作者

彭东林
pengdonglin137@163.com

平台

TQ2440
Linux-4.9

概述

前面分析了DMA控制器驱动，下面我们调用DMAENGINE的API写一个MEM2MEM的驱动

参考博文

Linux DMA Engine framework(3)_dma controller驱动
Linux DMA Engine framework(2)_功能介绍及解接口分析
内核文档：Documentation/dmaengine/client.txt

正文

一、写一个client驱动的步骤

- 申请一个DMA channel

dma_request_chan 或者 dma_request_channel

- 根据设备（slave）的特性，配置DMA channel的参数

dmaengine_slave_config

- 要进行DMA传输的时候，获取一个用于识别本次传输（transaction）的描述符（descriptor）

dmaengine_prep_slave_sg
dmaengine_prep_dma_cyclic
dmaengine_prep_interleaved_dma
device_prep_dma_memcpy

- 将本次传输（transaction）提交给dma engine并启动传输

dmaengine_submit

dma_async_issue_pending

- 等待传输（transaction）结束

dma_async_is_tx_complete 或者 等待回调函数被执行

二、结合驱动分析

1、驱动源码

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/of.h>
#include <linux/dmaengine.h>
#include <linux/dma-mapping.h>
#include <linux/miscdevice.h>
#include <linux/uaccess.h>

#define BUFSIZE  (4*1024)

typedef struct
{
    struct device *dev;
    struct dma_device *dma_dev;  // dma 控制器
    struct dma_chan *ch;  // 申请到的虚拟DMA通道
    struct dma_async_tx_descriptor *tx;  // 传输描述符
    struct dma_slave_config *cfg;
    struct completion completion;
    dma_addr_t src;  // 存放src物理地址
    dma_addr_t dst;  // 存放dst物理地址
    char *src_virt;  // 存放src对应的虚拟地址
    char *dst_virt;  // 存放dst对应的虚拟地址
    dma_cookie_t cookie;
}mem_dma_t;

static mem_dma_t *mem_dma;

static void mem_dma_dmacb(void *data)
{
    dev_info(mem_dma->dev, "transfer complete.\n");
    complete(&mem_dma->completion);
}

static int mem_dma_open(struct inode *inode, struct file *file)
{
    // 清除垃圾数据
    memset(mem_dma->dst_virt, 0x00, BUFSIZE);
```

```
39.     memset(mem_dma->src_virt, 0x00, BUFSIZE);
40.
41.     return 0;
42. }
43.
44. static ssize_t mem_dma_write(struct file *file, const char *data,
45.                 size_t len, loff_t *ppos)
46. {
47.     struct dma_async_tx_descriptor *tx;
48.     struct dma_device *dma_dev = mem_dma->dma_dev;
49.     struct device *dev = mem_dma->dev;
50.     dma_cookie_t cookie;
51.     int ret;
52.
53.     dev_info(dev, "%s enter.\n", __func__);
54.
55.     // 重新初始化完成变量
56.     reinit_completion(&mem_dma->completion);
57.
58.     // 将用户写下来的数据拷贝到源地址空间
59.     // 注： 这里用的是源地址对应的虚拟地址src_virt
60.     copy_from_user(mem_dma->src_virt, data, len);
61.     dev_info(dev, "from user: %s\n", mem_dma->src_virt);
62.
63.     // 获得一个传输描述符，同时根据传入的dst、src和len封装dsg到tx的dsg_list中
64.     tx = dma_dev->device_prep_dma_memcpy(mem_dma->ch, mem_dma->dst, mem_dma
65.     if (!tx) {
66.         dev_err(dev, "%s: device_prep_dma_memcpy failed\n", __func__);
67.         ret = -EIO;
68.         goto err5;
69.     }
70.
71.     // 设置回调函数，当这个tx的dsg_list中的所有dsg都处理完毕后
72.     // 该回调函数会被执行
73.     tx->callback = mem_dma_dmacb;
74.     tx->callback_param = mem_dma;
75.
76.     // 将tx提交给dma engine
77.     cookie = dmaengine_submit(tx);
78.     ret = dma_submit_error(cookie);
79.     if (ret) {
80.         dev_err(dev, "dma_submit_error %d\n", cookie);
81.         ret = -EIO;
82.         goto err5;
83.     }
84.
85.     // 启动DMA传输
86.     dma_async_issue_pending(mem_dma->ch);
87.     // 等待dma传输完毕，超时时间设置为1s，
88.     // tx下的dsg都处理完后，回调函数被执行，在回调函数中会设置completion， 然后该函数回
89.     ret = wait_for_completion_timeout(&mem_dma->completion, msecs_to_jiffies
90.     if (!ret) {
91.         dev_err(dev, "%s: timeout !!\n", __func__);
92.         ret = -EIO;
```

```
 93.         goto err5;
 94.     }
 95.
 96.     dev_info(dev, "we get: %s\n", mem_dma->dst_virt);
 97.     return len;
 98.
 99. err5:
100.     return ret;
101. }
102.
103. static const struct file_operations mem_dma_fops = {
104.     .owner      = THIS_MODULE,
105.     .open       = mem_dma_open,
106.     .write      = mem_dma_write,
107. };
108.
109. static struct miscdevice mem_dma_miscdev = {
110.     .minor  = MISC_DYNAMIC_MINOR,
111.     .name   = "mem_dma_test",
112.     .fops   = &mem_dma_fops,
113. };
114.
115. static int memory_dma_probe(struct platform_device *pdev) {
116.     struct device *dev = &pdev->dev;
117.     struct dma_chan *ch;
118.     struct dma_slave_config *cfg;
119.     int ret = 0;
120.     dma_addr_t *src, *dst;
121.     void *src_virt, *dst_virt;
122.     dma_cap_mask_t mask;
123.
124.     dev_info(dev, "%s enter.\n", __func__);
125.
126.     if (!dev->of_node) {
127.         dev_err(dev, "no platform data.\n");
128.         return -EINVAL;
129.     }
130.
131.     mem_dma = devm_kzalloc(dev, sizeof(mem_dma_t), GFP_KERNEL);
132.     if (!mem_dma) {
133.         dev_err(dev, "can not alloc memory for mem_dma\n");
134.         return -ENOMEM;
135.     }
136.     mem_dma->dev = dev;
137.
138.     // 下面是申请用于进行MEM2MEM的虚拟DMA通道
139.     // 对于S3C2440来说，由于所有的物理DMA都支持MEM2MEM，所以只要
140.     // 找到具备DMA_MEMCPY的dma device下的任意一个空闲的虚拟DMA通道即可
141.     dma_cap_zero(mask);
142.     dma_cap_set(DMA_MEMCPY, mask);
143.     ch = dma_request_channel(mask, NULL, NULL);
144.     if (IS_ERR(ch)) {
145.         ret = PTR_ERR(ch);
146.         dev_err(dev, "%s: dma_request_channel failed: %d\n", __func__, ret)
```

```
147.         goto err1;
148.     }
149.     mem_dma->ch = ch;
150.     mem_dma->dma_dev = ch->device;
151.
152.     src = &mem_dma->src;
153.     // 申请一块大小为BUFSIZE的连续的nocache的物理地址作为DMA传输的源地址
154.     // 物理地址存放在*src中，而对应的虚拟地址就是返回值
155.     src_virt = dma_zalloc_coherent(dev, BUFSIZE, src, GFP_KERNEL);
156.     if (!src_virt) {
157.         dev_err(dev, "%s: alloc src failed.\n", __func__);
158.         ret = -ENOMEM;
159.         goto err2;
160.     }
161.     mem_dma->src_virt = src_virt;
162.
163.     dst = &mem_dma->dst;
164.     // 同上
165.     dst_virt = dma_zalloc_coherent(dev, BUFSIZE, dst, GFP_KERNEL);
166.     if (!dst_virt) {
167.         dev_err(dev, "%s: alloc dst failed.\n", __func__);
168.         ret = -ENOMEM;
169.         goto err3;
170.     }
171.     mem_dma->dst_virt = dst_virt;
172.
173.     cfg = devm_kzalloc(dev, sizeof(*cfg), GFP_KERNEL);
174.     if (!cfg) {
175.         dev_err(dev, "%s: alloc cfg failed.\n", __func__);
176.         ret = -ENOMEM;
177.         goto err4;
178.     }
179.     mem_dma->cfg = cfg;
180.
181.     // 配置
182.     cfg->direction = DMA_MEM_TO_MEM;  // 方向
183.     cfg->src_addr_width = DMA_SLAVE_BUSWIDTH_1_BYTE; // 一个read读1个字节
184.     cfg->dst_addr_width = DMA_SLAVE_BUSWIDTH_1_BYTE; // 一个write写1个字节
185.     cfg->src_addr = *src; // 设置源物理地址
186.     cfg->dst_addr = *dst; // 设置目的物理地址
187.     cfg->src_maxburst = 1;  // 每次DMA传输read/write各一次
188.     dmaengine_slave_config(ch, cfg); // 将配置设置给申请到的虚拟DMA通道
189.
190.     init_completion(&mem_dma->completion); // 初始化完成变量
191.     ret = misc_register(&mem_dma_miscdev);
192.     if (ret < 0) {
193.         pr_err("failed to register mem_dma device\n");
194.         goto err4;;
195.     }
196.
197.     return 0;
198.
199. err4:
200.     dma_free_coherent(dev, BUFSIZE, dst_virt, *dst);
```

