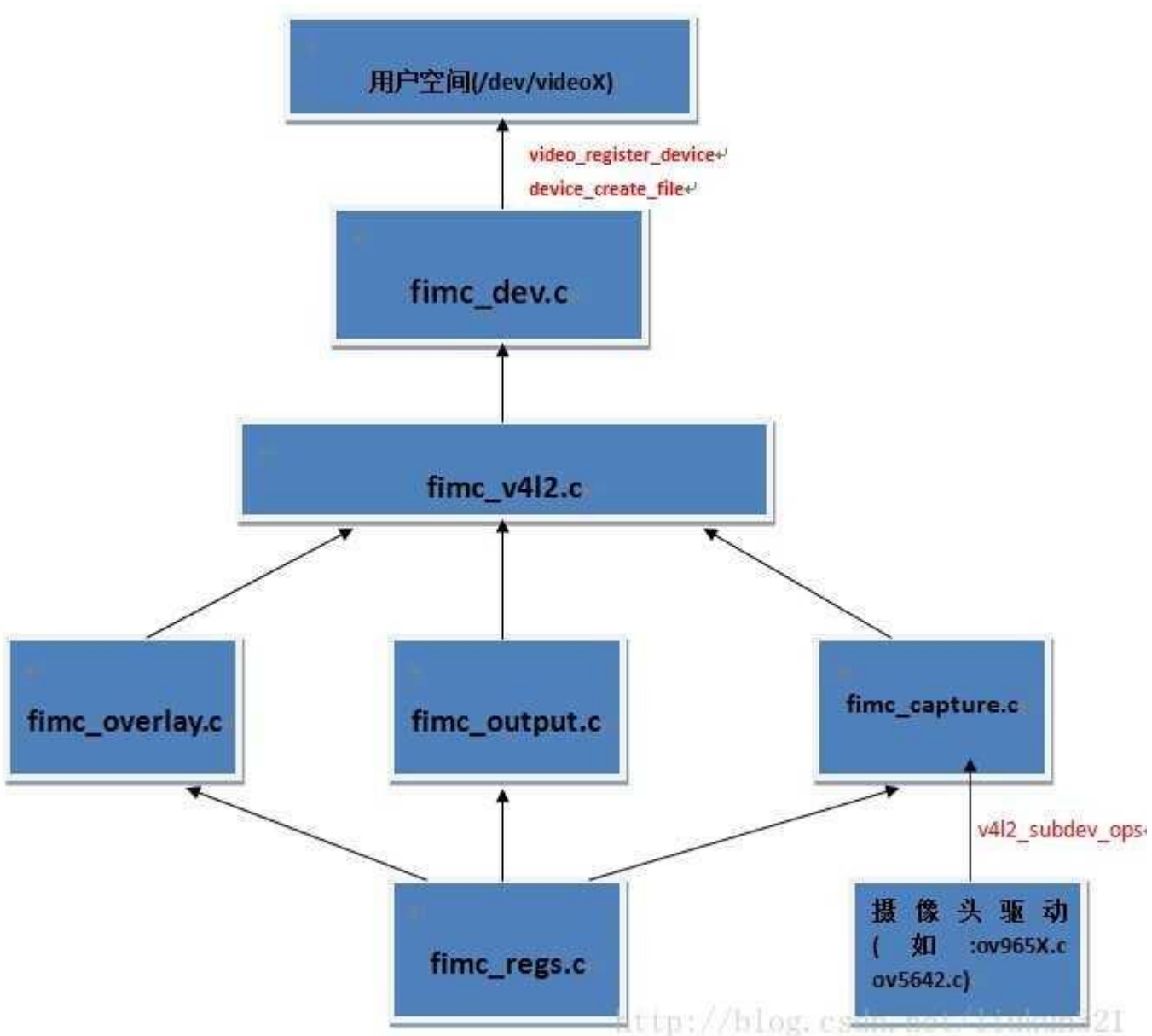
```
int fimc_querybuf_capture(void *fh, struct v4l2_buffer *b)
int fimc_g_ctrl_capture(void *fh, struct v4l2_control *c)
int fimc_s_ctrl_capture(void *fh, struct v4l2_control *c)
int fimc_s_ext_ctrls_capture(void *fh, struct v4l2_ext_controls *c)
int fimc_cropcap_capture(void *fh, struct v4l2_cropcap *a)
int fimc_g_crop_capture(void *fh, struct v4l2_crop *a)
int fimc_s_crop_capture(void *fh, struct v4l2_crop *a)
int fimc_start_capture(struct fimc_control *ctrl)
int fimc_stop_capture(struct fimc_control *ctrl)
int fimc_streamon_capture(void *fh)
int fimc_streamoff_capture(void *fh)
int fimc_qbuf_capture(void *fh, struct v4l2_buffer *b)
int fimc_dqbuf_capture(void *fh, struct v4l2_buffer *b)
```

