1. V4L2框架概述

2. video_device

3. videobuf2

4. Subdev

5. media framework

6. v4l2设备节点组织流程

6.1设备树分析

6.2 驱动注册流程分析

6.3 SOC中CSI驱动异步注册流程

6.4 Video 驱动注册节点流程

1.V4L2框架概述

V4L2是Video for linux2的简称,为linux中关于视频设备的内核驱动。在Linux中，视频设备是设备文件，可以像访问普通文件一样对其进行读写。

V4L2在设计之初时，是要支持很多广泛的设备的,如声卡, display, FB, I2C, camera等.它们之中只有一部分在本质上是真正的视频设备,也是造成V4l2源码冗余的原因之一。

kernel更新速度快,在display上有drm 框架　和framebuffer框架, 声卡上有ALSA框架. 目前V4L2主要用于camera驱动,本文也是通过camera驱动讲解V4l2内部原理。

1.1 v4l2设备应用层流程

注册的设备节点有/dev/video和/dev/v4l2-subdev。

应用层操作video设备主要流程如下:

1. 通过打开video设备设置video参数。

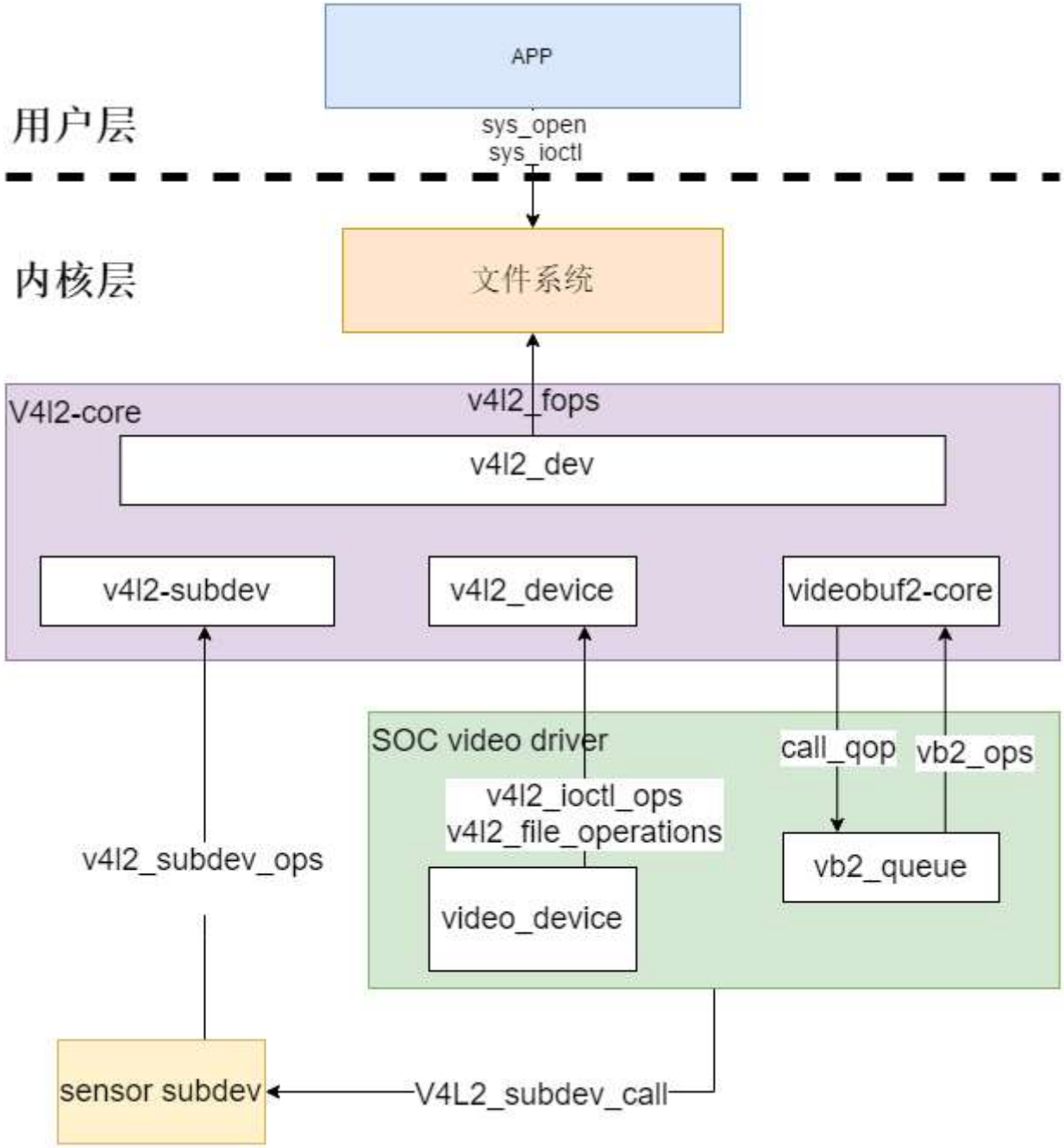
2. 设置采集方式。

3. 将数据取出,处理,放回, 可循环处理。

4. 完成相应的任务后关闭。

1.2 内核V4L2模块

应用层流程之所以简单, 是因为内核相关模块做了很多工作夯实了基础。 与V4L2相关的摸如下:



① video_device

用于实例化一个/dev/video设备的结构体。里面包含该video的类型, 回调函数,以及操作缓冲的队列。 接触内核v4l2驱动, 理解video_device结构体内部很重要。

② v4l2_subdev

用于实例化一个/dev/subdev 设备的结构体。 一般只需通过ioctl设置采样属性即可。 内部实现部分v4l2_subdev_ops回调函数, 也可以用与其他驱动模块通讯.

③ videobuf2

用于video缓存的分配,释放,出队,入队等。提供多种缓存类型管理。

2.video_device

video_device 即注册 /dev/video# 设备的结构体, 应用层需要使用camera的时候open 与ioctl它。 下面分析video_device的数据结构, 以及内核注册流程。

video_device 结构体定义位于v4l2_dev.h, 注册流程位于对应的soc 驱动下, 一般位于driver/media/platform/[soc 对应的图像处理模块]。

2.1 图像处理模块

图像处理模块在不同芯片厂商中而不同, 如:i.mx6q 为IPU, i.mx8q为ISI, renesas rcar-m3为VIN。主要作用是接收将CSI 或者 parallel digital信号,并进行图像数据进行格式转换,缩放,色域转换,DMA通道管理等, 然后映射到相应的内存上, 给用户使用。

2.2 video注册流程

当总线检测到图像处理设备后,便会进入该设备driver中的probe函数, 在probe函数调用video_register_device注册video_device, 注册前需要先初始化video_device结构体内部成员。

2.2.1 video_device 结构体成员介绍:

① video_device->v4l2_dev: 必须的设置成V4l2_device父设备,该父设备通过v4l2_device_register函数初始化

② video_device->name: 设置一个唯一设备名描述符

③ video_device->fops: 设置成v4l2_file_operations 结构体。 图像处理video驱动一般只实现open release 成员函数,其他交给V4l2-core 实现。

```
static const struct v4l2_file_operations rvin_fops = {

        .owner = THIS_MODULE,

        .unlocked_ioctl = video_ioctl2,

        .open = rvin_open, // soc 驱动实现

        .release = rvin_release, // soc 驱动实现

        .poll = vb2_fop_poll, //videobuf2内部实现

        .mmap = vb2_fop_mmap, //videobuf2内部实现

        .read = vb2_fop_read, //videobuf2内部实现

};
```

④ video_device->ioctl_ops: 设置成v4l2_ioctl_ops结构, 该结构体成员为应用层操作video_device_fd句柄的ioctl(VIDEOC_XXX) ,所回调的函数指针。 我们配置/dev/video主要通过使用ioctl(VIDEOC_XXX), 命令宏如:VIDIOC_S_FMT, VIDIOC_G_FMT, VIDIOC_REQBUFS, VIDIOC_QUERYBUF, VIDIOC_DQBUF, VIDIOC_STREAMOFF。

对应v4l2_ioctl_ops内部的函数指针成员为:

```
vidioc_s_fmt_vid_cap 为VIDIOC_S_FMT 的ioctl逻辑中所指向的函数指针。
vidioc_g_fmt_vid_cap 为VIDIOC_G_FMT 的ioctl逻辑中所指向的函数指针。
vidioc_reqbufs       为VIDIOC_REQBUFS 的ioctl逻辑中所指向的函数指针。
vidioc_querybuf      为VIDIOC_QUERYBUF 的ioctl逻辑中所指向的函数指针。
```

```
vidioc_querybuf       为VIDIOC_DQBUF 的ioctl逻辑中所指向的函数指针。
vidioc_streamoff      为VIDIOC_STREAMOFF 的ioctl逻辑中所指向的函数指针。
```

soc图像处理驱动必须实现v4l2_ioctl_ops对应的成员函数。 应用层在ioctl时,才不会出错。

⑤ video_device->queue: 指向该video设备节点的vb2_queue指针。 很关键的一个成员,与缓冲数据操作有关。 详细分析在下一节。

3.videobuf2

videobuf2 是v4l2驱动一个重要组成部分, 用来管理和存储video帧缓存。

3.1 与video device的关系:

① video_device结构体内部有vb2_queue结构体成员指针。

② 在应用层操作/dev/video# 句柄的函数read, poll, mmap(),ioctl等,即与帧缓存相关的函数。涉及到videobuf2。

③ video_device结构体内部有 v4l2_ioctl_ops结构体, v4l2_ioctl_ops内部函数成员的实现在V4l2-core中的videobuf2-v4l2.c 。 在videobuf2-core.c中又会回调平台图像处理器驱动实现vb2_ops。

User Function name	Description	v4l2_ioctl_ops 's callback in video_device struct	Implement in videobuf2-v4l2.c
ioctl(VIDIOC_REQBUFS)	缓冲需求与发送模式选择	vidioc_reqbufs	vb2_ioctl_reqbufs
ioctl(VIDIOC_DQBUF)	在队列中弹出一个空缓冲	vidioc_dqbuf	vb2_ioctl_dqbuf
ioctl(VIDIOC_QBUF)	在队列中压入一个填入数据的缓冲	vidioc_qbuf	vb2_ioctl_qbuf

3.2 buffer类型

3.2.1 vmalloc() buffers

使用虚拟连续内存. 完全不需要考虑内存分配问题. videobuf2内部完成所有细节。 但是性能最低。

3.2.2 Physically contiguous dma

物理地址连续内存传输,传输完一个block物理地址连续内存后,发生中断。 大多数capture中使用,使用较为简单,性能中等。

3.2.3 Physically scattered

一种特殊类型的流DMA映射机制–发散/汇聚映射。该机制允许一次为多个缓冲区创建DMA映射,它是用一个链表描述物理不连续的存储器，然后把链表首地址告诉dma master。dma master传输完一块物理连续的数据后，就不用再发中断了，而是根据链表传输下一块物理连续的数据。性能最高,实现起来复杂。

3.2.4 通过调用的以下那个头文件, 即可查出soc V4I2使用的内存类型..

/* Physically scattered */

/* vmalloc() buffers */

/* Physically contiguous */

3.3 vb2_ops回调函数

提供 videobuf2-core 回调的函数的结构体体vb2_ops

3.3.1 vb2_ops 结构体

上文提到v4l2-core的videobuf2-core.c中,需要回调图像处理驱动中的实现的函数。 这些函数在vb2_ops结构体中, 如下图所示:

```
static const struct vb2_ops rvin_qops = {

        .queue_setup = vin_queue_setup,
```

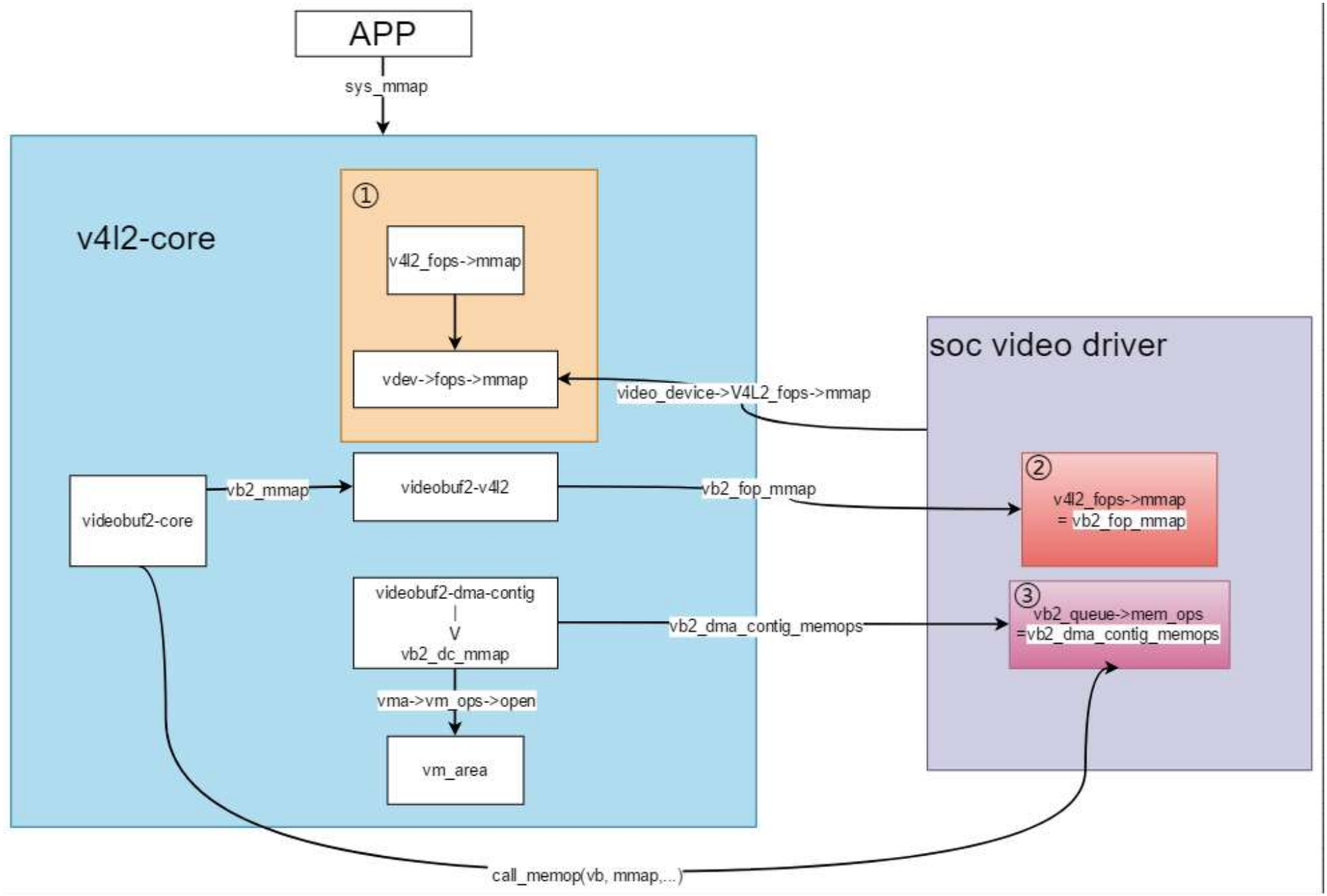
```
        .buf_prepare = vin_buffer_prepare,
    .buf_queue = vin_buffer_queue,
    .start_streaming = vin_start_streaming,
    .stop_streaming =vin_stop_streaming,
    .wait_prepare = vin_ops_wait_prepare,
    .wait_finish = vin_ops_wait_finish,
};
```

3.3.2 videobuf2-core回调vb2_ops函数指针宏:

```
call_qop(q, queue_setup....);
call_vb_qop(vb, buf_prepare, vb);
call_void_vb_qop(vb, buf_queue, vb);
call_qop(q, start_streaming, ...);
call_void_qop(q, stop_streaming, q);
call_void_qop(q, wait_prepare, q);
call_void_qop(q, wait_finish, q);
```

3.4 mmap 流程

下面通过分析 应用层mmap 函数的在kernel中流程,buffer类型为Physically contiguous dma. 进一步理解videobuf2在V4L2中的逻辑。



- 应用层调用mmap 后,系统调用sys_mmap。 v4l2-core调用vdev->fops->mmap. vdev 是由soc 平台驱动注册。
- 在soc驱动中, 将V4l2-core 中的videobuf2-v4l2中的vb2_fop_mmap赋值给 V4l2_fops->mmap。 vb2_fop_mmap调用了videobuf2-core中的vb2_mmap

- vb2_mmap 又会调用call_memop(vb,mmap,...)。 call_memop中有的需要在soc驱动中的初始化vb2_queue->mem_ops为vb2_dma_contig_memops。 vb2_dma_contig_memops中的vb2_dc_mmap即为最终回调。 之后操作vm_area

小结: 上面的mmap流程分析, 可谓三进三出。 v4l2-core需要调用soc驱动注册的video device以及回调其内部实现的函数, 而soc 驱动初始化时有需要v4l2-core定义好的好的结构体和成员函数。 随着内部成员的增多,功能多样化。这要来来回回的调用必定增多,所以分析起来也很复杂。

4.Subdev

4.1 概念

在camera驱动中,subdev一般指与video device 相关的外围senor子设备, 可以是csi接口,fpdlink,camera内部isp等。 大部分为I2c_subdev。 subdev可用于应用层的调用,以及驱动间的交互。 video device 可组织和控制V4l2_subdev。

4.2 subdev注册流程

4.2.1 v4l2_subdev结构体

每个subdev驱动必须实现一个v4l2_subdev结构体, 这个结构体可以单独存在,或者嵌入到其他更大的结构体中,如果该subdev需要储存更多信息。

如果集成了media framework, 必须通过media_entity_pads_init()初始化media_entity,作为media framework 构建单元。 而且subdev驱动还需实现v4l2_subdev_video_ops的v4l2_subdev_pad_ops。 用于 ioctl 的handler,或者其他设备驱动回调。 部分回调函数对应如下:

v4l2_subdev_video_ops中的成员	User ioctl 命令	其他驱动回调
g_std	VIDIOC_G_STD	v4l2_subdev_call(sd, video, g_std, a);
s_std	VIDIOC_S_STD	v4l2_subdev_call(sd, video, s_std, a);

v4l2_subdev_pad_ops中的成员	User ioctl 命令	其他驱动回调

get_fmt	VIDIOC_SUBDEV_G_FMT	v4l2_subdev_call(sd, pad, get_fmt...)
set_fmt	VIDIOC_SUBDEV_S_FMT	v4l2_subdev_call(sd, pad, set_fmt...)

v4l2_subdev_video_ops与 video device 中v4l2_ioctl_ops结构体内部成员有很多看上去相似的地方。 确实在v4l2 头文件 中, 许多V4l2_xxx_ops结构体成员看上去类似, 再加上这些ops 要被其他driver或者user 的调用, 如果是新手, 很容易混淆。

那么如何理解V4l2_xxx_ops中的逻辑?