```
201.  err3:
202.      dma_free_coherent(dev, BUFSIZE, src_virt, *src);
203.  err2:
204.      dma_release_channel(ch);
205.  err1:
206.      return ret;
207.
208.  }
209.
210.  static int memory_dma_remove(struct platform_device *pdev) {
211.      struct device *dev = &pdev->dev;
212.
213.      dev_info(dev, "%s enter.\n", __func__);
214.
215.      dmaengine_terminate_async(mem_dma->ch); // 终止该虚拟DMA通道下的所有传输
216.      misc_deregister(&mem_dma_miscdev);
217.      dma_release_channel(mem_dma->ch); // 释放虚拟DMA通道
218.      dma_free_coherent(dev, BUFSIZE, mem_dma->dst_virt, mem_dma->dst); // 释放
219.      dma_free_coherent(dev, BUFSIZE, mem_dma->src_virt, mem_dma->src); // 释放
220.
221.      return 0;
222.  }
223.
224.  static const struct of_device_id memory_dma_dt_ids[] = {
225.      { .compatible = "tq2440,memory_dma_v4", },
226.      {},
227.  };
228.
229.  MODULE_DEVICE_TABLE(of, memory_dma_dt_ids);
230.
231.  static struct platform_driver memory_dma_driver = {
232.      .driver         = {
233.          .name    = "memory_dma",
234.          .of_match_table    = of_match_ptr(memory_dma_dt_ids),
235.      },
236.      .probe         = memory_dma_probe,
237.      .remove         = memory_dma_remove,
238.  };
239.
240.  static int __init memory_dma_init(void)
241.  {
242.      int ret;
243.
244.      ret = platform_driver_register(&memory_dma_driver);
245.      if (ret)
246.          printk(KERN_ERR "memory_dma: probe failed: %d\n", ret);
247.
248.      return ret;
249.  }
250.  module_init(memory_dma_init);
251.
252.  static void __exit memory_dma_exit(void)
253.  {
254.      platform_driver_unregister(&memory_dma_driver);
```

```
255.  }
256.  module_exit(memory_dma_exit);
257.
258.  MODULE_LICENSE("GPL");
```

2、详细分析

上面的驱动实现的一个简单的MEM2MEM的DMA传输： 分配了两块内存，起始地址分别是src和dst，同时在/dev下创建了一个名为mem_dma_test的设备节点，然后可以向给节点echo一些内容。驱动会将用户echo下来的字符串拷贝到src中，然后通过DMA传输的方式再将其拷贝到dst中，最后将dst中的内容打印出来，看看跟用户写入的内容是否一致。下面我们分析一下：

- 申请虚拟DMA通道

上面调用的函数是dma_request_channel函数，而且仅仅设置了mask这个过滤条件。因为2440的四个物理DMA都支持MEM2MEM的传输，所以要求简单，只要从具备DMA_MEMCPY能力的dma device中找到一条空闲的虚拟DMA通道即可。如果需要实现的是MEM2DEV或在DEV2MEM之间的DMA传输的话，就需要使用dma_request_chan这个接口了。下面一一分析

dma_request_channel:

```
1.  struct dma_chan *__dma_request_channel(const dma_cap_mask_t *mask,
2.                          dma_filter_fn fn, void *fn_param)
3.  {
4.      struct dma_device *device, *_d;
5.      struct dma_chan *chan = NULL;
6.
7.      list_for_each_entry_safe(device, _d, &dma_device_list, global_node) {
8.          chan = find_candidate(device, mask, fn, fn_param);
9.          if (!IS_ERR(chan))
10.             break;
11.
12.         chan = NULL;
13.     }
14.
15.     return chan;
16. }
```

这个函数会遍历dma_device_list中的每一个dma device，然后调用find_candidate从中找到符合条件的虚拟DMA通道:

```
1.  static struct dma_chan *find_candidate(struct dma_device *device,
2.                          const dma_cap_mask_t *mask,
3.                          dma_filter_fn fn, void *fn_param)
4.  {
5.      struct dma_chan *chan = private_candidate(mask, device, fn, fn_param);
6.      int err;
```

```
7.
8.     if (chan) {
9.         /* Found a suitable channel, try to grab, prep, and return it.
10.         * We first set DMA_PRIVATE to disable balance_ref_count as this
11.         * channel will not be published in the general-purpose
12.         * allocator
13.         */
14.        dma_cap_set(DMA_PRIVATE, device->cap_mask);
15.        device->privatecnt++;
16.        err = dma_chan_get(chan);
17.
18.    }
19.
20.    return chan ? chan : ERR_PTR(-EPROBE_DEFER);
21. }
```

这里主要涉及两个函数： private_candidate和dma_chan_get

private_candidate：从dma device中找到符合条件的虚拟DMA通道

```
1.  static struct dma_chan *private_candidate(const dma_cap_mask_t *mask,
2.                      struct dma_device *dev,
3.                      dma_filter_fn fn, void *fn_param)
4.  {
5.      struct dma_chan *chan;
6.
7.      if (mask && !__dma_device_satisfies_mask(dev, mask)) {
8.          dev_dbg(dev->dev, "%s: wrong capabilities\n", __func__);
9.          return NULL;
10.     }
11.     /* devices with multiple channels need special handling as we need to
12.      * ensure that all channels are either private or public.
13.      */
14.     if (dev->chancnt > 1 && !dma_has_cap(DMA_PRIVATE, dev->cap_mask))
15.         list_for_each_entry(chan, &dev->channels, device_node) {
16.             /* some channels are already publicly allocated */
17.             if (chan->client_count)
18.                 return NULL;
19.         }
20.
21.     list_for_each_entry(chan, &dev->channels, device_node) {
22.         if (chan->client_count) {
23.             dev_dbg(dev->dev, "%s: %s busy\n",
24.                  __func__, dma_chan_name(chan));
25.             continue;
26.         }
27.         if (fn && !fn(chan, fn_param)) {
28.             dev_dbg(dev->dev, "%s: %s filter said false\n",
29.                  __func__, dma_chan_name(chan));
30.             continue;
31.         }
32.         return chan;
33.     }
34.
```

```
35.     return NULL;
36. }
```

第7行会首先检查传入的dma device是否具备mask所指定的能力，如果具备的话返回true
第14行由于我们注册的memcpy dma device具备DMA_PRIVATE，所以if条件不成立
第21行的for循环会遍历该dma deivce下的虚拟DMA通道（在DMA控制器驱动被注册的时候调用vchan_init添加的），直到找到一条符合条件的。需要符合两个条件：第一是该虚拟DMA通道的client_count为0，即该虚拟DMA通道当前没有被占用，第二是如果设置了过滤函数的话，过滤函数需要返回true， 由于这里我们没有传递fn参数，所以只要直到一条没有被占用的虚拟DMA通道即可
回到find_candidate，在找到满足条件的DMA虚拟通道后，会调用dma_chan_get增加该虚拟DMA通道的引用计数：

```
1.  static int dma_chan_get(struct dma_chan *chan)
2.  {
3.      struct module *owner = dma_chan_to_owner(chan);
4.      int ret;
5.
6.      /* The channel is already in use, update client count */
7.      if (chan->client_count) {
8.          __module_get(owner);
9.          goto out;
10.     }
11.
12.     if (!try_module_get(owner))
13.         return -ENODEV;
14.
15.     /* allocate upon first client reference */
16.     if (chan->device->device_alloc_chan_resources) {
17.         ret = chan->device->device_alloc_chan_resources(chan);
18.         if (ret < 0)
19.             goto err_out;
20.     }
21.
22.     if (!dma_has_cap(DMA_PRIVATE, chan->device->cap_mask))
23.         balance_ref_count(chan);
24.
25. out:
26.     chan->client_count++;
27.     return 0;
28.
29. err_out:
30.     module_put(owner);
31.     return ret;
32. }
```

第7行对于一条空闲的虚拟DMA通道，其client_count是0，所以if条件不成立
第12行增加所属模块的引用计数

第16行，如果该虚拟DMA通道所属的dma device设置了device_alloc_chan_resources，那么就会调用该函数为虚拟DMA通道分配资源。2440的dma控制器启动没有设置
第26行将该虚拟DMA通道的client_count加一，这样就不会被其他驱动再次request了
上面分析完了dma_request_channel，接下来再分析一下应用更多的dma_request_chan函数。
dma_request_chan的第二个参数是需要申请的虚拟DMA通道的name，会通过解析设备树获得名为name的虚拟DMA通道，我们结合设备树分析一下：

```
1.      dma: s3c2410-dma@4B000000 {
2.          compatible = "s3c2440-dma";
3.          reg = <0x4B000000 0x1000>;
4.          interrupts = <0 0 17 3>, <0 0 18 3>,
5.                  <0 0 19 3>, <0 0 20 3>;
6.          #dma-cells = <1>;
7.      };
8.
9.      s3c2440_iis@55000000 {
10.         compatible = "s3c24xx-iis";
11.         reg = <0x55000000 0x100>;
12.         clocks = <&clocks PCLK_I2S>;
13.         clock-names = "iis";
14.         pinctrl-names = "default";
15.         pinctrl-0 = <&s3c2440_iis_pinctrl>;
16.         dmas = <&dma DMACH_I2S_IN>, <&dma DMACH_I2S_OUT>;
17.         dma-names = "rx", "tx";
18.     };
```

上面我们以IIS的设备树为例说一下。
第1行是DMA控制器的设备树，特点是具备一个"#dma-cells"属性，这里属性值为1，表示描述一个DMA资源，需要一个传递1个参数。
第9行是I2S的设备树，它所引用的DMA资源在第16和17行进行了描述，在I2S的驱动中如果想获得"rx"这个虚拟DMA资源，可以使用下面的接口：

