## 作者

彭东林
pengdonglin137@163.com

## 平台

TQ2440
Linux-4.9

## 概述

上一篇直接操作DMA控制器实现了一个mem2mem的DMA传输，但是这样不符合linux driver设计的思想，应该将DMA驱动拆成几个部分：

- DMA控制器驱动
- DMA core
- DMA设备驱动

可以类比I2C以及SPI驱动的框架， 这一节我们学习一下DMA控制器驱动。
S3C2440的DMA驱动对应的文件是drivers/dma/s3c24xx-dma.c，代码已经上传到github上了，可以用下面
的命令下载：

```
1. git clone git@github.com:pengdonglin137/linux-4.9.git -b tq2440_dt
```

## 参考

Linux DMA Engine framework(3)_dma controller驱动
内核文档：Documentation/dmaengine/provider.txt

## 正文

## 一、设备树

对应的文件：arch/arm/boot/dts/s3c2440-tq2440-dt.dts

```
    dma: s3c2410-dma@4B000000 {
        compatible = "s3c2440-dma";
        reg = <0x4B000000 0x1000>;
        interrupts = <0 0 17 3>, <0 0 18 3>,
                <0 0 19 3>, <0 0 20 3>;
        #dma-cells = <1>;
    };
```

简单解释以下：

- reg属性表示的是DMA控制器的寄存器资源的地址空间
- interrupts表示的是DMA控制器的中断资源，在讲芯片手册的那一篇博客中已经说过了
- #dma-cells表示引用一个dma资源需要传递几个参数，这个是DMA控制器必备的，比如引用DMA资源的I2S设备树节点如下：

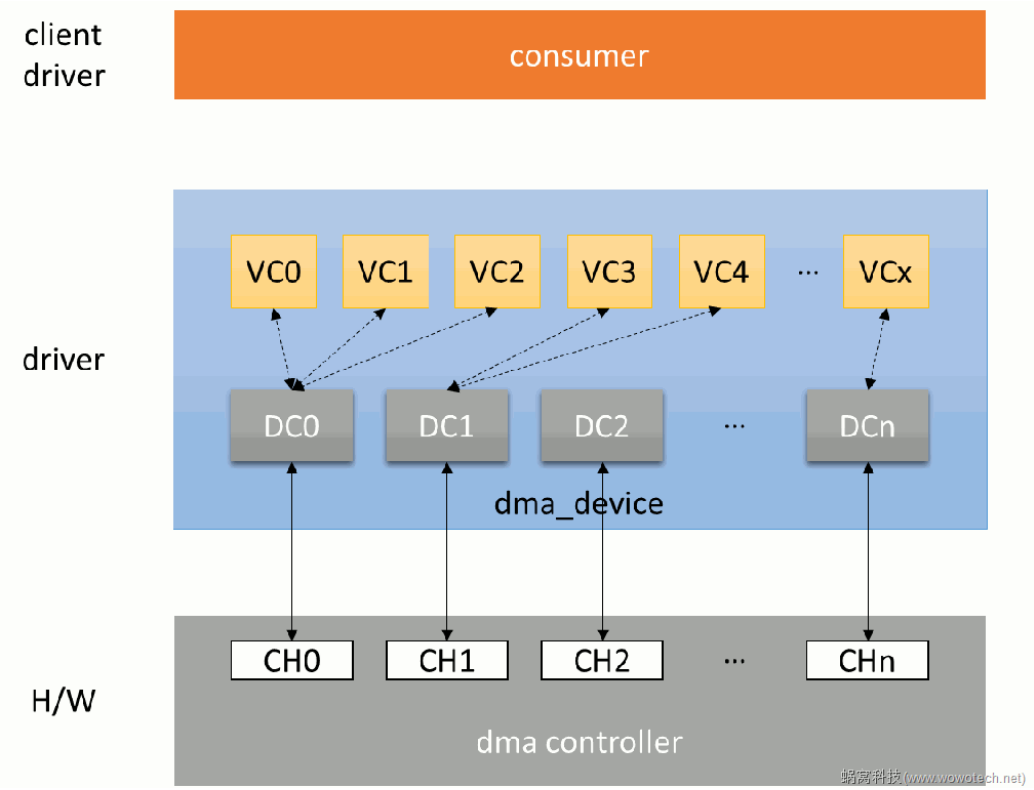
```
    s3c2440_iis@55000000 {
        compatible = "s3c24xx-iis";
        reg = <0x55000000 0x100>;
        clocks = <&clocks PCLK_I2S>;
        clock-names = "iis";
        pinctrl-names = "default";
        pinctrl-0 = <&s3c2440_iis_pinctrl>;
        dmas = <&dma DMACH_I2S_IN>, <&dma DMACH_I2S_OUT>;
        dma-names = "rx", "tx";
    };
```