① 得区分它是那个subdev的 ops。 可以通过打印subdev指针区分。

② 分析这个ops的功能。 是set还是get? ops是否对sensor的寄存器更改了? 还是没有进行任何操作。

③ 然后在分析它的调用逻辑。 执行完之后, 成功和错误分别会执行些什么?

④ 最好做个笔记, 描述某些特殊情况, 例如某些图像处理器无法处理某些特定格式。

4.2.2 v4l2_subdev注册函数调用

① 给包含v4l2_subdev的数据结构kzalloc()一片内存出来

② 调用v4l2_subdev_init函数初始化V4l2_subdev结构体成员

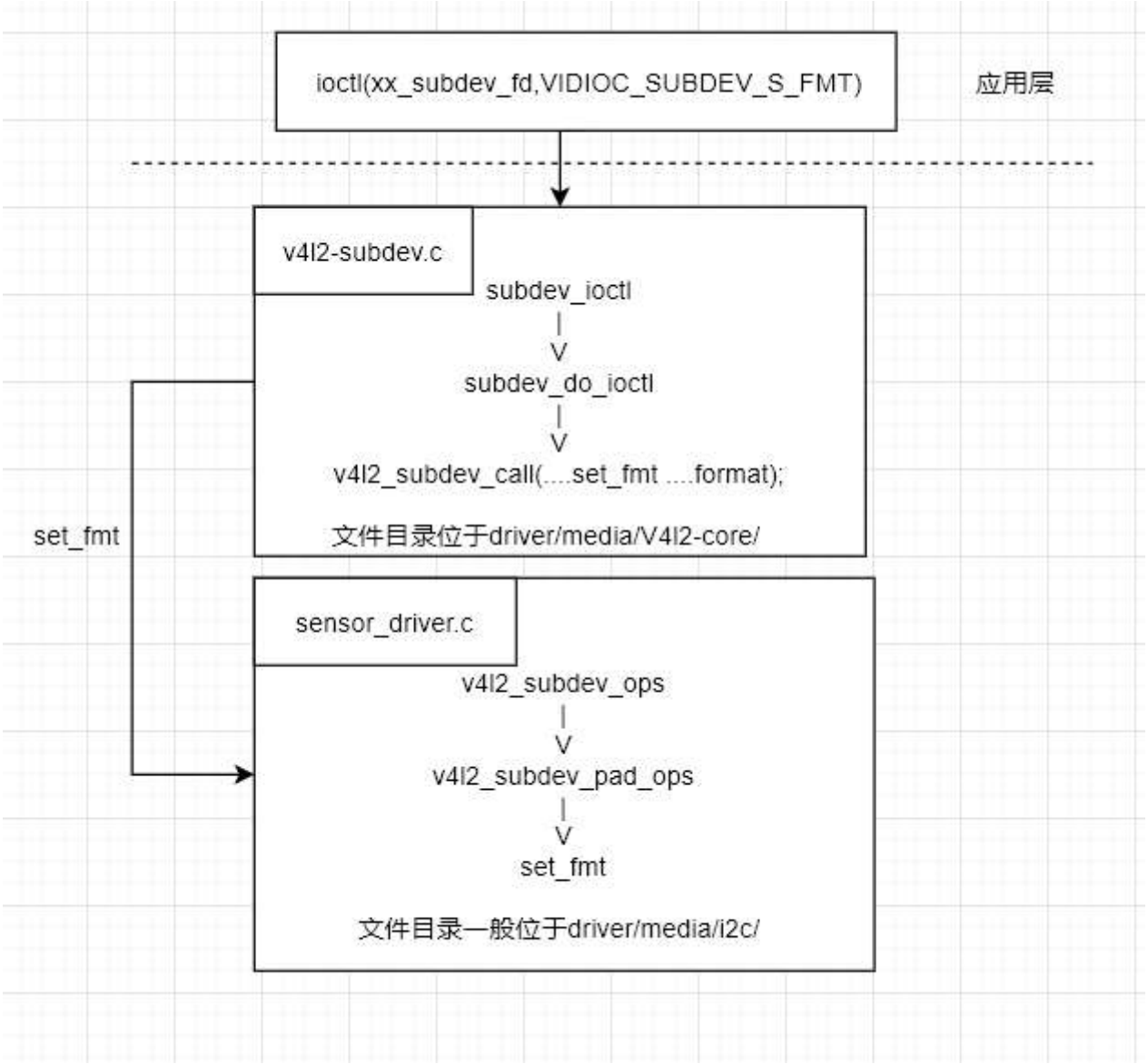
③ v4l2_async_register_subdev 异步注册, 没有真正的注册. 缓存到某个变量中

⑤ 主设备调用组织subdev后,调用v4l2_device_register_subdev_nodes函数真正注册subdev节点

那么主设备是如何组织subdev的呢? 这事就需要 media framework, V4L2 asynchronous 模块, V4L2 fwnode绑定分析模块辅助,以及设备树。 逻辑复杂,第6节v4l2设备节点组织流程中有分析。

4.3 应用层操作subdev

应用层一般通过ioctl(VIDIOC_SUBDEV_XXXX), 操作subdev. 对应流程如下:



小结: V4l2_subdev是video设备的一个子设备, 在操作video时,一般需要先配置它,或者获取它的参数。 subdev驱动如果在probe初始化或者定义回调函数时有问题,导致主设备video 也无法正常工作。

5.media framework

5.1 概念

Media framework的目标之一构建媒体设备内部的拓扑, 并且在运行时配置该设备。 V4L2为了实现它, 将与V4L2 相关的设备抽象化成一个个media entity, 然后通过pad 将 entity 连接(link) 起来, 组成一个media pipeline。 然后使pipeline可控化。

① Entity

是一个多媒体硬件设备的基础构块。它可以虚拟的设备为:cmos senor外围设备, soc 图像处理模块, dma通道,或者硬件接口(CSI, parallel)。

② Pad

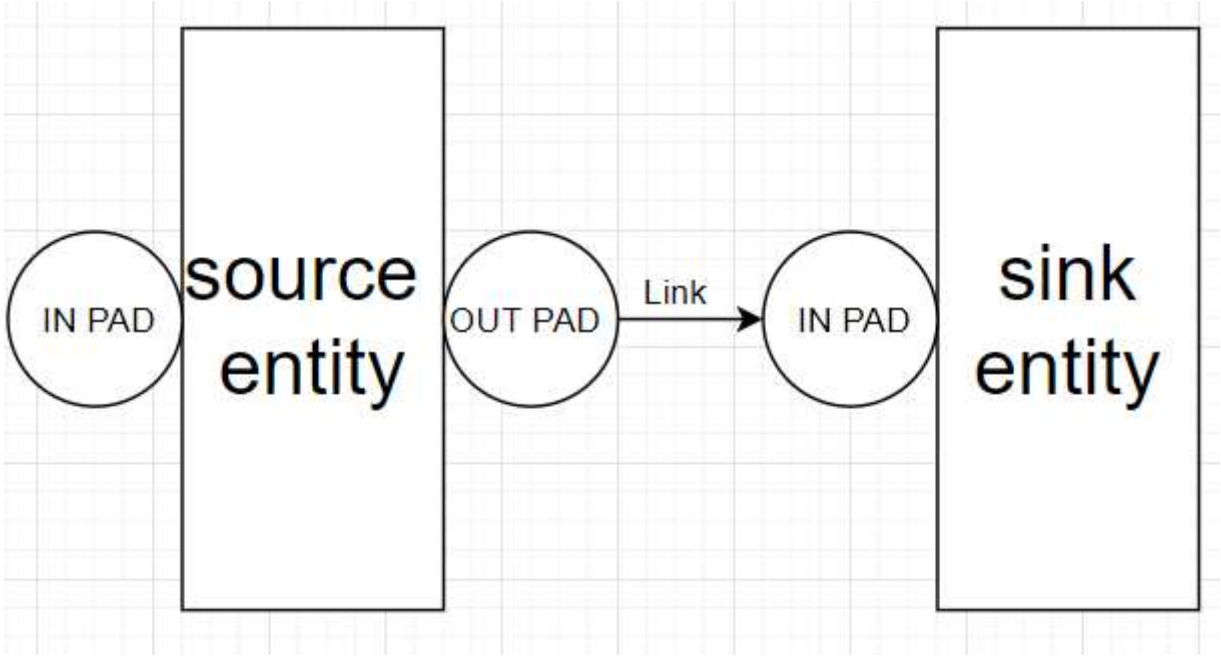
是一个可以使一个或者多个entity交互的可连接端点,单独存在没有意义,可以理解为一个硬件设备对外的接口。