fimc_output.c 实现 fimc m2m 操作，需要用 FIMC 实现硬件颜色空间转换的时候，这个文件里的函数就派上作用了，另外在 fimc 用于 Capture 和 overlay 过程本质上也包含 m2m 操作。因此除了提供功能函数接口，由 **fimc_v4l2.c** 文件进一步封装。另外还提供了一些功能函数供 **fimc_dev.c** 调用，比如用于设置一个 m2m 过程的 srcAddr（源地址） 和 dstAddr（目的地址）。这部分接口太多就不贴出来了。

fimc_overlay.c 实现 fimc overlay 操作。同样提供函数接口，由 **fimc_v4l2.c** 文件进一步封装。

fimc_regs.c Fimc 硬件相关操作，基本寄存器配置等。这个文件提供函数接口供 **fimc_capture.c、fimc_output.c、fimc_overlay.c** 调用。

通过刚才的分析，可以总结出下面的源码结构图：



好了，框架有了，再来看源码就轻松多了。

1、设备节点

接下来，先来看看 FIMC 设备的注册过程。以 FIMC-0 为例，说说/dev/video0 这个设备文件是怎么出来的。先看几个关键结构：

首先是 s3c_platform_fimc fimc_plat_lsi；也就是抽象 fimc 模块的数据结构，fimc_plat_lsi 还包含了一个.camera 成员。该结构初始化如下：

```
static struct s3c_platform_fimc  fimc_plat_lsi = {
        .srclk_name     = "mout_mpll",
        .clk_name       = "sclk_fimc",
        .lclk_name      = "fimc",
        .clk_rate       = 166750000,
#if defined(CONFIG_VIDEO_S5K4EA)
        .default_cam    = CAMERA_CSI_C,
#else
#ifdef CAM_ITU_CH_A
        .default_cam    = CAMERA_PAR_A,
#else
        .default_cam    = CAMERA_PAR_B,
```

```
#endif
#endif
        .camera         = {
#ifdef CONFIG_VIDEO_S5K4ECGX
                        &s5k4ecgx,
#endif
#ifdef CONFIG_VIDEO_S5KA3DFX
                        &s5ka3dfx,
#endif
#ifdef CONFIG_VIDEO_S5K4BA
                        &s5k4ba,
#endif
#ifdef CONFIG_VIDEO_S5K4EA
                        &s5k4ea,
#endif
#ifdef CONFIG_VIDEO_TVP5150
                        &tvp5150,
#endif
#ifdef CONFIG_VIDEO_OV9650
                        &ov9650,
#endif
        },
        .hw_ver         = 0x43,
};
```

可以看到在 s3c_platform_fimc 中有一个 camera 成员。这里重点看一下 ov9650.展开 ov9650

```
static struct s3c_platform_camera ov9650 = {
        #ifdef CAM_ITU_CH_A
        .id             = CAMERA_PAR_A,
        #else
        .id             = CAMERA_PAR_B,
        #endif
        .type           = CAM_TYPE_ITU,
        .fmt            = ITU_601_YCBCR422_8BIT,
        .order422       = CAM_ORDER422_8BIT_YCBYCR,
        .i2c_busnum     = 0,
        .info           = &ov9650_i2c_info,
        .pixelformat    = V4L2_PIX_FMT_YUYV,
        .srclk_name     = "mout_mpll",
        /* .srclk_name  = "xusbxti", */
        .clk_name       = "sclk_cam1",
        .clk_rate       = 40000000,
        .line_length    = 1920,
        .width          = 1280,
        .height         = 1024,
        .window         = {
                .left   = 0,
```

```
                .top    = 0,
                .width  = 1280,
                .height = 1024,
        },

        /* Polarity */
        .inv_pclk       = 1,
        .inv_vsync      = 1,
        .inv_href       = 0,
        .inv_hsync      = 0,

        .initialized    = 0,
        .cam_power      = ov9650_power_en,
};
```