```
1.  dma_request_chan(dev, "rx")
```

定义如下：

```
1.  struct dma_chan *dma_request_chan(struct device *dev, const char *name)
2.  {
3.      struct dma_device *d, *_d;
4.      struct dma_chan *chan = NULL;
5.
6.      /* If device-tree is present get slave info from here */
7.      if (dev->of_node)
8.          chan = of_dma_request_slave_channel(dev->of_node, name);
9.
10.     /* If device was enumerated by ACPI get slave info from here */
```

```
11.     if (has_acpi_companion(dev) && !chan)
12.         chan = acpi_dma_request_slave_chan_by_name(dev, name);
13. 
14.     if (chan) {
15.         /* Valid channel found or requester need to be deferred */
16.         if (!IS_ERR(chan) || PTR_ERR(chan) == -EPROBE_DEFER)
17.             return chan;
18.     }
19. 
20.     /* Try to find the channel via the DMA filter map(s) */
21.     mutex_lock(&dma_list_mutex);
22.     list_for_each_entry_safe(d, _d, &dma_device_list, global_node) {
23.         dma_cap_mask_t mask;
24.         const struct dma_slave_map *map = dma_filter_match(d, name, dev);
25. 
26.         if (!map)
27.             continue;
28. 
29.         dma_cap_zero(mask);
30.         dma_cap_set(DMA_SLAVE, mask);
31. 
32.         chan = find_candidate(d, &mask, d->filter.fn, map->param);
33.         if (!IS_ERR(chan))
34.             break;
35.     }
36.     mutex_unlock(&dma_list_mutex);
37. 
38.     return chan ? chan : ERR_PTR(-EPROBE_DEFER);
39. }
```

由于采用了设备树，所以第7行的if条件成立，接下来会调用of_dma_request_slave_channel获得name指定的虚拟DMA资源
如果申请成功的话，最终会在第17行返回。
第22行的循环是另一种获取DMA资源的方式，如果没有使用设备树的话或者使用设备树的方式申请失败（-EPROBE_DEFER除外）的时候，才会用到。
这里我们只分析使用设备树分配成功的情况：

```
 1. struct dma_chan *of_dma_request_slave_channel(struct device_node *np,
 2.                                               const char *name)
 3. {
 4.     struct of_phandle_args  dma_spec;
 5.     struct of_dma       *ofdma;
 6.     struct dma_chan     *chan;
 7.     int         count, i, start;
 8.     int         ret_no_channel = -ENODEV;
 9.     static atomic_t     last_index;
10. 
11.     /* Silently fail if there is not even the "dmas" property */
12.     if (!of_find_property(np, "dmas", NULL))
13.         return ERR_PTR(-ENODEV);
14. 
15.     count = of_property_count_strings(np, "dma-names");
```

```
16.
17.     start = atomic_inc_return(&last_index);
18.     for (i = 0; i < count; i++) {
19.         if (of_dma_match_channel(np, name,
20.                     (i + start) % count,
21.                     &dma_spec))
22.             continue;
23.
24.         mutex_lock(&of_dma_lock);
25.         ofdma = of_dma_find_controller(&dma_spec);
26.
27.         if (ofdma) {
28.             chan = ofdma->of_dma_xlate(&dma_spec, ofdma);
29.         } else {
30.             ret_no_channel = -EPROBE_DEFER;
31.             chan = NULL;
32.         }
33.
34.         mutex_unlock(&of_dma_lock);
35.
36.         of_node_put(dma_spec.np);
37.
38.         if (chan)
39.             return chan;
40.     }
41.
42.     return ERR_PTR(ret_no_channel);
43. }
```

第12行检查是否具备dmas属性
第15行用于统计"dma-names"属性中字符串的个数，也就是引用的DMA资源的个数
第17行的start是为了加快获取DMA资源速度，因为如果设备树中引用了多个DMA资源的话，在设备驱动中获取DMA资源一般是按照dma-names属性中的字符串的先后顺序进行的，下一个DMA资源索引只需从前一个DMA资源索引开始， 从而避免了无用的循环。
第19行的of_dma_match_channel会根据传入的index获取dma-names属性值的第index项，如果跟传入的name匹配，那么就解析dmas属性值的第index项，结果存放到dma_spec中
第25行根据dma_spec找到引用的ofdma，也就是在DMA控制器驱动里调用of_dma_controller_register注册的哪个
第28行调用of_dma_xlate函数，在2440的DMA控制器驱动中设置的是of_dma_simple_xlate：

```
1.  struct dma_chan *of_dma_simple_xlate(struct of_phandle_args *dma_spec,
2.                          struct of_dma *ofdma)
3.  {
4.      int count = dma_spec->args_count;
5.      struct of_dma_filter_info *info = ofdma->of_dma_data;
6.
7.      if (!info || !info->filter_fn)
```

```
8.          return NULL;
9.
10.     if (count != 1)
11.         return NULL;
12.
13.     return dma_request_channel(info->dma_cap, info->filter_fn,
14.             &dma_spec->args[0]);
15. }
```

第5行的of_dma_filter_info就是s3c24xx_dma_info
第7行检查是否设置了filter_fn，对于2440，设置的是s3c24xx_dma_filter
第8行检查传递的参数个数是否是1
第13行调用dma_request_channel，这个函数前面分析了，只是传递的参数不同：dma_cap设置的是s3cdma->slave.cap_mask，也就是DMA_SLAVE|DMA_CYCLIC|DMA_PRIVATE， filer_fn对应的是s3c24xx_dma_filter， 对于"rx"，对应的args[0]是DMACH_I2S_IN，也就是9.
这里分析一下s3c24xx_dma_filter：

```
1.  bool s3c24xx_dma_filter(struct dma_chan *chan, void *param)
2.  {
3.      struct s3c24xx_dma_chan *s3cchan;
4.      int ch = *(uintptr_t *)param;
5.
6.      if (chan->device->dev->driver != &s3c24xx_dma_driver.driver)
7.          return false;
8.
9.      s3cchan = to_s3c24xx_dma_chan(chan);
10.
11.     return s3cchan->id == ch;
12. }
```

第1行的param就是&args[0]，那么*param就是9了
第11行比较传入的虚拟DMA通道的id是否是9，如果相等，返回true， 表示该虚拟DMA通道符合条件， 否则返回false，那么dma_request_channel会继续需要符合条件虚拟DMA通道。

- 配置DMA channel的参数

调用的函数是dmaengine_slave_config

```
1.  static inline int dmaengine_slave_config(struct dma_chan *chan,
2.                          struct dma_slave_config *config)
3.  {
4.      if (chan->device->device_config)
5.          return chan->device->device_config(chan, config);
6.
7.      return -ENOSYS;
8.  }
```

这个函数间接调用了dma device的device_config函数，对于2440就是

s3c24xx_dma_set_runtime_config。

```
1.  static int s3c24xx_dma_set_runtime_config(struct dma_chan *chan,
2.                   struct dma_slave_config *config)
3.  {
4.      struct s3c24xx_dma_chan *s3cchan = to_s3c24xx_dma_chan(chan);
5.      unsigned long flags;
6.      int ret = 0;
7.
8.      /* Reject definitely invalid configurations */
9.      if (config->src_addr_width == DMA_SLAVE_BUSWIDTH_8_BYTES ||
10.         config->dst_addr_width == DMA_SLAVE_BUSWIDTH_8_BYTES)
11.         return -EINVAL;
12.
13.     spin_lock_irqsave(&s3cchan->vc.lock, flags);
14.
15.     if (!s3cchan->slave) {
16.         ret = -EINVAL;
17.         goto out;
18.     }
19.
20.     s3cchan->cfg = *config;
21.
22. out:
23.     spin_unlock_irqrestore(&s3cchan->vc.lock, flags);
24.     return ret;
25. }
```

第9行检查data size，因为2440的DMA只支持1byte、2bytes以及4bytes的data size，不支持8tyes
第15行如果该虚拟DMA属于memcpy这个dma device的话，其slave为0，那么直接报错返回，否则的话，将传入的config赋值给虚拟DMA通道的cfg成员，注：这里仅仅是赋值，并没有设置到硬件中，将来在给该虚拟DMA通道分配到物理DMA，在启动物理DMA之前会根据cfg配置硬件寄存器。而且对于MEM2MEM这种传输，从上面的代码看，不需要调用dmaengine_slave_config

- 获取一个用于识别本次传输（transaction）的描述符（descriptor）

这里调用的函数是device_prep_dma_memcpy， 入参分别是：虚拟DMA通道，目的物理地址，源物理地址，长度以及flags
2440上对应的函数是： s3c24xx_dma_prep_memcpy

```
1.  static struct dma_async_tx_descriptor *s3c24xx_dma_prep_memcpy(
2.          struct dma_chan *chan, dma_addr_t dest, dma_addr_t src,
3.          size_t len, unsigned long flags)
4.  {
5.      struct s3c24xx_dma_chan *s3cchan = to_s3c24xx_dma_chan(chan);
6.      struct s3c24xx_dma_engine *s3cdma = s3cchan->host;
7.      struct s3c24xx_txd *txd;
8.      struct s3c24xx_sg *dsg;
9.      int src_mod, dest_mod;
```