③ Link

是两个pads的有向连接,并且是点对点的

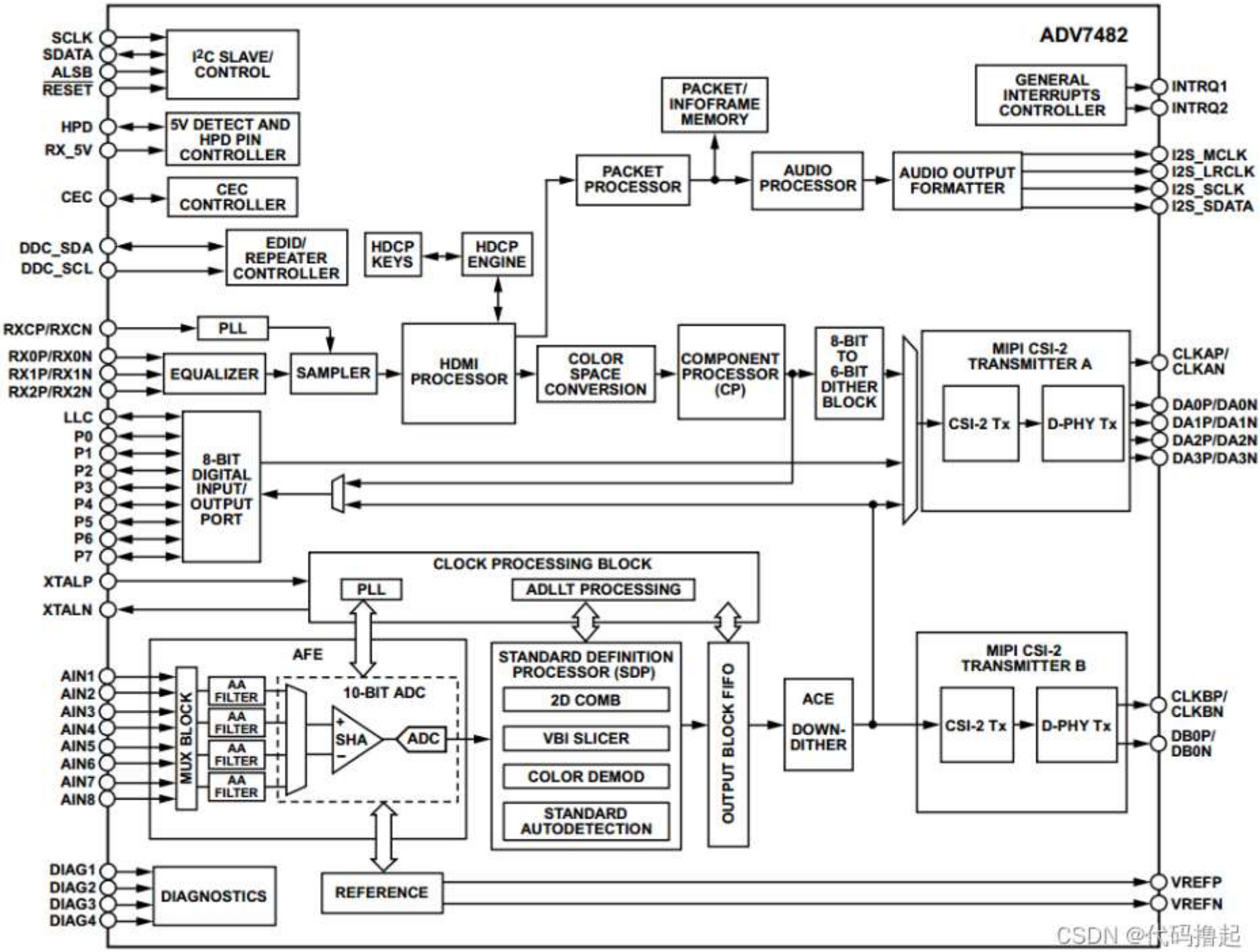
④ pipeline

由entity pad link 组成, 一个entity可以有多个pad, pad 可以link其他entity的pad,所以pipeline形式可以多种多样的,最基础的如下:



这样说还是有点抽象,下面以adv7482作为例子来说明。

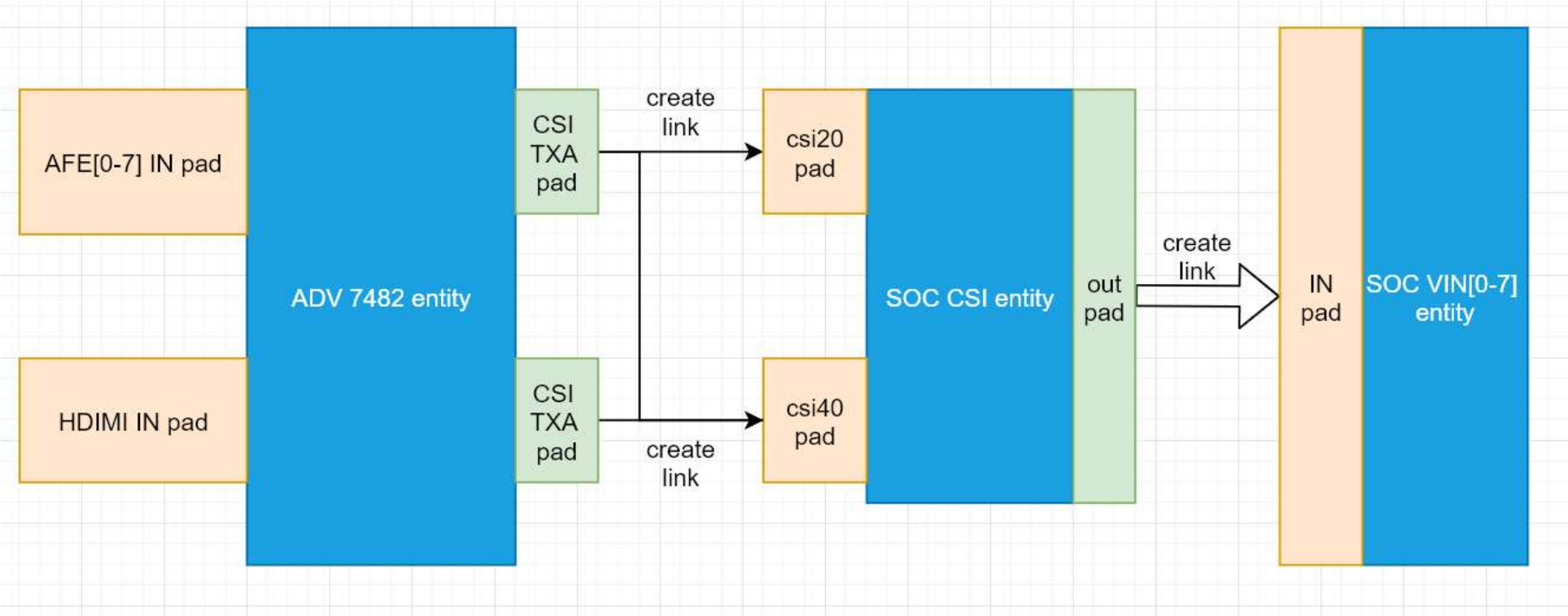
adv7482功能: 可以接受八路CVBS模拟和一路HDMI输入,然后转换成一路或者两路的CSI信号输出到SOC上。 硬件框图如下:


ADV7482
SCLK
SDATA
ALSB
RESET
I²C SLAVE/ CONTROL
HPD
RX_5V
5V DETECT AND HPD PIN CONTROLLER
CEC
CEC CONTROLLER
DDC_SDA
DDC_SCL
EDID/ REPEATER CONTROLLER
HDCP KEYS
HDCP ENGINE
PACKET/ INFOFRAME MEMORY
PACKET PROCESSOR
AUDIO PROCESSOR
AUDIO OUTPUT FORMATTER
GENERAL INTERRUPTS CONTROLLER
INTRQ1
INTRQ2
I2S_MCLK
I2S_LRCLK
I2S_SCLK
I2S_SDATA
RXCP/RXCN
PLL
RX0P/RX0N
RX1P/RX1N
RX2P/RX2N
EQUALIZER
SAMPLER
HDMI PROCESSOR
COLOR SPACE CONVERSION
COMPONENT PROCESSOR (CP)
8-BIT TO 6-BIT DITHER BLOCK
MIPI CSI-2 TRANSMITTER A
CSI-2 Tx
D-PHY Tx
CLKAP/ CLKAN
DA0P/DA0N
DA1P/DA1N
DA2P/DA2N
DA3P/DA3N
LLC
P0
P1
P2
P3
P4
P5
P6
P7
8-BIT DIGITAL INPUT/ OUTPUT PORT
CLOCK PROCESSING BLOCK
XTALP
XTALN
PLL
ADLLT PROCESSING
AFE
AIN1
AIN2
AIN3
AIN4
AIN5
AIN6
AIN7
AIN8
MUX BLOCK
AA FILTER
AA FILTER
AA FILTER
AA FILTER
10-BIT ADC
SHA
ADC
STANDARD DEFINITION PROCESSOR (SDP)
2D COMB
VBI SLICER
COLOR DEMOD
STANDARD AUTODETECTION
OUTPUT BLOCK FIFO
ACE
DOWN-DITHER
MIPI CSI-2 TRANSMITTER B
CSI-2 Tx
D-PHY Tx
CLKBP/ CLKBN
DB0P/ DB0N
DIAG1
DIAG2
DIAG3
DIAG4
DIAGNOSTICS
REFERENCE
VREFP
VREFN
CSDN @代码撸起

简单介绍以下内部的模块, 左侧有输入接口如: AIN1-AIN8为模拟输入接口, RX0~RX2为hdmi差分输入接口. 右侧有CSI-2 transmiter A, CSI-2 transmiter B输出接口. 内部模块负责信号检测,模数转换, 始终管理, 色域转换等.