这个结构体，实现了对 ov9650 摄像头硬件结构的抽象。定义了摄像头的关键参数和基本特性。

因为 fimc 设备在 linux3.0.8 内核中作为一个平台设备加载，而上面提到的 s3c_platform_fimc　fimc_plat_lsi 仅是 fimc 的抽象数据而非设备。这就需要将抽象 fimc 的结构体作为 fimc　platform_device 的一个私有数据。所以就有了下面的过程。s3c_platform_fimc　fimc_plat_lsi 结构在板级设备初始化 XXX_machine_init(void) 过程作为 s3c_fimc0_set_platdata 的实参传入。之后 fimc_plat_lsi 就成为了 fimc 设备的 platform_data 。

```
s3c_fimc0_set_platdata(&fimc_plat_lsi);
s3c_fimc1_set_platdata(&fimc_plat_lsi);
s3c_fimc2_set_platdata(&fimc_plat_lsi);
```

以 s3c_fimc0_set_platdata 为例展开

```
void __init s3c_fimc0_set_platdata(struct s3c_platform_fimc *pd)
{
        struct s3c_platform_fimc *npd;

        if (!pd)
                pd = &default_fimc0_data;

        npd = kmemdup(pd, sizeof(struct s3c_platform_fimc), GFP_KERNEL);
        if (!npd)
                printk(KERN_ERR "%s: no memory for platform data\n", __func__);
        else {
                if (!npd->cfg_gpio)
                        npd->cfg_gpio = s3c_fimc0_cfg_gpio;

                if (!npd->clk_on)
                        npd->clk_on = s3c_fimc_clk_on;

                if (!npd->clk_off)
                        npd->clk_off = s3c_fimc_clk_off;

                npd->hw_ver = 0x45;
```

```
                /* starting physical address of memory region */
                npd->pmem_start = s5p_get_media_memory_bank(S5P_MDEV_FIMC0, 1);
                /* size of memory region */
                npd->pmem_size = s5p_get_media_memsize_bank(S5P_MDEV_FIMC0, 1);

                s3c_device_fimc0.dev.platform_data = npd;
        }
}
```

最后一句是关键 s3c_device_fimc0.dev.platform_data = npd; 看一下 s3c_device_fimc0 定义：

```
struct platform_device s3c_device_fimc0 = {
        .name           = "s3c-fimc",
        .id             = 0,
        .num_resources  = ARRAY_SIZE(s3c_fimc0_resource),
        .resource       = s3c_fimc0_resource,
};
```

而 fimc 的抽象数据，则作为它的私有数据被包含进了 s3c_device_fimc0 这个结构中。到这里才完成了 FIMC 平台设备的最终定义。而这个平台设备的定义 s3c_device_fimc0 又被添加到了整个硬件平台的 platform_device 列表中，最终在 XXX_machine_init(void) 函数中调用 platform_add_devices(mini210_devices, ARRAY_SIZE(mini210_devices)); 完成所有 platform_device 的注册：

```
static struct platform_device *mini210_devices[] __initdata = {
        &s3c_device_adc,
        &s3c_device_cfcon,
        &s3c_device_nand,
        。。。
        &s3c_device_fb,
        &mini210_lcd_dev,
#ifdef CONFIG_VIDEO_FIMC
        &s3c_device_fimc0,
        &s3c_device_fimc1,
        &s3c_device_fimc2,
}
```

```
platform_add_devices(mini210_devices, ARRAY_SIZE(mini210_devices));
```