```
10.
11.     if ((len & S3C24XX_DCON_TC_MASK) != len) {
12.         dev_err(&s3cdma->pdev->dev, "memcpy size %zu to large\n", len);
13.         return NULL;
14.     }
15.
16.     txd = s3c24xx_dma_get_txd();
17.
18.     dsg = kzalloc(sizeof(*dsg), GFP_NOWAIT);
19.
20.     list_add_tail(&dsg->node, &txd->dsg_list);
21.
22.     dsg->src_addr = src;
23.     dsg->dst_addr = dest;
24.     dsg->len = len;
25.
26.     /*
27.      * Determine a suitable transfer width.
28.      * The DMA controller cannot fetch/store information which is not
29.      * naturally aligned on the bus, i.e., a 4 byte fetch must start at
30.      * an address divisible by 4 - more generally addr % width must be 0.
31.      */
32.     src_mod = src % 4;
33.     dest_mod = dest % 4;
34.     switch (len % 4) {
35.     case 0:
36.         txd->width = (src_mod == 0 && dest_mod == 0) ? 4 : 1;
37.         break;
38.     case 2:
39.         txd->width = ((src_mod == 2 || src_mod == 0) &&
40.                   (dest_mod == 2 || dest_mod == 0)) ? 2 : 1;
41.         break;
42.     default:
43.         txd->width = 1;
44.         break;
45.     }
46.
47.     txd->disrcc = S3C24XX_DISRCC_LOC_AHB | S3C24XX_DISRCC_INC_INCREMENT;
48.     txd->didstc = S3C24XX_DIDSTC_LOC_AHB | S3C24XX_DIDSTC_INC_INCREMENT;
49.     txd->dcon |= S3C24XX_DCON_DEMAND | S3C24XX_DCON_SYNC_HCLK |
50.              S3C24XX_DCON_SERV_WHOLE;
51.
52.     return vchan_tx_prep(&s3cchan->vc, &txd->vd, flags);
53. }
```

第11行检查len的大小是否超出限制，对于2440，DMA传输次数是由DCON[19:0]控制的，这里是DMA传输次数， 次数*单次数据量 就是该txd需要DMA传输的总数据。

第16行分配一个txd，然后初始化txd->dsg_list链表， 然后设置txd->dcon为S3C24XX_DCON_INT | S3C24XX_DCON_NORELOAD，即每次TC减到0，都会触发中断，而且不要auto-reload，即TC减到0便停止DMA

第18到24分配一个dsg，然后将src、dst以及len存放到该dsg中，最后将该dsg假如

txd->dsg_list中。这里的len需要注意，如果datasize是1byte的话，就不需要修改了，否则len还需要调整
第32到45行，根据src和dst的地址特点以及len的大小设置txd->width，即data size， 也就是一个read或write操作几个字节的数据
第47行设置disrcc，表示源在AHB上，地址递增
第48行设置didstc，表示目的在AHB上，地址递增
第49行修改dcon，设置为demand模式、跟HCLK同步，Whole Service模式
第52行调用vchan_tx_prep，对传输描述符进行prepare：

```
1.  static inline struct dma_async_tx_descriptor *vchan_tx_prep(struct virt_dma_
2.      struct virt_dma_desc *vd, unsigned long tx_flags)
3.  {
4.      unsigned long flags;
5.
6.      dma_async_tx_descriptor_init(&vd->tx, &vc->chan);
7.      vd->tx.flags = tx_flags;
8.      vd->tx.tx_submit = vchan_tx_submit;
9.      vd->tx.desc_free = vchan_tx_desc_free;
10.
11.     spin_lock_irqsave(&vc->lock, flags);
12.     list_add_tail(&vd->node, &vc->desc_allocated);
13.     spin_unlock_irqrestore(&vc->lock, flags);
14.
15.     return &vd->tx;
16. }
```

第6行将&vc->chan赋值给(&vd->tx)->chan，这样通过txd就可以找到所属的vc
第8行，在调用dmaengine_submit的时候，vchan_tx_submit会被调用， 主要是分配一个新的cookie，同时将txd移入desc_submitted中
第9行，用于将txd从vc的链表中删除，然后调用用户设置的vc->desc_free，释放txd，2440的DMA控制器驱动用s3c24xx_txd对传输描述符进行了封装，所以有额外的数据需要释放，所以需要调用用户自定义的desc_free
第12行将txd添加到desc_allocated链表中
对于MEM2DEV和DEV2MEM应该调用下面的两个函数来得到txd：
dmaengine_prep_slave_sg
使用：DMA a list of scatter gather buffers from/to a peripheral，比如用DMA传输数据的SPI控制器驱动
dmaengine_prep_dma_cyclic
用法：Perform a cyclic DMA operation from/to a peripheral till the operation is explicitly stopped， 比如Audio
dmaengine_prep_slave_sg： 作用： DMA a list of scatter gather buffers from/to a peripheral

```
1.  static inline struct dma_async_tx_descriptor *dmaengine_prep_slave_sg(
```

```
    struct dma_chan *chan, struct scatterlist *sgl, unsigned int sg_len,
    enum dma_transfer_direction dir, unsigned long flags)
{
    if (!chan || !chan->device || !chan->device->device_prep_slave_sg)
        return NULL;

    return chan->device->device_prep_slave_sg(chan, sgl, sg_len,
                        dir, flags, NULL);
}
```

首先进行环境检查，然后调用dma device的device_prep_slave_sg， 对于2440就是s3c24xx_dma_prep_slave_sg：

```
static struct dma_async_tx_descriptor *s3c24xx_dma_prep_slave_sg(
        struct dma_chan *chan, struct scatterlist *sgl,
        unsigned int sg_len, enum dma_transfer_direction direction,
        unsigned long flags, void *context)
{
    struct s3c24xx_dma_chan *s3cchan = to_s3c24xx_dma_chan(chan);
    struct s3c24xx_dma_engine *s3cdma = s3cchan->host;
    const struct s3c24xx_dma_platdata *pdata = s3cdma->pdata;
    struct s3c24xx_dma_channel *cdata = &pdata->channels[s3cchan->id];
    struct s3c24xx_txd *txd;
    struct s3c24xx_sg *dsg;
    struct scatterlist *sg;
    dma_addr_t slave_addr;
    u32 hwcfg = 0;
    int tmp;

    txd = s3c24xx_dma_get_txd();

    if (cdata->handshake)
        txd->dcon |= S3C24XX_DCON_HANDSHAKE;

    switch (cdata->bus) {
    case S3C24XX_DMA_APB:
        txd->dcon |= S3C24XX_DCON_SYNC_PCLK;
        hwcfg |= S3C24XX_DISRCC_LOC_APB;
        break;
    case S3C24XX_DMA_AHB:
        txd->dcon |= S3C24XX_DCON_SYNC_HCLK;
        hwcfg |= S3C24XX_DISRCC_LOC_AHB;
        break;
    }

    /*
     * Always assume our peripheral desintation is a fixed
     * address in memory.
     */
    hwcfg |= S3C24XX_DISRCC_INC_FIXED;

    /*
     * Individual dma operations are requested by the slave,
```

```
41.          * so serve only single atomic operations (S3C24XX_DCON_SERV_SINGLE).
42.          */
43.         txd->dcon |= S3C24XX_DCON_SERV_SINGLE;
44. 
45.         if (direction == DMA_MEM_TO_DEV) {
46.             txd->disrcc = S3C24XX_DISRCC_LOC_AHB |
47.                       S3C24XX_DISRCC_INC_INCREMENT;
48.             txd->didstc = hwcfg;
49.             slave_addr = s3cchan->cfg.dst_addr;
50.             txd->width = s3cchan->cfg.dst_addr_width;
51.         } else if (direction == DMA_DEV_TO_MEM) {
52.             txd->disrcc = hwcfg;
53.             txd->didstc = S3C24XX_DIDSTC_LOC_AHB |
54.                       S3C24XX_DIDSTC_INC_INCREMENT;
55.             slave_addr = s3cchan->cfg.src_addr;
56.             txd->width = s3cchan->cfg.src_addr_width;
57.         } else {
58.             s3c24xx_dma_free_txd(txd);
59.             dev_err(&s3cdma->pdev->dev,
60.                 "direction %d unsupported\n", direction);
61.             return NULL;
62.         }
63. 
64.         for_each_sg(sgl, sg, sg_len, tmp) {
65.             dsg = kzalloc(sizeof(*dsg), GFP_NOWAIT);
66. 
67.             list_add_tail(&dsg->node, &txd->dsg_list);
68. 
69.             dsg->len = sg_dma_len(sg);
70.             if (direction == DMA_MEM_TO_DEV) {
71.                 dsg->src_addr = sg_dma_address(sg);
72.                 dsg->dst_addr = slave_addr;
73.             } else { /* DMA_DEV_TO_MEM */
74.                 dsg->src_addr = slave_addr;
75.                 dsg->dst_addr = sg_dma_address(sg);
76.             }
77.         }
78. 
79.         return vchan_tx_prep(&s3cchan->vc, &txd->vd, flags);
80.     }
```

第9行的&pdata->channels[s3cchan->id]中channels数组定义在arch/arm/mach-s3c24xx/common.c中：

```
1.  static struct s3c24xx_dma_channel s3c2440_dma_channels[] = {
2.      [DMACH_XD0] = { S3C24XX_DMA_AHB, true, S3C24XX_DMA_CHANREQ(0, 0), },
3.      [DMACH_XD1] = { S3C24XX_DMA_AHB, true, S3C24XX_DMA_CHANREQ(0, 1), },
4.      [DMACH_SDI] = { S3C24XX_DMA_APB, false, S3C24XX_DMA_CHANREQ(2, 0) |
5.                          S3C24XX_DMA_CHANREQ(6, 1) |
6.                          S3C24XX_DMA_CHANREQ(2, 2) |
7.                          S3C24XX_DMA_CHANREQ(1, 3),
8.      },
9.      [DMACH_SPI0] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(3, 1), },
```