这种sensor controller 可以抽象为多个subdev 负责不同模块之间的处理, 再与SOC 内部模块交互,组成可用的一个可用video节点. 但是单纯控制subdev没有数据流的概念,很难分析逻辑层次和拓扑关系, 这时就需要media framework 抽象化这些设备的拓扑,组成一个media graph.

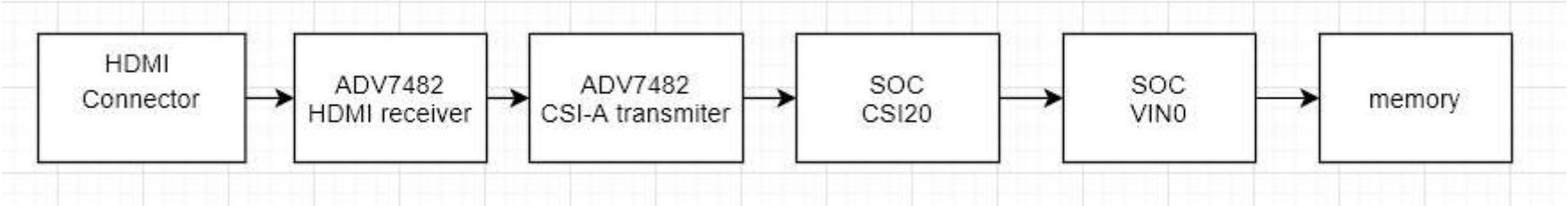
adv7482与renesas rcar-m3 SOC组成的media graph, 如下图所示:



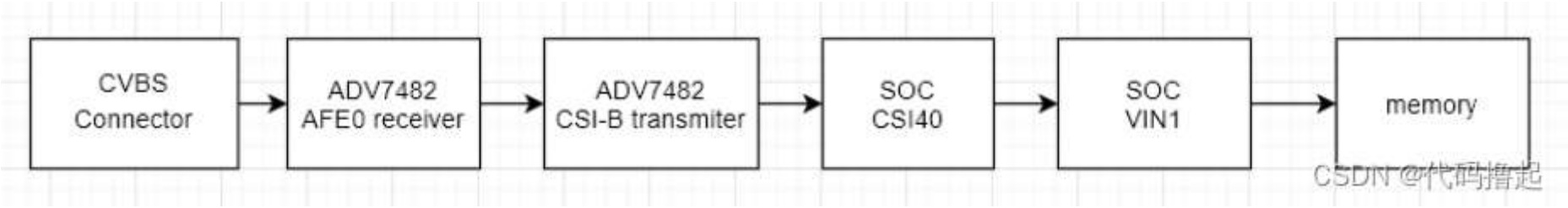
media pipeline 是在media graph 选出来的一个通道, 可以这么理解media graph是一张地图, pipeline就是其中的一条规划好的路线.

在驱动实现中,一个由三个media device: adv7482 sensor , SOC中的CSI, SOC 图像处理模块VIN[0-7]. 组成的pipeline如下:

① pipeline 1



② pipeline 2



当然pad 与entity 的属性还可以设置其他不同的值,这样link其他可以组成非常多的不同形式pipeline, 如:

①ADV7186可以选择8个camera模拟信号输入通道

②SOC中的CSI模块到VIN模块有4个DAM channel

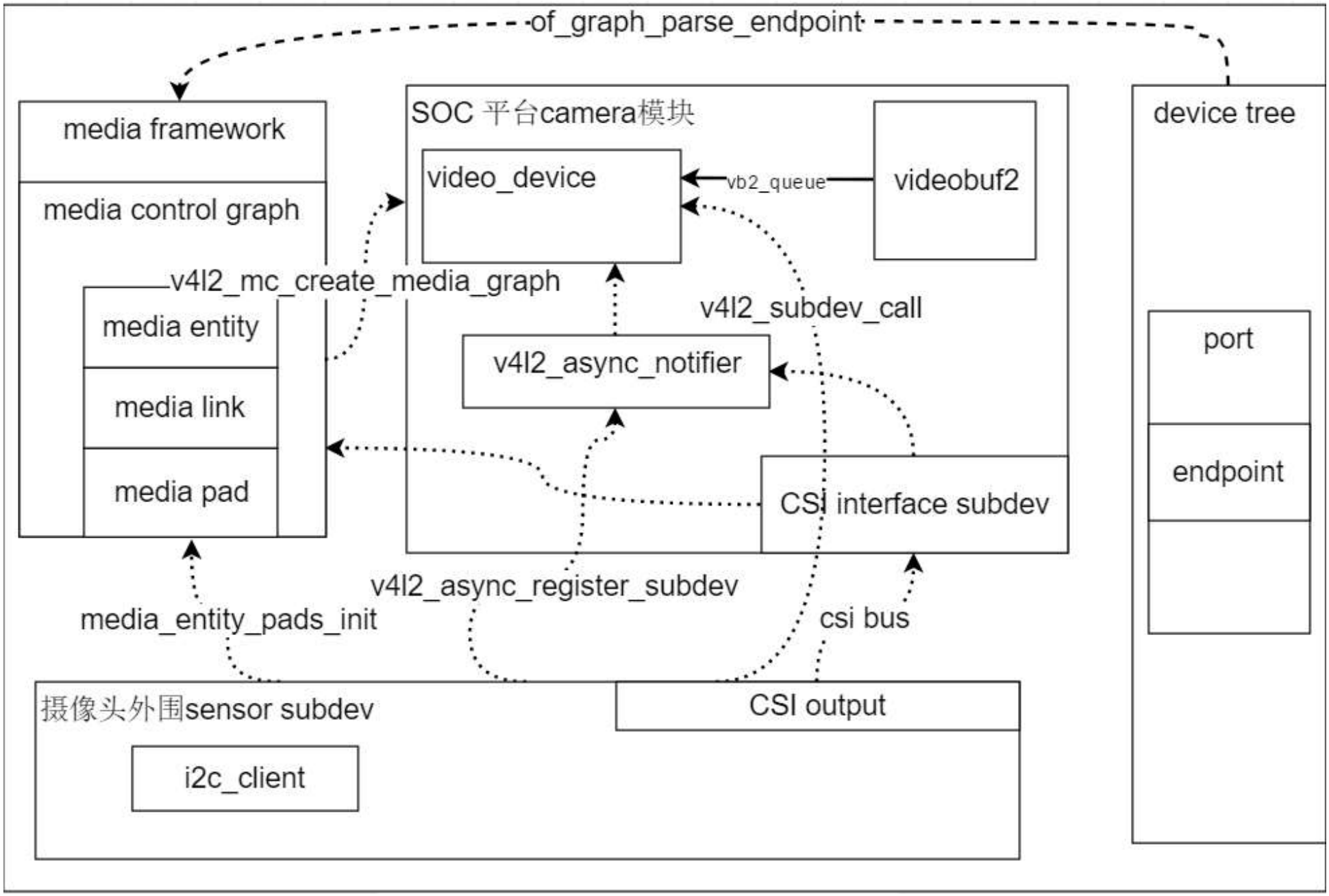
③SOC VIN模块可以设置不同的视频数据格式

不同平台的SOC, 图像处理模块设计不同,media graph也不一样.具体详见下一节v4l2设备节点组织流程中有分析.

6.v4l2设备节点组织流程

在media framework一文中,应用层根据不同需要, 可以通过设置media entity属性,组织不同的模块,构建一个media pipeline.

media pipeline 是在media control graph 选出来的一个通道, 可以这么理解media graph是一张地图, pipeline就是其中的一条规划好的路线. 而media control graph需要 内核组织media device节点去构建.在V4L2中, 需要 media framework, V4L2 asyn模块, 以及设备树等模块构建 media control graph. 下面是V4l2设备节点组织图.



6.1设备树分析:

设备树通过endpoint组织每一个点到点的连接.