上面的dmas和dma-names两个属性描述了两个DMA资源，在具体的驱动中申请虚拟DMA Channel的时候可以使用下面的接口：

```
ch_capture  = dma_request_chan(dev, "rx");
ch_playback = dma_request_chan(dev, "tx");
```

# 二、概念

## 1、虚拟DMA和物理DMA




图中的dma_device表示DMA控制器，DCx表示一个物理DMA在driver中的抽象，VCx表示的是虚拟DMA通道，可以看到，一个物理DMA通道上可以关联多个虚拟DMA通道，但是某一个时刻一个DMA物理通道只能为一个虚拟DMA通道服务，处理完一个虚拟DMA通道上的请求后，才可以服务下一个。

## 2、DMA_MEMCPY 和 DMA_SLAVE

这两个表示DMA控制器的能力，前一个表示支持两个块内存之间数据的DMA传输，而DMA_SLAVE表示支持内存和其他外设之间的DMA数据传输。
对于S3C2440来说，所有的物理DMA通道都支持DMA_MEMCPY，也就是只要是MEM2MEM的虚拟DMA，在需要DMA传输时只要直到一条空闲的物理DMA通道即可，但是对DMA_SLAVE，虚拟DMA在关联物理DMA的时候会有限制：

| | Source0 | Source1 | Source2 | Source3 | Source4 | Source5 | Source6 |
|---|---|---|---|---|---|---|---|
| Ch-0 | nXDREQ0 | UART0 | SDI | Timer | USB device EP1 | I2SSDO | PCMIN |
| Ch-1 | nXDREQ1 | UART1 | I2SSDI | SPI0 | USB device EP2 | PCMOUT | SDI |
| Ch-2 | I2SSDO | I2SSDI | SDI | Timer | USB device EP3 | PCMIN | MICIN |
| Ch-3 | UART2 | SDI | SPI1 | Timer | USB device EP4 | MICIN | PCMOUT |

上面的图中表示的就是虚拟DMA跟物理DMA的对应关系，妻子Ch-x表示的就是物理DMA通道，而Sourcex下面的每一项都是一个虚拟DMA通道，比如UART0只能关联到物理DMA0上面，而I2SSDO可以关联到物理DMA0/2上。所以在软件抽象上，可以将一个DMA控制器上创建两个dma_device，一个具备DMA_MEMCPY，另一个具备DMA_SLAVE。

## 三、驱动

这个驱动还是比较简单的，就不一行一行分析了，这里仅仅分析一些关键和不易看懂的地方。
虚拟

## 1、数据结构

- s3c24xx_dma_chan_state

```
enum s3c24xx_dma_chan_state {
    S3C24XX_DMA_CHAN_IDLE,
    S3C24XX_DMA_CHAN_RUNNING,
    S3C24XX_DMA_CHAN_WAITING,
};
```

用于表示虚拟DMA通道的状态：
S3C24XX_DMA_CHAN_IDLE： 表示该虚拟通道空闲，没有分配物理DMA通道
S3C24XX_DMA_CHAN_RUNNING： 表示该虚拟通道已经分配到物理DMA通道，并且正在进行数据传输
S3C24XX_DMA_CHAN_WAITING： 表示该虚拟通道正在等待一个物理DMA通道，也就是给该虚拟通道分配物理通道时，发现可用的物理通道资源都被占用了

- struct s3c24xx_sg

```
struct s3c24xx_sg {
    dma_addr_t src_addr;
    dma_addr_t dst_addr;
    size_t len;
    struct list_head node;
};
```