```
10.      [DMACH_SPI1] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(2, 3), },
11.      [DMACH_UART0] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(1, 0), },
12.      [DMACH_UART1] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(1, 1), },
13.      [DMACH_UART2] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(0, 3), },
14.      [DMACH_TIMER] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(3, 0) |
15.                              S3C24XX_DMA_CHANREQ(3, 2) |
16.                              S3C24XX_DMA_CHANREQ(3, 3),
17.      },
18.      [DMACH_I2S_IN] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(2, 1) |
19.                              S3C24XX_DMA_CHANREQ(1, 2),
20.      },
21.      [DMACH_I2S_OUT] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(5, 0) |
22.                               S3C24XX_DMA_CHANREQ(0, 2),
23.      },
24.      [DMACH_PCM_IN] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(6, 0) |
25.                              S3C24XX_DMA_CHANREQ(5, 2),
26.      },
27.      [DMACH_PCM_OUT] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(5, 1) |
28.                              S3C24XX_DMA_CHANREQ(6, 3),
29.      },
30.      [DMACH_MIC_IN] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(6, 2) |
31.                              S3C24XX_DMA_CHANREQ(5, 3),
32.      },
33.      [DMACH_USB_EP1] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(4, 0), }
34.      [DMACH_USB_EP2] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(4, 1), }
35.      [DMACH_USB_EP3] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(4, 2), }
36.      [DMACH_USB_EP4] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(4, 3), }
37.  };
```

这个数组记录了每一个虚拟DMA通道DEV一端的配置，比如挂载哪个总线，使用什么模式，可以使用哪个物理DMA资源，比如：

```
1.       [DMACH_I2S_IN] = { S3C24XX_DMA_APB, true, S3C24XX_DMA_CHANREQ(2, 1) |
2.                               S3C24XX_DMA_CHANREQ(1, 2),
3.       },
```

表示从I2S控制器读数据到内存的DMA： dev在APB上，使用握手模式，可以在CH-1物理DMA通道发起request2或在CH-2物理DMA上发起request1，这个跟2440的手册是对应的：

Table 8-1. DMA Request Sources for Each Channel

	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

回到s3c24xx_dma_prep_slave_sg继续分析：
第19行，设置demand或在handshake模式
第22到30行中的hwcfg将来会设置给dev那一端

第37行，dev这一端的地址固定
第43行，Single Service模式，这个模式下，每进行完一次DMA传输，状态机切到状态1，然后等待外设的再次DMA请求
第45到60，根据dirction设置DMA参数。

对于MEM2DEV，MEM是src，DEV是dst，src在AHB，地址递增， hwcfg赋值给didstc，目的地址以及data size由cfg给出。为什么这里的源地址不从cfg中获得呢？因为此时源地址在内存中，每一个scatterlist的源地址都不相同

对于DEV2MEM，DEV是src，MEM是dst，dst在AHB中，地址递增，hwcfg赋值给disrcc，源地址和data size有cfg给出，暂时不设置目的地址，理由同上
第64到77行，将scatterlist中的每一项都转换为一个dsg，挂到txd->dsg_list中

对于MEM2DEV，dsg的源地址由sg_dma_address(sg)给出

对于DEV2MEM，dsg的目的地址由sg_dma_address(sg)给出

下面分析dmaengine_prep_dma_cyclic， 这个函数主要用于Audio，将来在分析声卡驱动的时候就会用到。
的

```
1.  static inline struct dma_async_tx_descriptor *dmaengine_prep_dma_cyclic(
2.          struct dma_chan *chan, dma_addr_t buf_addr, size_t buf_len,
3.          size_t period_len, enum dma_transfer_direction dir,
4.          unsigned long flags)
5.  {
6.      if (!chan || !chan->device || !chan->device->device_prep_dma_cyclic)
7.          return NULL;
8.
9.      return chan->device->device_prep_dma_cyclic(chan, buf_addr, buf_len,
10.                     period_len, dir, flags);
11. }
```

对于2440，调用的实际是s3c24xx_dma_prep_dma_cyclic：

```
1.  static struct dma_async_tx_descriptor *s3c24xx_dma_prep_dma_cyclic(
2.      struct dma_chan *chan, dma_addr_t addr, size_t size, size_t period,
3.      enum dma_transfer_direction direction, unsigned long flags)
4.  {
5.      struct s3c24xx_dma_chan *s3cchan = to_s3c24xx_dma_chan(chan);
6.      struct s3c24xx_dma_engine *s3cdma = s3cchan->host;
7.      const struct s3c24xx_dma_platdata *pdata = s3cdma->pdata;
8.      struct s3c24xx_dma_channel *cdata = &pdata->channels[s3cchan->id];
9.      struct s3c24xx_txd *txd;
10.     struct s3c24xx_sg *dsg;
11.     unsigned sg_len;
12.     dma_addr_t slave_addr;
13.     u32 hwcfg = 0;
14.     int i;
15.
16.     if (!is_slave_direction(direction)) {
```

```
17.             dev_err(&s3cdma->pdev->dev,
18.                 "direction %d unsupported\n", direction);
19.             return NULL;
20.         }
21.
22.         txd = s3c24xx_dma_get_txd();
23.
24.         txd->cyclic = 1;
25.
26.         if (cdata->handshake)
27.             txd->dcon |= S3C24XX_DCON_HANDSHAKE;
28.
29.         switch (cdata->bus) {
30.         case S3C24XX_DMA_APB:
31.             txd->dcon |= S3C24XX_DCON_SYNC_PCLK;
32.             hwcfg |= S3C24XX_DISRCC_LOC_APB;
33.             break;
34.         case S3C24XX_DMA_AHB:
35.             txd->dcon |= S3C24XX_DCON_SYNC_HCLK;
36.             hwcfg |= S3C24XX_DISRCC_LOC_AHB;
37.             break;
38.         }
39.
40.         /*
41.          * Always assume our peripheral desintation is a fixed
42.          * address in memory.
43.          */
44.         hwcfg |= S3C24XX_DISRCC_INC_FIXED;
45.
46.         /*
47.          * Individual dma operations are requested by the slave,
48.          * so serve only single atomic operations (S3C24XX_DCON_SERV_SINGLE).
49.          */
50.         txd->dcon |= S3C24XX_DCON_SERV_SINGLE;
51.
52.         if (direction == DMA_MEM_TO_DEV) {
53.             txd->disrcc = S3C24XX_DISRCC_LOC_AHB |
54.                       S3C24XX_DISRCC_INC_INCREMENT;
55.             txd->didstc = hwcfg;
56.             slave_addr = s3cchan->cfg.dst_addr;
57.             txd->width = s3cchan->cfg.dst_addr_width;
58.         } else {
59.             txd->disrcc = hwcfg;
60.             txd->didstc = S3C24XX_DIDSTC_LOC_AHB |
61.                       S3C24XX_DIDSTC_INC_INCREMENT;
62.             slave_addr = s3cchan->cfg.src_addr;
63.             txd->width = s3cchan->cfg.src_addr_width;
64.         }
65.
66.         sg_len = size / period;
67.
68.         for (i = 0; i < sg_len; i++) {
69.             dsg = kzalloc(sizeof(*dsg), GFP_NOWAIT);
70.
```

```
71.         list_add_tail(&dsg->node, &txd->dsg_list);
72.
73.         dsg->len = period;
74.         /* Check last period length */
75.         if (i == sg_len - 1)
76.             dsg->len = size - period * i;
77.         if (direction == DMA_MEM_TO_DEV) {
78.             dsg->src_addr = addr + period * i;
79.             dsg->dst_addr = slave_addr;
80.         } else { /* DMA_DEV_TO_MEM */
81.             dsg->src_addr = slave_addr;
82.             dsg->dst_addr = addr + period * i;
83.         }
84.     }
85.
86.     return vchan_tx_prep(&s3cchan->vc, &txd->vd, flags);
87. }
```

第16行检查direction，这个函数只支持MEM2DEV和DEV2MEM
这个函数有一个名为period的参数，表示每个dsg的长度，也就是DMA传输多少数据中断一次， 而这个函数的入参中的addr却是一块连续的物理地址，第68行到84行将这块连续的地址空间以period为单位长度，划分为sg_len段，每一段都封装为一个dsg

- 将本次传输（transaction）提交给dma engine

调用的是dmaengine_submit:

```
1.  static inline dma_cookie_t dmaengine_submit(struct dma_async_tx_descriptor
2.  {
3.      return desc->tx_submit(desc);
4.  }
```

在调用vchan_tx_prep时，tx_submit被设置为vchan_tx_submit

```
1.  dma_cookie_t vchan_tx_submit(struct dma_async_tx_descriptor *tx)
2.  {
3.      struct virt_dma_chan *vc = to_virt_chan(tx->chan);
4.      struct virt_dma_desc *vd = to_virt_desc(tx);
5.      unsigned long flags;
6.      dma_cookie_t cookie;
7.
8.      spin_lock_irqsave(&vc->lock, flags);
9.      cookie = dma_cookie_assign(tx);
10.
11.     list_move_tail(&vd->node, &vc->desc_submitted);
12.     spin_unlock_irqrestore(&vc->lock, flags);
13.
14.     dev_dbg(vc->chan.device->dev, "vchan %p: txd %p[%x]: submitted\n",
15.         vc, vd, cookie);
16.
17.     return cookie;
18. }
```

第9行为tx分配一个唯一的cookie，用于唯一标识该tx
第11行将txd移入desc_submitted链表

- 启动DMA

调用的是dma_async_issue_pending