① 先从ti964设备树节点出发, 定义了CSI2_964 endpoint 并连接到CSI40_IN endpoint. endpoint通过v4l2-fwnode.c内部函数解析.

```
ds90ub964@3D {
        status = "okay";
        compatible = "ti,ds90ub964";
        reg = <0x3D>;

         pdb-gpio = <&gpio6 4 GPIO_ACTIVE_HIGH>;


        port {
        csi2_964: endpoint@0{
                clock-lanes = <0>;
                data-lanes = <3 1 4 2>;
                lane-polarities = <0 0 1 0 1>;
                remote-endpoint = <&csi40_in>;
                };
         };
};
```

② csi40_in的endpoint为 SOC的 CSI40 设备节点的endpoint,要注意该endpoint上的port的reg等于0. csi40驱动通过V4L2-fwnode在解析该port时,设置port的reg为0 的为数据输入port.

```
&csi40 {
        status = "okay";

        ports {
                port@0 {
                        reg = <0>;

                        csi40_in: endpoint {
                                clock-lanes = <0>;
                                data-lanes = <1 2 3 4>;
                                remote-endpoint = <&csi2_964>;
                        };
                };
        };
};
```

③ csi40 中port的reg为1,即包括所有数据输出的endpoint. 输出到vin0csi40 endpoint

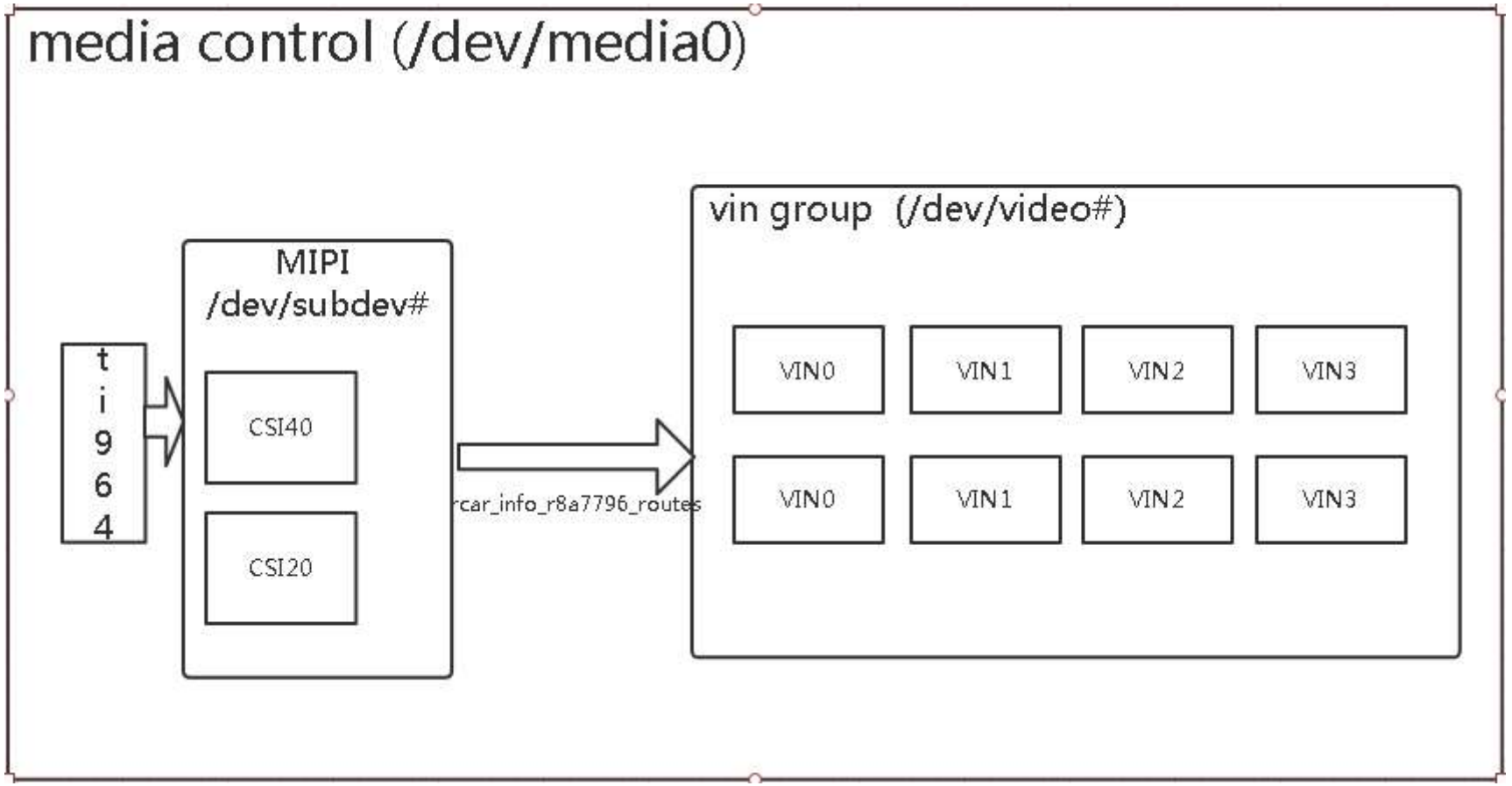
```
csi40: csi2@feaa0000 {
        compatible = "renesas,r8a7796-csi2";
        reg = <0 0xfeaa0000 0 0x10000>;
        interrupts = <GIC_SPI 246 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&cpg CPG_MOD 716>;
        power-domains = <&sysc R8A7796_PD_ALWAYS_ON>;
        resets = <&cpg 716>;
        status = "disabled";

        ports {
                #address-cells = <1>;
                #size-cells = <0>;

                port@1 {
                        #address-cells = <1>;
                        #size-cells = <0>;

                        reg = <1>;

                        csi40vin0: endpoint@0 {
                                reg = <0>;
                                remote-endpoint = <&vin0csi40>;
                        };
                        csi40vin1: endpoint@1 {
                                reg = <1>;
                                remote-endpoint = <&vin1csi40>;
                        };
                        csi40vin2: endpoint@2 {
                                reg = <2>;
                                remote-endpoint = <&vin2csi40>;
                        };
```

csi40vin output to vin

③ vin0 中vin0csi20 endpoint 连接csi20vin0 endpoint

```
vin0: video@e6ef0000 {
        compatible = "renesas,vin-r8a7796";
        reg = <0 0xe6ef0000 0 0x1000>;
        interrupts = <GIC_SPI 188 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&cpg CPG_MOD 811>;
        power-domains = <&sysc R8A7796_PD_ALWAYS_ON>;
        resets = <&cpg 811>;
        renesas,id = <0>;
        status = "disabled";

        ports {
                #address-cells = <1>;
                #size-cells = <0>;

                port@1 {
                        #address-cells = <1>;
                        #size-cells = <0>;

                        reg = <1>;

                        vin0csi20: endpoint@0 {
                                reg = <0>;
                                remote-endpoint= <&csi20vin0>;
                        };
                        vin0csi40: endpoint@2 {
                                reg = <2>;
                                remote-endpoint= <&csi40vin0>;
                        };
                };
        };
};
```

driver通过设备树的endpoint将ti964 sensor, csi, vin 模块组成一个整体的media control graph.如下:



6.2 驱动注册流程分析

6.2.1 sensor驱动异步注册流程

在sensor ti 964驱动的ds90ub964_probe函数中,解析设备树获取 ti964 endpoint, 异步注册subdev.下面分析流程:

① 通过解析device tree 中endpoint获取fwnode赋值给sd->fwnode. fwnode成员是用来做match的,在SOC中的CSI驱动中会分析到.

```
for_each_endpoint_of_node(np, ep_np) {
        of_graph_parse_endpoint(ep_np, &ep);

        v4l2_fwnode_endpoint_parse(of_fwnode_handle(ep_np), &v4l2_ep);

        flags = v4l2_ep.bus.parallel.flags;
        mbus_type = v4l2_ep.bus_type;
        debug_msg( "Endpoint %s on port %d  flags = 0x%x mbus_type = %d\n",
                        of_node_full_name(ep.local_node),ep.port,flags,mbus_type);
        if (v4l2_ep.bus_type == V4L2_MBUS_PARALLEL &&
        !(flags & V4L2_MBUS_HSYNC_ACTIVE_HIGH &&
        flags & V4L2_MBUS_VSYNC_ACTIVE_HIGH &&
        flags & V4L2_MBUS_FIELD_EVEN_LOW)) {

        }
        of_node_get(ep_np);
        if(CURRENT_PORT ==  ep.port){
                sd->fwnode =of_fwnode_handle(ep_np);//only get one not support multi
        }
}
```

② 初始化sd->media_entity,设置flag为 MEDIA_PAD_FL_SOURCE. 有source就需要sink,这个是gstream的概念, source与sink分别类似于camera与screen.

```
pad.flags =  MEDIA_PAD_FL_SOURCE;
sd->flags |= V4L2_SUBDEV_FL_HAS_DEVNODE;

sd->owner = client->dev.driver->owner;
sd->dev = &client->dev;

sd->entity.function = MEDIA_ENT_F_UNKNOWN;
ret = media_entity_pads_init(&sd->entity, 1,
                &pad);
```