platform_device 被加载后，等待与之匹配的 platform_driver 。若此时 fimc driver 的驱动模块被加载。这个时候， fimc_dev.c 文件里的 static int __devinit fimc_probe(structplatform_device *pdev) 函数上场了。

```
static int __devinit fimc_probe(struct platform_device *pdev)
{
        struct s3c_platform_fimc *pdata;
        struct fimc_control *ctrl;
        struct clk *srclk;
        int ret;
        if (!fimc_dev) {
```

```
        fimc_dev = kzalloc(sizeof(*fimc_dev), GFP_KERNEL);
        if (!fimc_dev) {
                dev_err(&pdev->dev, "%s: not enough memory\n", __func__);
                return -ENOMEM;
        }
}

ctrl = fimc_register_controller(pdev);
if (!ctrl) {
        printk(KERN_ERR "%s: cannot register fimc\n", __func__);
        goto err_alloc;
}

pdata = to_fimc_plat(&pdev->dev);
if (pdata->cfg_gpio)
        pdata->cfg_gpio(pdev);

#ifdef REGULATOR_FIMC
        /* Get fimc power domain regulator */
        ctrl->regulator = regulator_get(&pdev->dev, "pd");
        if (IS_ERR(ctrl->regulator)) {
                fimc_err("%s: failed to get resource %s\n", __func__, "s3c-fimc");
                return PTR_ERR(ctrl->regulator);
        }
#endif //REGULATOR_FIMC
        /* fimc source clock */
        srclk = clk_get(&pdev->dev, pdata->srclk_name);
        if (IS_ERR(srclk)) {
                fimc_err("%s: failed to get source clock of fimc\n", __func__);
                goto err_v4l2;
        }

        /* fimc clock */
        ctrl->clk = clk_get(&pdev->dev, pdata->clk_name);
        if (IS_ERR(ctrl->clk)) {
                fimc_err("%s: failed to get fimc clock source\n", __func__);
                goto err_v4l2;
        }

        /* set parent for mclk */
        clk_set_parent(ctrl->clk, srclk);

        /* set rate for mclk */
        clk_set_rate(ctrl->clk, pdata->clk_rate);

        /* V4L2 device-subdev registration */
        ret = v4l2_device_register(&pdev->dev, &ctrl->v4l2_dev);
        if (ret) {
```

```
        fimc_err("%s: v4l2 device register failed\n", __func__);
        goto err_fimc;
    }

    /* things to initialize once */
    if (!fimc_dev->initialized) {
        ret = fimc_init_global(pdev);
        if (ret)
            goto err_v4l2;
    }

    /* video device register */
    ret = video_register_device(ctrl->vd, VFL_TYPE_GRABBER, ctrl->id);
    if (ret) {
        fimc_err("%s: cannot register video driver\n", __func__);
        goto err_v4l2;
        }

        video_set_drvdata(ctrl->vd, ctrl);

        ret = device_create_file(&(pdev->dev), &dev_attr_log_level);
        if (ret < 0) {
            fimc_err("failed to add sysfs entries\n");
            goto err_global;
        }
        printk(KERN_INFO "FIMC%d registered successfully\n", ctrl->id);

        return 0;

err_global:
        video_unregister_device(ctrl->vd);

err_v4l2:
        v4l2_device_unregister(&ctrl->v4l2_dev);

err_fimc:
        fimc_unregister_controller(pdev);

err_alloc:
        kfree(fimc_dev);
        return -EINVAL;

}
```

在 fimc_probe 函数中有这么一段

```
if(!fimc_dev->initialized) {
            ret = fimc_init_global(pdev);
```

```
                if (ret)
                        goto err_v4l2;
        }
```

这段代码执行过程：首先判断 fimc 是否已经被初始化完成（此时 FIMC 是忙状态的），如果没有被初始化，则执行 fimc_init_global(pdev);函数，它的作用是先判断平台数据中是否初始化了摄像头结构（即前面提到的.camera 成员），从平台数据中获得摄像头的时钟频率并将平台数据中内嵌的 s3c_platform_camera 结构数据保存到该驱动模块全局的 fimc_dev 中，感兴趣的朋友可以展开这个函数看一下，这里就不再贴出来了。

紧接着这段代码还执行了两个非常关键的过程：

```
ret= v4l2_device_register(&pdev->dev, &ctrl->v4l2_dev);
        if (ret) {
                fimc_err("%s: v4l2device register failed\n", __func__);
                goto err_fimc;
        }
```

这个函数里的核心完成了对 v4l2_dev->subdev 链表头的初始化，并将 ctrl->v4l2_dev 关联到 pdev->dev 结构的私有数据的 driver_data 成员中（即完成了 pdev->dev->p->driver_data= ctrl->v4l2_dev; ），也就是实现了 v4l2_dev 向内核结构注册的过程。

```
ret= video_register_device(ctrl->vd, VFL_TYPE_GRABBER, ctrl->id);
        if (ret) {
                fimc_err("%s: cannotregister video driver\n", __func__);
                goto err_v4l2;
        }

        video_set_drvdata(ctrl->vd, ctrl);

        ret = device_create_file(&(pdev->dev),&dev_attr_log_level);
```

上面的过程完成了对 video_device 设备的注册，并且在 sys 目录下生成了对应的属性文件。如果系统中移植有 mdev，将会生成对应设备节点/dev/videoX。

其实到目前为止，只完成了 fimc 设备主要数据结构的初始化和注册，几乎没有操作 fimc 或摄像头的硬件寄存器。也没有完成 FIMC 驱动和摄像头的驱动模块的软件关联。我们是如何做到仅操作 fimc 的设备节点/dev/videoX 就能控制摄像头设备的效果呢？下回分解吧。。。