```
1.  static inline void dma_async_issue_pending(struct dma_chan *chan)
2.  {
3.      chan->device->device_issue_pending(chan);
4.  }
```

对于2440，device_issue_pending被设置为s3c24xx_dma_issue_pending

```
1.  static void s3c24xx_dma_issue_pending(struct dma_chan *chan)
2.  {
3.      struct s3c24xx_dma_chan *s3cchan = to_s3c24xx_dma_chan(chan);
4.      unsigned long flags;
5.
6.      spin_lock_irqsave(&s3cchan->vc.lock, flags);
7.      if (vchan_issue_pending(&s3cchan->vc)) {
8.          if (!s3cchan->phy && s3cchan->state != S3C24XX_DMA_CHAN_WAITING)
9.              s3c24xx_dma_phy_alloc_and_start(s3cchan);
10.     }
11.     spin_unlock_irqrestore(&s3cchan->vc.lock, flags);
12. }
```

第7行vchan_issue_pending将desc_submitted中的txd移动到desc_issued链表中，如果desc_issued非空的话返回true，表示有txd需要处理
第8行如果该虚拟DMA通道还没有分配物理DMA并且不是WAITING状态（进入这个状态是因为要申请的物理DMA都被占用，导致物理DMA申请失败，此时虚拟DMA通道进入WAITING，这里过滤WAITING状态的原因是，被占用的物理DMA资源空闲后会自动处理WAITING状态的虚拟DMA通道），那么就调用s3c24xx_dma_phy_alloc_and_start分配物理DMA并启动DMA

```
1.  static void s3c24xx_dma_phy_alloc_and_start(struct s3c24xx_dma_chan *s3cchan
2.  {
3.      struct s3c24xx_dma_engine *s3cdma = s3cchan->host;
4.      struct s3c24xx_dma_phy *phy;
5.
6.      phy = s3c24xx_dma_get_phy(s3cchan);
7.      if (!phy) {
8.          dev_dbg(&s3cdma->pdev->dev, "no physical channel available for xfer
9.              s3cchan->name);
10.         s3cchan->state = S3C24XX_DMA_CHAN_WAITING;
11.         return;
12.     }
13.
14.     dev_dbg(&s3cdma->pdev->dev, "allocated physical channel %d for xfer on %
15.         phy->id, s3cchan->name);
16.
```

```
17.     s3cchan->phy = phy;
18.     s3cchan->state = S3C24XX_DMA_CHAN_RUNNING;
19. 
20.     s3c24xx_dma_start_next_txd(s3cchan);
21. }
```

第6行会为虚拟DMA分配一个物理DMA，如果返回NULL，就将该虚拟DMA设置为WAITING状态，当物理DMA空闲后会自动处理。
下面我们看看是如何分配物理DMA的：

```
 1. static
 2. struct s3c24xx_dma_phy *s3c24xx_dma_get_phy(struct s3c24xx_dma_chan *s3cchan
 3. {
 4.     struct s3c24xx_dma_engine *s3cdma = s3cchan->host;
 5.     const struct s3c24xx_dma_platdata *pdata = s3cdma->pdata;
 6.     struct s3c24xx_dma_channel *cdata;
 7.     struct s3c24xx_dma_phy *phy = NULL;
 8.     unsigned long flags;
 9.     int i;
10.     int ret;
11. 
12.     if (s3cchan->slave)
13.         cdata = &pdata->channels[s3cchan->id];
14. 
15.     for (i = 0; i < s3cdma->pdata->num_phy_channels; i++) {
16.         phy = &s3cdma->phy_chans[i];
17. 
18.         if (!phy->valid)
19.             continue;
20. 
21.         if (!s3c24xx_dma_phy_valid(s3cchan, phy))
22.             continue;
23. 
24.         spin_lock_irqsave(&phy->lock, flags);
25. 
26.         if (!phy->serving) {
27.             phy->serving = s3cchan;
28.             spin_unlock_irqrestore(&phy->lock, flags);
29.             break;
30.         }
31. 
32.         spin_unlock_irqrestore(&phy->lock, flags);
33.     }
34. 
35.     /* No physical channel available, cope with it */
36.     if (i == s3cdma->pdata->num_phy_channels) {
37.         dev_warn(&s3cdma->pdev->dev, "no phy channel available\n");
38.         return NULL;
39.     }
40. 
41. 
42.     return phy;
```

```
43.  }
```

第15行的for循环会从遍历所有的物理DMA，直到找到符合虚拟DMA要求的物理DMA
第18行的if条件不成立
第21行用于判断该物理DMA是否符合虚拟DMA的要求， 如果返回false，就继续比较其他的物理DMA。如果虚拟DMA属于memcpy这个dma device，直接返回true，因为对于MEM2MEM的DMA，所有的物理DMA都支持，否则的话，会看该虚拟DMA是否可以匹配该物理DMA，匹配的话，返回true，如果不匹配的话，返回false。
第26行判断该物理DMA是否空闲，serving指向该物理DMA正在被哪个虚拟DMA使用，如果为空，表示该物理DMA空闲
第36行如果if成立，表示没有找到符合条件的物理DMA，那么就返回NULL
第37行将找到的符合条件的物理DMA返回
回到s3c24xx_dma_phy_alloc_and_start继续分析：
第17行记录找到的物理DMA
第18行将虚拟DMA的状态切到RUNNING状态
第20行启动DMA：

```
1.  static void s3c24xx_dma_start_next_txd(struct s3c24xx_dma_chan *s3cchan)
2.  {
3.      struct s3c24xx_dma_phy *phy = s3cchan->phy;
4.      struct virt_dma_desc *vd = vchan_next_desc(&s3cchan->vc);
5.      struct s3c24xx_txd *txd = to_s3c24xx_txd(&vd->tx);
6.
7.      list_del(&txd->vd.node);
8.
9.      s3cchan->at = txd;
10.
11.     /* Wait for channel inactive */
12.     while (s3c24xx_dma_phy_busy(phy))
13.         cpu_relax();
14.
15.     /* point to the first element of the sg list */
16.     txd->at = txd->dsg_list.next;
17.     s3c24xx_dma_start_next_sg(s3cchan, txd);
18. }
```

第7行将该txd从desc_issued中删除
第9行记录该虚拟DMA下正在处理的txd，从这里知道，一个虚拟DMA下可以挂多个txd
第12行判断物理DMA是否正在有数据传输，如果有数据传输的话，就休息一会
第16行at指向dsg_list中的一个将要被处理的dsg
第17行启动DMA：

```
1.  static void s3c24xx_dma_start_next_sg(struct s3c24xx_dma_chan *s3cchan,
2.                                       struct s3c24xx_txd *txd)
3.  {
4.      struct s3c24xx_dma_engine *s3cdma = s3cchan->host;
```

```
    struct s3c24xx_dma_phy *phy = s3cchan->phy;
    const struct s3c24xx_dma_platdata *pdata = s3cdma->pdata;
    struct s3c24xx_sg *dsg = list_entry(txd->at, struct s3c24xx_sg, node);
    u32 dcon = txd->dcon;
    u32 val;

    /* transfer-size and -count from len and width */
    switch (txd->width) {
    case 1:
        dcon |= S3C24XX_DCON_DSZ_BYTE | dsg->len;
        break;
    case 2:
        dcon |= S3C24XX_DCON_DSZ_HALFWORD | (dsg->len / 2);
        break;
    case 4:
        dcon |= S3C24XX_DCON_DSZ_WORD | (dsg->len / 4);
        break;
    }

    if (s3cchan->slave) {
        struct s3c24xx_dma_channel *cdata =
                    &pdata->channels[s3cchan->id];

        if (s3cdma->sdata->has_reqsel) {
            writel_relaxed((cdata->chansel << 1) |
                           S3C24XX_DMAREQSEL_HW,
                    phy->base + S3C24XX_DMAREQSEL);
        } else {
            int csel = cdata->chansel >> (phy->id *
                           S3C24XX_CHANSEL_WIDTH);

            csel &= S3C24XX_CHANSEL_REQ_MASK;
            dcon |= csel << S3C24XX_DCON_HWSRC_SHIFT;
            dcon |= S3C24XX_DCON_HWTRIG;
        }
    } else {
        if (s3cdma->sdata->has_reqsel)
            writel_relaxed(0, phy->base + S3C24XX_DMAREQSEL);
    }

    writel_relaxed(dsg->src_addr, phy->base + S3C24XX_DISRC);
    writel_relaxed(txd->disrcc, phy->base + S3C24XX_DISRCC);
    writel_relaxed(dsg->dst_addr, phy->base + S3C24XX_DIDST);
    writel_relaxed(txd->didstc, phy->base + S3C24XX_DIDSTC);
    writel_relaxed(dcon, phy->base + S3C24XX_DCON);

    val = readl_relaxed(phy->base + S3C24XX_DMASKTRIG);
    val &= ~S3C24XX_DMASKTRIG_STOP;
    val |= S3C24XX_DMASKTRIG_ON;

    /* trigger the dma operation for memcpy transfers */
    if (!s3cchan->slave)
        val |= S3C24XX_DMASKTRIG_SWTRIG;

```

```
59.     writel(val, phy->base + S3C24XX_DMASKTRIG);
60. }
```

第12到22行根据txd->width计算获得dcon[19:0], 比如如果txd->width为4的话，表示一个read/write操作4个字节，同时将dsg->len/4，表示处理完这个dsg需要进行几次DMA传输
第24到43行是针对MEM2DEV和DEV2MEM设置DMA Request Source和硬件触发，而对MEM2MEM没有意义
第45行设置源物理地址设置到寄存器
第46行将源地址配置参数设置到寄存器
第47行设置目的物理地址到寄存器
第48行将目的地址配置设置到寄存器
第49行将设置DCON寄存器
第51到57行构造写入dmasktrig的值，对于mem2mem使用软件触发
第59行将val写入寄存器，启动DMA

启动DMA后，当DCON[19:0]指定数量的数据传输完毕后，物理DMA的中断就会触发。对于2440，这里的中断处理函数是s3c24xx_dma_irq.