③ 然后调用 v4l2_async_register_subdev(sd) 异步注册

```
int v4l2_async_register_subdev(struct v4l2_subdev *sd)
{
        struct v4l2_async_notifier *subdev_notifier;
        struct v4l2_async_notifier *notifier;
        int ret;

        /*
         * No reference taken. The reference is held by the device
         * (struct v4l2_subdev.dev), and async sub-device does not
         * exist independently of the device at any point of time.
         */
        if (!sd->fwnode && sd->dev)
                sd->fwnode = dev_fwnode(sd->dev);

        mutex_lock(&list_lock);

        INIT_LIST_HEAD(&sd->async_list);

    ① list_for_each_entry(notifier, &notifier_list, list) {
                struct v4l2_device *v4l2_dev =
                        v4l2_async_notifier_find_v4l2_dev(notifier);
                struct v4l2_async_subdev *asd;

                if (!v4l2_dev)
                        continue;

           ②    asd = v4l2_async_find_match(notifier, sd);
                if (!asd)
                        continue;

                ret = v4l2_async_match_notify(notifier, notifier->v4l2_dev, sd,
                                              asd);
                if (ret)
                        goto err_unbind;

           ③    ret = v4l2_async_notifier_try_complete(notifier);
                if (ret)
                        goto err_unbind;

                goto out_unlock;
        }

        /* None matched, wait for hot-plugging */
    ④   list_add(&sd->async_list, &subdev_list);
```

```
static LIST_HEAD(subdev_list);
static LIST_HEAD(notifier_list);
```

静态链表头变量

1. 遍历notifier_list为头静态链表成员.
2. 查找match到了相应的asd.

3. 如果查找到调用v4l2_async_notifier中ops->complete
4. 没有匹配到就加入subdev_list链表头, 等待hot-plugging真正注册,这大概就是异步的含义吧.

分析上述逻辑:

1. 如果notifier之前没注册, v4l2_async_register_subdev功能就是将subdev加入subdev_list链表中,而且没有真正注册.
2. 如果有notifier注册,一般在soc video驱动中注册,下文有分析.

两个的静态变量notifier_list和subdev_list很关键, 后续的异步调用都与它链接到的成员有关.

6.3 SOC中CSI驱动异步注册流程

在SOC CSI的驱动的rcsi2_probe函数中. CSI 解析设备树获取 ti964 endpoint, 并通过匹配sd->fwnode

6.3.1 通过 fwnode_graph_get_remote_endpoint获取ti964 的endpoint

```
        priv->asd.match.fwnode.fwnode =
                fwnode_graph_get_remote_endpoint(of_fwnode_handle(ep));
        priv->asd.match_type = V4L2_ASYNC_MATCH_FWNODE;

        of_node_put(ep);

        priv->notifier.subdevs = devm_kzalloc(priv->dev,
                                              sizeof(*priv->notifier.subdevs),
                                              GFP_KERNEL);
        if (!priv->notifier.subdevs)
                return -ENOMEM;

        priv->notifier.num_subdevs = 1;
        priv->notifier.subdevs[0] = &priv->asd;
        priv->notifier.ops = &rcar_csi2_notify_ops;

        dev_dbg(priv->dev, "Found '%pOF'\n",
                to_of_node(priv->asd.match.fwnode.fwnode));

        return v4l2_async_subdev_notifier_register(&priv->subdev,
                                                   &priv->notifier);
}
```

6.3.2 调用__v4l2_async_subdev_notifier_register 注册notifier.

```
static int __v4l2_async_notifier_register(struct v4l2_async_notifier *notifier)
{
        struct device *dev =
                notifier->v4l2_dev ? notifier->v4l2_dev->dev : NULL;
        struct v4l2_async_subdev *asd;
        int ret;
        int i;

        if (notifier->num_subdevs > V4L2_MAX_SUBDEVS)
                return -EINVAL;

        INIT_LIST_HEAD(&notifier->waiting);
        INIT_LIST_HEAD(&notifier->done);

        mutex_lock(&list_lock);

        for (i = 0; i < notifier->num_subdevs; i++) {
                asd = notifier->subdevs[i];

                switch (asd->match_type) {
                case V4L2_ASYNC_MATCH_CUSTOM:
                case V4L2_ASYNC_MATCH_DEVNAME:
                case V4L2_ASYNC_MATCH_I2C:
                        break;
                case V4L2_ASYNC_MATCH_FWNODE:
                        break;
                default:
                        dev_err(dev, "Invalid match type %u on %p\n",
                                asd->match_type, asd);
                        ret = -EINVAL;
                        goto err_unlock;
                }
    ①           list_add_tail(&asd->list, &notifier->waiting);
        }

    ②   ret = v4l2_async_notifier_try_all_subdevs(notifier);
        if (ret < 0)
                goto err_unbind;

    ③   ret = v4l2_async_notifier_try_complete(notifier);
        if (ret < 0)
                goto err_unbind;

        /* Keep also completed notifiers on the list */
    ④   list_add(&notifier->list, &notifier_list);

        mutex_unlock(&list_lock);

        return 0;
```

6.3.3 v4l2_async_notifier_try_all_subdevs 流程分析

```
/* Test all async sub-devices in a notifier for a match. */
static int v4l2_async_notifier_try_all_subdevs(
        struct v4l2_async_notifier *notifier)
{
        struct v4l2_device *v4l2_dev =
                v4l2_async_notifier_find_v4l2_dev(notifier);
        struct v4l2_subdev *sd;

        if (!v4l2_dev)
                return 0;

again:
        list_for_each_entry(sd, &subdev_list, async_list) {
                struct v4l2_async_subdev *asd;
                int ret;

                asd = v4l2_async_find_match(notifier, sd);
                if (!asd)
                        continue;

                ret = v4l2_async_match_notify(notifier, v4l2_dev, sd, asd);
                if (ret < 0)
                        return ret;

                /*
                 * v4l2_async_match_notify() may lead to registering a
                 * new notifier and thus changing the async subdevs
                 * list. In order to proceed safely from here, restart
                 * parsing the list from the beginning.
                 */
                goto again;
        }

        return 0;
}
```

① 遍历静态链表头subdev_list.

② v4l2_async_find_match查找match到的asd

v4l2_async_find_match 函数功能为,根据match_type找到合适的async_subdev,并返回, match_type在CSI驱动中已经设置.如下:

```
static struct v4l2_async_subdev *v4l2_async_find_match(
        struct v4l2_async_notifier *notifier, struct v4l2_subdev *sd)
{
        bool (*match)(struct v4l2_subdev *, struct v4l2_async_subdev *);
        struct v4l2_async_subdev *asd;

        list_for_each_entry(asd, &notifier->waiting, list) {
                /* bus_type has been verified valid before */
                switch (asd->match_type) {
                case V4L2_ASYNC_MATCH_CUSTOM:
                        match = match_custom;
                        break;
                case V4L2_ASYNC_MATCH_DEVNAME:
                        match = match_devname;
                        break;
                case V4L2_ASYNC_MATCH_I2C:
                        match = match_i2c;
                        break;
                case V4L2_ASYNC_MATCH_FWNODE:
                        match = match_fwnode;
                        break;
                default:
                        /* Cannot happen, unless someone breaks us */
                        WARN_ON(true);
                        return NULL;
                }

                /* match cannot be NULL here */
                if (match(sd, asd))
                        return asd;
        }

        return NULL;
}
```

在CSI驱动中, 已经将match_type 设置为 V4L2_ASYNC_MATCH_FWNODE,所以sensor driver中必须设置sd->fwnode, 否则match失败,注册流程会异常.

```
static int rcar_csi2_parse_dt(struct rcar_csi2 *priv)
{
        struct device_node *ep;
        struct v4l2_fwnode_endpoint v4l2_ep;
        int ret;

        ep = of_graph_get_endpoint_by_regs(priv->dev->of_node, 0, 0);
        if (!ep) {
                dev_dbg(priv->dev, "Not connected to subdevice\n");
                return 0;
        }

        ret = v4l2_fwnode_endpoint_parse(of_fwnode_handle(ep), &v4l2_ep);
        if (ret) {
                dev_err(priv->dev, "Could not parse v4l2 endpoint\n");
                of_node_put(ep);
                return -EINVAL;
        }

        ret = rcar_csi2_parse_v4l2(priv, &v4l2_ep);
        if (ret)
                return ret;

        priv->remote.match.fwnode.fwnode =
                fwnode_graph_get_remote_endpoint(of_fwnode_handle(ep));
        priv->remote.match_type = V4L2_ASYNC_MATCH_FWNODE;

        of_node_put(ep);
```

③ v4l2_async_match_notify

```
static int v4l2_async_match_notify(struct v4l2_async_notifier *notifier,
                                   struct v4l2_device *v4l2_dev,
                                   struct v4l2_subdev *sd,
                                   struct v4l2_async_subdev *asd)
{
        struct v4l2_async_notifier *subdev_notifier;
        int ret;

    ①   ret = v4l2_device_register_subdev(v4l2_dev, sd);
        if (ret < 0)
                return ret;

    ②   ret = v4l2_async_notifier_call_bound(notifier, sd, asd);
        if (ret < 0) {
                v4l2_device_unregister_subdev(sd);
                return ret;
        }

        /* Remove from the waiting list */
        list_del(&asd->list);
        sd->asd = asd;
        sd->notifier = notifier;

        /* Move from the global subdevice list to notifier's done */
        list_move(&sd->async_list, &notifier->done);

        ret = v4l2_subdev_call(sd, core, registered_async);
        if (ret < 0 && ret != -ENOIOCTLCMD)
                return ret;

        /*
         * See if the sub-device has a notifier. If not, return here.
         */
    ③   subdev_notifier = v4l2_async_find_subdev_notifier(sd);
        if (!subdev_notifier || subdev_notifier->parent)
                return 0;

        /*
         * Proceed with checking for the sub-device notifier's async
         * sub-devices, and return the result. The error will be handled by the
         * caller.
         */
        subdev_notifier->parent = notifier;

    ④   return v4l2_async_notifier_try_all_subdevs(subdev_notifier);
}
```

1. v4l2_device_register_subdev调用subdev驱动中实现的registered函数: sd->internal_ops->registered

II.v4l2_async_notifier_call_bound调用驱动中实现的bound函数:notifier->ops->bound

CSI driver 实现为:rcar_csi2_notify_bound

III.查找该sub-device的notifier

IV.又调用v4l2_async_notifier_try_all_subdevs是不是晕了? 两个函数相互调用!!!