存放启动一次DMA，需要传输的数据的源地址/目的地址/长度等的信息， 该结构体会通过node结构体挂到s3c24xx_txd的dsg_list上，下面简称dsg。

- s3c24xx_txd

```
struct s3c24xx_txd {
    struct virt_dma_desc vd;
    struct list_head dsg_list;
    struct list_head *at;
    u8 width;
    u32 disrcc;
    u32 didstc;
    u32 dcon;
    bool cyclic;
};
```

这个结构体是对dma_async_tx_descriptor的封装，每次发起一轮新的DMA传输传输，都需要重新分配一个dma_async_tx_descriptor，每一轮DMA传输可能需要启动DMA多次，每次处理一个s3c24xx_sg，直到该txd下的所有dsg都处理完。

其中，at指向的是当前正在处理的s3c24xx_sg

- struct s3c24xx_dma_phy

```
struct s3c24xx_dma_phy {
    unsigned int        id;
    bool                valid;
    void __iomem            *base;
    int             irq;
    struct clk          *clk;
    spinlock_t          lock;
    struct s3c24xx_dma_chan     *serving;
    struct s3c24xx_dma_engine   *host;
};
```

表示一个物理DMA通道，base是该物理DMA的寄存器物理起始地址，irq是该占用的系统中断资源，serving指向的是该物理DMA通道当前正关联的虚拟DMA通道。

- struct s3c24xx_dma_chan

```
struct s3c24xx_dma_chan {
    int id;
    const char *name;
    struct virt_dma_chan vc;
    struct s3c24xx_dma_phy *phy;
    struct dma_slave_config cfg;
    struct s3c24xx_txd *at;
    struct s3c24xx_dma_engine *host;
    enum s3c24xx_dma_chan_state state;
    bool slave;
};
```

表示一条虚拟DMA通道，是对virt_dma_chan的封装，phy指向的是分配的物理DMA通道，at指向的是当前正在处理的s3c24xx_txd

struct s3c24xx_dma_engine

```
struct s3c24xx_dma_engine {
    struct platform_device          *pdev;
    const struct s3c24xx_dma_platdata   *pdata;
    struct soc_data             *sdata;
    void __iomem                *base;
    struct dma_device           slave;
    struct dma_device           memcpy;
    struct s3c24xx_dma_phy          *phy_chans;
};
```

用于抽象SoC的DMA信息，base是第一个DMA的寄存器物理地址，memcpy是具备DMA_MEMCPY能力的dma控制器，slave是具备DMA_SLAVE能力的dma控制器，py_chans指向所有的物理DMA通道。

## 2、关键函数

- s3c24xx_dma_probe

**初始化负责mem2mem的dma device：**

```
1.      dma_cap_set(DMA_MEMCPY, s3cdma->memcpy.cap_mask);
2.      dma_cap_set(DMA_PRIVATE, s3cdma->memcpy.cap_mask);
3.      s3cdma->memcpy.dev = dev;
4.      s3cdma->memcpy.device_free_chan_resources =
5.                      s3c24xx_dma_free_chan_resources;
6.      s3cdma->memcpy.device_prep_dma_memcpy = s3c24xx_dma_prep_memcpy;
7.      s3cdma->memcpy.device_tx_status = s3c24xx_dma_tx_status;
8.      s3cdma->memcpy.device_issue_pending = s3c24xx_dma_issue_pending;
9.      s3cdma->memcpy.device_config = s3c24xx_dma_set_runtime_config;
10.     s3cdma->memcpy.device_terminate_all = s3c24xx_dma_terminate_all;
11.     s3cdma->memcpy.directions = BIT(DMA_MEM_TO_MEM);
12.     s3cdma->memcpy.src_addr_widths = S3C2440_DMA_BUSWIDTHS;
13.     s3cdma->memcpy.dst_addr_widths = S3C2440_DMA_BUSWIDTHS;
```

**第1行**的DMA_MEMCPY表示： The device is able to do memory to memory copies

**第2行**的DMA_PRIVATE表示： The devices only supports slave transfers, and as such isn't available for async transfers.

**第4行**的device_free_chan_resources用于释放虚拟DMA通道的txd资源以及txd下的dsg

**第6行**的device_prep_dma_memcpy，这个函数会分配一个txd，然后将传入的dest/src/len封装成一个dsg，然后将该dsg加入到txd->dsg_list中，同时根据dest和src对txd的其他成员进行设置，然后将txd(vd)放到dma_chan(vc)的desc_allocated链表，并将txd(vd)的地址返回

**第7行**的device_tx_status用于查询传入的cookie代表txd的传输状态，如果该txd下的所有dsg都已经处理完毕，那么返回DMA_COMPLETE，否着的话返回该txd下所有未处理的dsg所代表的数据总长度。

**第8行**的device_issue_pending是在dmaengine_submit被调用后才调用的，dmaengine_submit会把传入的txd(vd)移入vc的desc_submitted链表，而device_issue_pending会将desc_submitted链表内容全部移入desc_issued链表，如果发现desc_issued非空的话，就会启动DMA传输，当然，在传输之前首先需要为dma_chan分配一个物理DMA通道。

**第9行**的device_config用于修改的dma_chan的配置

**第10行**的device_terminate_all用于停止传入的dma_chan的DMA传输，同时关闭对应的物理DMA通道，如果在该物理DMA通道有其他正在等待处理的dma_chan的话，会继续处理其他dma_chan。

**第11行**的directions表示该dma device支持的传输方向，比如mem2mem/mem2dev/dev2mem等等

**第12行**的src_addr_widths表示DMA传输的read的时候，每个read操作支持的字节数，也就是之前芯片手册里面讲到的data size。
**第13行**的dst_addr_widths表示DMA传输的write的时候，每个write操作支持的字节数，也就是之前芯片手册里面讲到的data size。
注： 对于S3C2440的DMA控制器，read和write的data size始终相同，因为都是由DCON[21:20]控制的

**初始化负责MEM跟外设之间DMA传输的DMA控制器：**

```
1.      dma_cap_set(DMA_SLAVE, s3cdma->slave.cap_mask);
2.      dma_cap_set(DMA_CYCLIC, s3cdma->slave.cap_mask);
3.      dma_cap_set(DMA_PRIVATE, s3cdma->slave.cap_mask);
4.      s3cdma->slave.dev = dev;
5.      s3cdma->slave.device_free_chan_resources =
6.                      s3c24xx_dma_free_chan_resources;
7.      s3cdma->slave.device_tx_status = s3c24xx_dma_tx_status;
8.      s3cdma->slave.device_issue_pending = s3c24xx_dma_issue_pending;
9.      s3cdma->slave.device_prep_slave_sg = s3c24xx_dma_prep_slave_sg;
10.     s3cdma->slave.device_prep_dma_cyclic = s3c24xx_dma_prep_dma_cyclic;
11.     s3cdma->slave.device_config = s3c24xx_dma_set_runtime_config;
12.     s3cdma->slave.device_terminate_all = s3c24xx_dma_terminate_all;
13.     s3cdma->slave.directions = BIT(DMA_DEV_TO_MEM) | BIT(DMA_MEM_TO_DEV);
14.     s3cdma->slave.src_addr_widths = S3C2440_DMA_BUSWIDTHS;
15.     s3cdma->slave.dst_addr_widths = S3C2440_DMA_BUSWIDTHS;
```

**第1行**的DMA_SLAVE的含义：The device can handle device to memory transfers, including scatter-gather transfers.

While in the mem2mem case we were having two distinct types to deal with a single

chunk to copy or a collection of them, here, we just have a single transaction

type that is supposed to handle both. If you want to transfer a single contiguous

memory buffer, simply build a scatter list with only one item.

**第2行**的DMA_CYCLIC的含义： The device can handle **cyclic transfers**. A **cyclic transfer** is a transfer where the

chunk collection will loop over itself, with the last item pointing to the first.

It's usually used for audio transfers, where you want to operate on a single ring

buffer that you will fill with your audio data.