最后分析下 __v4l2_async_subdev_notifier_register 中 v4l2_async_notifier_try_complete

流程:

v4l2_async_notifier_try_complete

-> v4l2_async_notifier_call_complete

-> notifier->ops->complete但是 csi2中只有ops->bound.因为它也只是一个subdev,无法建立一个pipeline. Bound完成调用 media_create_pad_link 建立连接即可.

```
static const struct v4l2_async_notifier_operations rcar_csi2_notify_ops = {
        .bound = rcar_csi2_notify_bound,
};
```

```
static int rcsi2_notify_bound(struct v4l2_async_notifier *notifier,
                              struct v4l2_subdev *subdev,
                              struct v4l2_async_subdev *asd)
{
        struct rcar_csi2 *priv = notifier_to_csi2(notifier);
        int pad;

        pad = media_entity_get_fwnode_pad(&subdev->entity,
                                          asd->match.fwnode.fwnode,
                                          MEDIA_PAD_FL_SOURCE);
        if (pad < 0) {
                dev_err(priv->dev, "Failed to find pad for %s\n", subdev->name);
                return pad;
        }

        priv->remote = subdev;

        dev_dbg(priv->dev, "Bound %s pad: %d\n", subdev->name, pad);

        return media_create_pad_link(&subdev->entity, pad,
                                     &priv->subdev.entity, 0,
                                     MEDIA_LNK_FL_ENABLED |
                                     MEDIA_LNK_FL_IMMUTABLE);
}
```


小结: __v4l2_async_subdev_notifier_register跟之前的v4l2_async_register_subdev 逻辑很相似,再加上两个函数相互调用,这也是V4L2-async 容易搞混淆的地方.

6.4 Video 驱动注册节点流程

与csi异步机制机制一样vin probe中注册一个async_notifier

6.4.1 Video 解析整个endpoint

video驱动解析设备树遍历CSI endpoint,从probe开始如下:

rcar_vin_probe

-> rvin_group_init

->rvin_group_graph_parse

这里很多逻辑涉及到SOC的图像处理模块与CSI接口模块的之间硬件通道,不同的soc逻辑差异很大. 但一般都是通过设备树节点,获取subdev对应的entity,组成一个 media graph.

rvin_group_graph_parse函数部分源码

```
/* Parse all endpoints for CSI-2 and VIN nodes */
for (i = 0; i < RVIN_CSI_MAX; i++) {
        struct device_node *ep, *csi, *remote;

        /* Check if instance is connected to the CSI-2 */
        ep = of_graph_get_endpoint_by_regs(np, 1, i);
        if (!ep) {
                vin_dbg(vin, "VIN%d: ep %d not connected\n", id, i);
                continue;
        }

        if (vin->group->csi[i].asd.match.fwnode.fwnode) {
                of_node_put(ep);
                vin_dbg(vin, "VIN%d: ep %d already handled\n", id, i);
                continue;
        }

        csi = rvin_group_get_remote(vin, ep);
        of_node_put(ep);
        if (!csi)
                continue;

        vin->group->csi[i].asd.match.fwnode.fwnode =
                of_fwnode_handle(csi);
        vin->group->csi[i].asd.match_type = V4L2_ASYNC_MATCH_FWNODE;

        vin_dbg(vin, "VIN%d ep: %d handled CSI-2 %s\n", id, i,
                of_node_full_name(csi));

        /* Parse the CSI-2 for all VIN nodes connected to it */
        ep = NULL;
        while (1) {
                ep = of_graph_get_next_endpoint(csi, ep);
                if (!ep)
```

```
                        break;

                remote = rvin_group_get_remote(vin, ep);
                if (!remote)
                        continue;

                if (of_match_node(vin->dev->driver->of_match_table,
                                  remote)) {
                        ret = rvin_group_graph_parse(vin, remote);
                        if (ret)
                                return ret;

                }A
        }
}
```

- 解析在VIN 和CSI 上的所有endpoint.
- 解析与CSI连接的endpoint
- 已经解析过的会保存在VIN->group, 避免重复分析
- 再次调用rvin_group_graph_parse,递归深度遍历

解析过程,驱动会将已经解析的endpoint保存到vin->group. 其中fwnode值可用于subdev 的注册.

6.4.2 v4l2_async_notifier_operations回调

```
static const struct v4l2_async_notifier_operations rvin_group_notify_ops = {
        .bound = rvin_group_notify_bound,
        .unbind = rvin_group_notify_unbind,
        .complete = rvin_group_notify_complete,
};
```

6.4.3 match到相应的subdev 后回调v4l2_async_notifier_operations 中的bound 和 complete

① 回调 bound, 根据fwnode获取CSI的subdev 赋值到 vin->group->csi[i].subdev

```
static int rvin_group_notify_bound(struct v4l2_async_notifier *notifier,
                                   struct v4l2_subdev *subdev,
                                   struct v4l2_async_subdev *asd)
{
        struct rvin_dev *vin = notifier_to_vin(notifier);
        unsigned int i;

        mutex_lock(&vin->group->lock);

        for (i = 0; i < RVIN_CSI_MAX; i++) {
                if (vin->group->csi[i].fwnode != asd->match.fwnode.fwnode)
                        continue;
                vin->group->csi[i].subdev = subdev;
                vin_dbg(vin, "Bound CSI-2 %s to slot %u\n", subdev->name, i);
                break;
        }

        mutex_unlock(&vin->group->lock);

        return 0;
}
```

② 回调complete,即rvin_group_notify_complete

```
static int rvin_group_notify_complete(struct v4l2_async_notifier *notifier)
{
        struct rvin_dev *vin = notifier_to_vin(notifier);
        int ret;

        ret = v4l2_device_register_subdev_nodes(&vin->v4l2_dev);
        if (ret) {
                vin_err(vin, "Failed to register subdev nodes\n");
                return ret;
        }

        return rvin_group_update_links(vin);
}
```

③ 调用v4l2_device_register_subdev_nodes,真正注册subdev设备节点.

```
int v4l2_device_register_subdev_nodes(struct v4l2_device *v4l2_dev)
{
        struct video_device *vdev;
        struct v4l2_subdev *sd;
        int err;

        /* Register a device node for every subdev marked with the
         * V4L2_SUBDEV_FL_HAS_DEVNODE flag.
         */
        list_for_each_entry(sd, &v4l2_dev->subdevs, list) {
                if (!(sd->flags & V4L2_SUBDEV_FL_HAS_DEVNODE))
                        continue;

                if (sd->devnode)
                        continue;

                vdev = kzalloc(sizeof(*vdev), GFP_KERNEL);
                if (!vdev) {
                        err = -ENOMEM;
                        goto clean_up;
                }

                video_set_drvdata(vdev, sd);
                strlcpy(vdev->name, sd->name, sizeof(vdev->name));
                vdev->v4l2_dev = v4l2_dev;
                vdev->fops = &v4l2_subdev_fops;
                vdev->release = v4l2_device_release_subdev_node;
                vdev->ctrl_handler = sd->ctrl_handler;
                err = __video_register_device(vdev, VFL_TYPE_SUBDEV, -1, 1,
                                              sd->owner);
                if (err < 0) {
                        kfree(vdev);
                        goto clean_up;
                }
                sd->devnode = vdev;
#if defined(CONFIG_MEDIA_CONTROLLER)
                sd->entity.info.dev.major = VIDEO_MAJOR;
```

小结: SOC的内部图像处理模块,创建的video节点时,需要通过遍历设备树的endpoint,借助media framework, subdev, V4L2-async等子模块,分析media entity节点的拓扑是否符合SOC的要求.也就说media pipeline 能不能组建成功,以及如何组建依赖于soc video驱动构建的media graph. video驱动的设计的目的是将soc图像处理硬件的功能充分发挥. 所以想要理解video驱动的细节还必须必须仔细分析soc硬件模块功能. 最好结合代码, 硬件文档一起分析.