**第9行**的device_prep_slave_sg 会分配一个txd，然后将传入的scatterlist中的每一项都转换成一个dsg，然后dsg加入txd的dsg_list中，最后返回txd

**第10行**的device_prep_dma_cyclic 会分配一个txd，然后以period为单位长度，将需要传输的数据分成若干份，每一份都转换成一个dsg，dsg是启动一

次DMA需要处理的数据量，再将这些dsg加入到txd的dsg_list中，最后将txd返回。

**为dma device添加虚拟DMA通道：s3c24xx_dma_init_virtual_channels:**

这个函数会给传入的dma_device添加指定数目的虚拟DMA通道，这些通道都从0开始编号。对于memcpy这个dma device，给他添加了四条虚拟DMA通道，

这个dma device专门负责mem2mem的传输。对于slave这个dma device，给他添加了18个虚拟DMA通道，每一个通道都对应一个DMA Request Source。

**注册dma device: dma_async_device_register**

这个函数会对传入的dma device下的虚拟DMA通道进行一定的初始化，然后将该dma device加入到全局的链表dma_device_list中。

**将slave这个dma device加入到of_dma_list：of_dma_controller_register**

在外设驱动里调用dma_request_chan的时候会根据传入的字符串解析该设备的device tree的dmas属性，根据dmas属性找到所引用的dma device。

为什么这里不必注册memcpy这个dma deivce到of_dma_list中呢？因为memcpy对DMA物理通道没有要求，所以可以直接调用dma_request_channel

从注册的dma device中获得一个没有被占用的虚拟DMA通道。

## 3、dma_async_tx_descriptor的生命周期

在调用device_issue_pending后，便启动了DMA，开始进行传输，记录当前将要处理的txd，同时将该txd从vc的desc_issued中删除，开始处

理txd的dsg_list中的第一个dsg，每处理完毕一个dsg后都会触发中断

s3c24xx_dma_irq。

在中断处理函数中会获得正在处理的txd， 如果该txd下还有没有处理的dsg，那么会继续启动DMA，处理下一个待处理的dsg，知道把该txd下所有的dsg

都处理完毕，然后会调用vchan_cookie_complete来处理刚处理完毕的txd，这个函数首先将该txd移入vc的desc_completed链表，然后执行

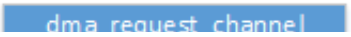
tasklet_schedule(&vc->task)，这个tasklet是在probe中给dma device添加虚拟DMA通道(vc)的时候设置的， 处理函数是vchan_complete。

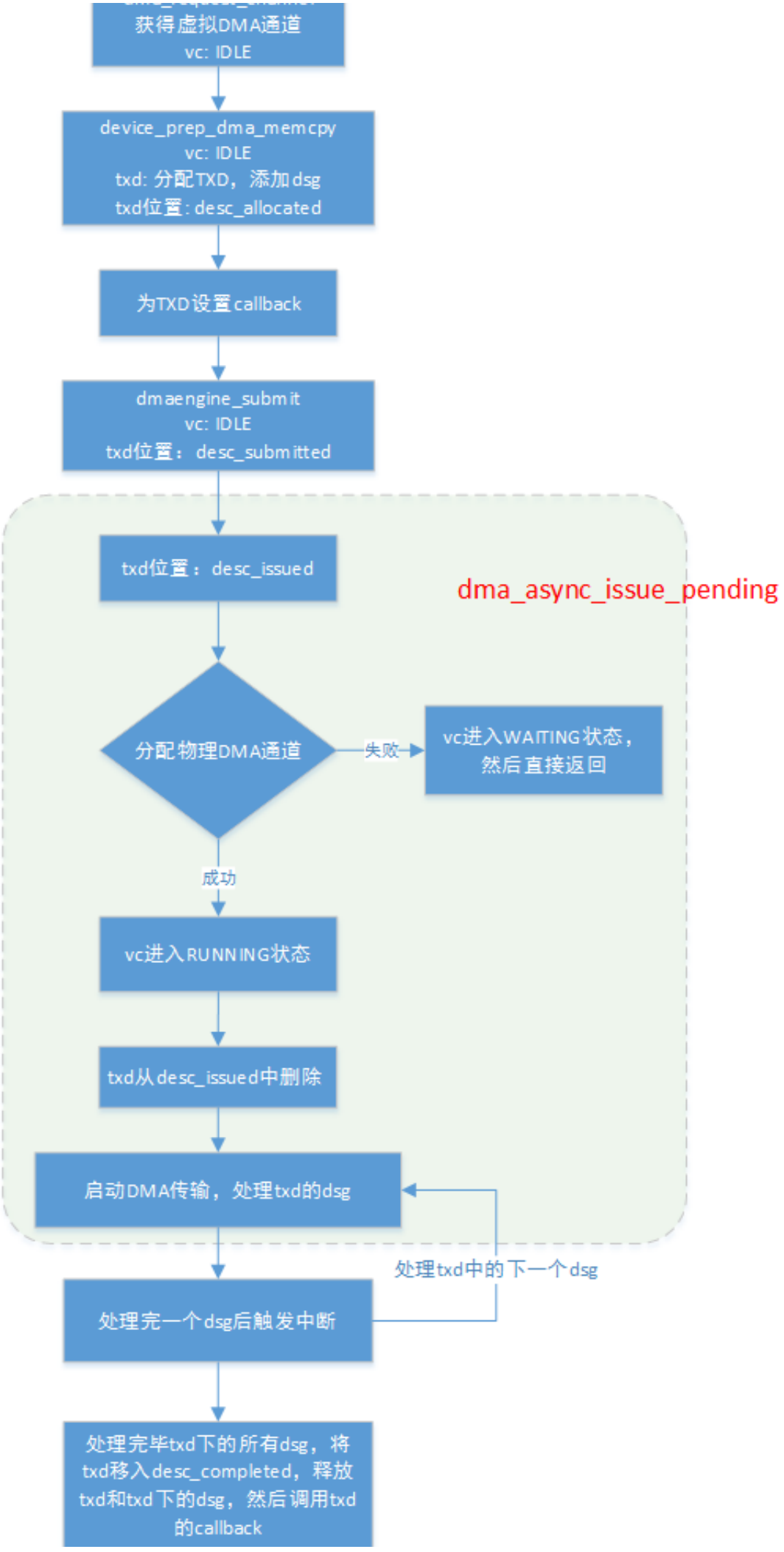
在vchan_complete中会处理vc的desc_completed链表中的每一个txd，如果该txd被设置了DMA_CTRL_REUSE标志，那么该txd会重新被

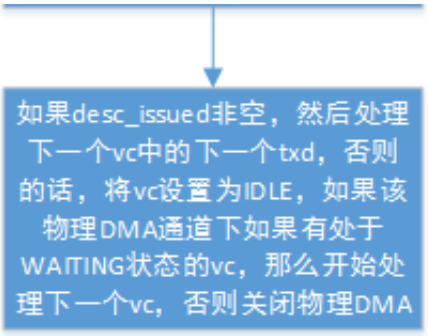
移入vc->desc_allocated， 否则的话，就会被释放，最后调用用户设置给该txd的回调函数， 用于通知用户该txd处理完毕。

在调用完vchan_complete后，会继续处理vc下的其他待处理的txd，如果没有了，就会关闭物理DMA通道。

下图中，vc代表虚拟DMA通道，txd代表s3c24xx_txd， dsg代表s3c24xx_sg




获得虚拟DMA通道
vc: IDLE
device_prep_dma_memcpy
vc: IDLE
txd: 分配TXD，添加dsg
txd位置: desc_allocated
为TXD设置callback
dmaengine_submit
vc: IDLE
txd位置： desc_submitted
txd位置： desc_issued
dma_async_issue_pending
分配物理DMA通道
失败
vc进入WAITING状态，
然后直接返回
成功
vc进入RUNNING状态
txd从desc_issued中删除
启动DMA传输，处理txd的dsg
处理txd中的下一个dsg
处理完一个dsg后触发中断
处理完毕txd下的所有dsg，将
txd移入desc_completed，释放
txd和txd下的dsg，然后调用txd
的callback


如果desc_issued非空，然后处理下一个vc中的下一个txd，否则的话，将vc设置为IDLE，如果该物理DMA通道下如果有处于WAITING状态的vc，那么开始处理下一个vc，否则关闭物理DMA

完