```
1.  static irqreturn_t s3c24xx_dma_irq(int irq, void *data)
2.  {
3.      struct s3c24xx_dma_phy *phy = data;
4.      struct s3c24xx_dma_chan *s3cchan = phy->serving;
5.      struct s3c24xx_txd *txd;
6.
7.      dev_dbg(&phy->host->pdev->dev, "interrupt on channel %d\n", phy->id);
8.
9.      /*
10.      * Interrupts happen to notify the completion of a transfer and the
11.      * channel should have moved into its stop state already on its own.
12.      * Therefore interrupts on channels not bound to a virtual channel
13.      * should never happen. Nevertheless send a terminate command to the
14.      * channel if the unlikely case happens.
15.      */
16.     if (unlikely(!s3cchan)) {
17.         dev_err(&phy->host->pdev->dev, "interrupt on unused channel %d\n",
18.             phy->id);
19.
20.         s3c24xx_dma_terminate_phy(phy);
21.
22.         return IRQ_HANDLED;
23.     }
24.
25.     spin_lock(&s3cchan->vc.lock);
26.     txd = s3cchan->at;
27.     if (txd) {
28.         /* when more sg's are in this txd, start the next one */
29.         if (!list_is_last(txd->at, &txd->dsg_list)) {
```

```
30.             txd->at = txd->at->next;
31.             if (txd->cyclic)
32.                 vchan_cyclic_callback(&txd->vd);
33.             s3c24xx_dma_start_next_sg(s3cchan, txd);
34.         } else if (!txd->cyclic) {
35.             s3cchan->at = NULL;
36.             vchan_cookie_complete(&txd->vd);
37. 
38.             /*
39.              * And start the next descriptor (if any),
40.              * otherwise free this channel.
41.              */
42.             if (vchan_next_desc(&s3cchan->vc))
43.                 s3c24xx_dma_start_next_txd(s3cchan);
44.             else
45.                 s3c24xx_dma_phy_free(s3cchan);
46.         } else {
47.             vchan_cyclic_callback(&txd->vd);
48. 
49.             /* Cyclic: reset at beginning */
50.             txd->at = txd->dsg_list.next;
51.             s3c24xx_dma_start_next_sg(s3cchan, txd);
52.         }
53.     }
54.     spin_unlock(&s3cchan->vc.lock);
55. 
56.     return IRQ_HANDLED;
57. }
```

第26行，at指向正在处理的txd
第29行，txd->at指向刚刚处理完毕的dsg，这里会判断该dsg是不是dsg->list中的最后一个，如果不是的话，继续处理下一个dsg。对于cyclic类型的txd，每处理完一个dsg，都会调用txd的会调函数：

```
1.  static inline void vchan_cyclic_callback(struct virt_dma_desc *vd)
2.  {
3.      struct virt_dma_chan *vc = to_virt_chan(vd->tx.chan);
4. 
5.      vc->cyclic = vd;
6.      tasklet_schedule(&vc->task);
7.  }
```

其中tasklet是在vchan_init时设置的，处理函数是vchan_complete

```
1.  static void vchan_complete(unsigned long arg)
2.  {
3.      struct virt_dma_chan *vc = (struct virt_dma_chan *)arg;
4.      struct virt_dma_desc *vd;
5.      struct dmaengine_desc_callback cb;
6.      LIST_HEAD(head);
7. 
8.      spin_lock_irq(&vc->lock);
```

```
 9.      list_splice_tail_init(&vc->desc_completed, &head);
10.      vd = vc->cyclic;
11.      if (vd) {
12.          vc->cyclic = NULL;
13.          dmaengine_desc_get_callback(&vd->tx, &cb);
14.      } else {
15.          memset(&cb, 0, sizeof(cb));
16.      }
17.      spin_unlock_irq(&vc->lock);
18.
19.      dmaengine_desc_callback_invoke(&cb, NULL);
20.
21.      while (!list_empty(&head)) {
22.          vd = list_first_entry(&head, struct virt_dma_desc, node);
23.          dmaengine_desc_get_callback(&vd->tx, &cb);
24.
25.          list_del(&vd->node);
26.          if (dmaengine_desc_test_reuse(&vd->tx))
27.              list_add(&vd->node, &vc->desc_allocated);
28.          else
29.              vc->desc_free(vd);
30.
31.          dmaengine_desc_callback_invoke(&cb, NULL);
32.      }
33.  }
```

第9行将desc_completed链表中的内容移动到head中
第10行，对于cyclic类型的txd，每处理完一个dsg，都会执行到这里，cyclic指向正在处理的txd，而对于其他类型的txd，cyclic为NULL
第13行获得txd对应的callback
第19行 如果存在回调函数的话，就调用callback，通知驱动有一个dsg处理完毕了
第21行，如果head内容为空，表示目前没有处理完毕的txd，否则的话： 第23行获得txd的callback，第25行将txd从head链表中删除，第26行如果该txd可以被重复使用的话，将该txd重新加入到desc_allocated链表中，否则的就将该txd释放，第31行调用txd的callback。
第29行的desc_free由用户定义，对于2440，设置的是s3c24xx_dma_desc_free

```
 1.  static void s3c24xx_dma_desc_free(struct virt_dma_desc *vd)
 2.  {
 3.      struct s3c24xx_txd *txd = to_s3c24xx_txd(&vd->tx);
 4.      struct s3c24xx_dma_chan *s3cchan = to_s3c24xx_dma_chan(vd->tx.chan);
 5.
 6.      if (!s3cchan->slave)
 7.          dma_descriptor_unmap(&vd->tx);
 8.
 9.      s3c24xx_dma_free_txd(txd);
10.  }
```

第9行用于释放txd以及txd->dsg_list的所有dsg

继续回到中断处理函数：
第33行用于处理下一个dsg
第34行，如果txd中的dsg被处理完毕，并且该txd不是cyclic类型的话，第36行调用vchan_cookie_complete表示该txd处理完毕，txd会被移动到desc_completed链表，然后调度vc->task底半部，然后vchan_complete会被执行。
第42行调用vchan_next_desc判断desc_issued是否为空，非空的话，开始处理下一个txd，否则的话调用s3c24xx_dma_phy_free:

```
1.  static void s3c24xx_dma_phy_free(struct s3c24xx_dma_chan *s3cchan)
2.  {
3.      struct s3c24xx_dma_engine *s3cdma = s3cchan->host;
4.      struct s3c24xx_dma_chan *p, *next;
5.
6.  retry:
7.      next = NULL;
8.
9.      /* Find a waiting virtual channel for the next transfer. */
10.     list_for_each_entry(p, &s3cdma->memcpy.channels, vc.chan.device_node)
11.         if (p->state == S3C24XX_DMA_CHAN_WAITING) {
12.             next = p;
13.             break;
14.         }
15.
16.     if (!next) {
17.         list_for_each_entry(p, &s3cdma->slave.channels,
18.                     vc.chan.device_node)
19.             if (p->state == S3C24XX_DMA_CHAN_WAITING &&
20.                       s3c24xx_dma_phy_valid(p, s3cchan->phy)) {
21.                 next = p;
22.                 break;
23.             }
24.     }
25.
26.     /* Ensure that the physical channel is stopped */
27.     s3c24xx_dma_terminate_phy(s3cchan->phy);
28.
29.     if (next) {
30.         bool success;
31.
32.         /*
33.          * Eww.  We know this isn't going to deadlock
34.          * but lockdep probably doesn't.
35.          */
36.         spin_lock(&next->vc.lock);
37.         /* Re-check the state now that we have the lock */
38.         success = next->state == S3C24XX_DMA_CHAN_WAITING;
39.         if (success)
40.             s3c24xx_dma_phy_reassign_start(s3cchan->phy, next);
41.         spin_unlock(&next->vc.lock);
42.
43.         /* If the state changed, try to find another channel */
```

```
44.         if (!success)
45.             goto retry;
46.     } else {
47.         /* No more jobs, so free up the physical channel */
48.     290.638067] memory_dma 4b000000.tq2440_mem_dma_demo_v4: we get: pengdong
[  290.638067]

四、其他
1、从哦哦可ie

完。


    s3c24xx_dma_put_phy(s3cchan->phy);
49.     }
50.
51.     s3cchan->phy = NULL;
52.     s3cchan->state = S3C24XX_DMA_CHAN_IDLE;
53. }
```

第10行的for循环判断memcpy这个dma device下的虚拟DMA是否有处于WAITING状态的
第17行的for循环用于判断slave这个dma device下的虚拟DMA是否有WAITING的
第27行关闭物理DMA
第29行如果next非空的话，表示找到一个处于WAITING状态的虚拟DMA，那么第40行开始处理该虚拟DMA:

```
 1. static void s3c24xx_dma_phy_reassign_start(struct s3c24xx_dma_phy *phy,
 2.     struct s3c24xx_dma_chan *s3cchan)
 3. {
 4.     struct s3c24xx_dma_engine *s3cdma = s3cchan->host;
 5.
 6.     dev_dbg(&s3cdma->pdev->dev, "reassigned physical channel %d for xfer on
 7.         phy->id, s3cchan->name);
 8.
 9.     /*
10.      * We do this without taking the lock; we're really only concerned
11.      * about whether this pointer is NULL or not, and we're guaranteed
12.      * that this will only be called when it _already_ is non-NULL.
13.      */
14.     phy->serving = s3cchan;
15.     s3cchan->phy = phy;
16.     s3cchan->state = S3C24XX_DMA_CHAN_RUNNING;
17.     s3c24xx_dma_start_next_txd(s3cchan);
18. }
```

第48行将phy->serving置为NULL，表示该物理DMA空闲
第643行将处理完毕的虚拟DMA设置为IDLE
继续回到中断处理函数：
第46行表示cyclic类型的txd被处理完毕， cyclic的txd没有调用

vchan_cookie_complete, 因为该txd会被循环处理
第47行的vchan_cyclic_callback上面已经分析过了，主要是调用txd的回调函数
第50行将at重新指向第一个dsg，然后第51行重新开始处理该txd

此外，还有三个函数需要分析：
1、dmaengine_terminate_async
2、dmaengine_tx_status
3、dma_release_channel

dmaengine_terminate_async用与强制终止某个虚拟DMA通道的DMA传输，并释放txd占用的内存，dmaengine_tx_status用于查询当前虚拟DMA通道中cookie指定的txd的传输状态（是否传输完毕，还剩余多少），dma_release_channel用于释放虚拟DMA通道

- dmaengine_terminate_async

```
1.  static inline int dmaengine_terminate_async(struct dma_chan *chan)
2.  {
3.      if (chan->device->device_terminate_all)
4.          return chan->device->device_terminate_all(chan);
5.
6.      return -EINVAL;
7.  }
```

对于2440，第4行的device_terminate_all被设置为s3c24xx_dma_terminate_all

```
1.  static int s3c24xx_dma_terminate_all(struct dma_chan *chan)
2.  {
3.      struct s3c24xx_dma_chan *s3cchan = to_s3c24xx_dma_chan(chan);
4.      struct s3c24xx_dma_engine *s3cdma = s3cchan->host;
5.      unsigned long flags;
6.      int ret = 0;
7.
8.      spin_lock_irqsave(&s3cchan->vc.lock, flags);
9.
10.     if (!s3cchan->phy && !s3cchan->at) {
11.         dev_err(&s3cdma->pdev->dev, "trying to terminate already stopped cha
12.             s3cchan->id);
13.         ret = -EINVAL;
14.         goto unlock;
15.     }
16.
17.     s3cchan->state = S3C24XX_DMA_CHAN_IDLE;
18.
19.     /* Mark physical channel as free */
20.     if (s3cchan->phy)
21.         s3c24xx_dma_phy_free(s3cchan);
22.
```

```
23.     /* Dequeue current job */
24.     if (s3cchan->at) {
25.         s3c24xx_dma_desc_free(&s3cchan->at->vd);
26.         s3cchan->at = NULL;
27.     }
28.
29.     /* Dequeue jobs not yet fired as well */
30.     s3c24xx_dma_free_txd_list(s3cdma, s3cchan);
31. unlock:
32.     spin_unlock_irqrestore(&s3cchan->vc.lock, flags);
33.
34.     return ret;
35. }
```

第17行将虚拟DMA设置为IDLE

第21行s3c24xx_dma_phy_free上面分析过了，首先寻找WAITING状态的虚拟DMA，然后关闭物理DMA，如果找到WAITING状态的虚拟DMA的话，开始处理

第25行，释放虚拟DMA正在处理的txd

第30行，释放虚拟DMA中所有的txd

- dmaengine_tx_status

```
1.  static inline enum dma_status dmaengine_tx_status(struct dma_chan *chan,
2.      dma_cookie_t cookie, struct dma_tx_state *state)
3.  {
4.      return chan->device->device_tx_status(chan, cookie, state);
5.  }
```

对于2440，第4行的device_tx_status被设置为了s3c24xx_dma_tx_status

```
1.  static enum dma_status s3c24xx_dma_tx_status(struct dma_chan *chan,
2.          dma_cookie_t cookie, struct dma_tx_state *txstate)
3.  {
4.      struct s3c24xx_dma_chan *s3cchan = to_s3c24xx_dma_chan(chan);
5.      struct s3c24xx_txd *txd;
6.      struct s3c24xx_sg *dsg;
7.      struct virt_dma_desc *vd;
8.      unsigned long flags;
9.      enum dma_status ret;
10.     size_t bytes = 0;
11.
12.     spin_lock_irqsave(&s3cchan->vc.lock, flags);
13.     ret = dma_cookie_status(chan, cookie, txstate);
14.
15.     /*
16.      * There's no point calculating the residue if there's
17.      * no txstate to store the value.
18.      */
19.     if (ret == DMA_COMPLETE || !txstate) {
20.         spin_unlock_irqrestore(&s3cchan->vc.lock, flags);
21.         return ret;
```

```
22.     }
23.
24.     vd = vchan_find_desc(&s3cchan->vc, cookie);
25.     if (vd) {
26.         /* On the issued list, so hasn't been processed yet */
27.         txd = to_s3c24xx_txd(&vd->tx);
28.
29.         list_for_each_entry(dsg, &txd->dsg_list, node)
30.             bytes += dsg->len;
31.     } else {
32.         /*
33.          * Currently running, so sum over the pending sg's and
34.          * the currently active one.
35.          */
36.         txd = s3cchan->at;
37.
38.         dsg = list_entry(txd->at, struct s3c24xx_sg, node);
39.         list_for_each_entry_from(dsg, &txd->dsg_list, node)
40.             bytes += dsg->len;
41.
42.         bytes += s3c24xx_dma_getbytes_chan(s3cchan);
43.     }
44.     spin_unlock_irqrestore(&s3cchan->vc.lock, flags);
45.
46.     /*
47.      * This cookie not complete yet
48.      * Get number of bytes left in the active transactions and queue
49.      */
50.     dma_set_residue(txstate, bytes);
51.
52.     /* Whether waiting or running, we're in progress */
53.     return ret;
54. }
```

第13行，查询虚拟DMA中cookie指定的txd的是否传输完毕
第24行，根据cookie从虚拟DMA通道的desc_issued链表中找到对应的txd， 我们知道desc_issued中是已经submic但是还没有开始传输的txd
第25行，如果vd非空的话，表示该txd还没有开始传输，然后统计该txd的dsg_list中需要传输的数据总大小
第31行，vd为NULL的话表示cookie表示的txd已经开始进行DMA传输了，那么我们就需要统计dsg_list中还没有被处理的dsg表示的数据的总长度， 第36行获得正在处理的txd，第38到42行统计剩余还没有传输的数据
第50行，将尚未传输的字节数存放到txstate->residue中

- dma_release_channel

```
1.  void dma_release_channel(struct dma_chan *chan)
2.  {
3.      mutex_lock(&dma_list_mutex);
4.      WARN_ONCE(chan->client_count != 1,
5.            "chan reference count %d != 1\n", chan->client_count);
```

```
6.      dma_chan_put(chan);
7.      /* drop PRIVATE cap enabled by __dma_request_channel() */
8.      if (--chan->device->privatecnt == 0)
9.          dma_cap_clear(DMA_PRIVATE, chan->device->cap_mask);
10.     mutex_unlock(&dma_list_mutex);
11. }
```

这里的核心函数是第6行：dma_chan_put

```
1.  static void dma_chan_put(struct dma_chan *chan)
2.  {
3.      /* This channel is not in use, bail out */
4.      if (!chan->client_count)
5.          return;
6.
7.      chan->client_count--;
8.      module_put(dma_chan_to_owner(chan));
9.
10.     /* This channel is not in use anymore, free it */
11.     if (!chan->client_count && chan->device->device_free_chan_resources) {
12.         /* Make sure all operations have completed */
13.         dmaengine_synchronize(chan);
14.         chan->device->device_free_chan_resources(chan);
15.     }
16.
17.     /* If the channel is used via a DMA request router, free the mapping */
18.     if (chan->router && chan->router->route_free) {
19.         chan->router->route_free(chan->router->dev, chan->route_data);
20.         chan->router = NULL;
21.         chan->route_data = NULL;
22.     }
23. }
```

这个函数会将该client_count减一，如果client_count减到0的话，就会释放虚拟DMA占用的内存：
第13行：dmaengine_synchronize在2440中没有定义
第14行，2440设置的是s3c24xx_dma_free_chan_resources

```
1.  static void s3c24xx_dma_free_chan_resources(struct dma_chan *chan)
2.  {
3.      /* Ensure all queued descriptors are freed */
4.      vchan_free_chan_resources(to_virt_chan(chan));
5.  }
```

第4行调用vchan_free_chan_resources释放虚拟通道下所有的txd，以及txd下所有的dsg，而且会清除txd的reuse标志

```
1.  static inline void vchan_free_chan_resources(struct virt_dma_chan *vc)
2.  {
3.      struct virt_dma_desc *vd;
4.      unsigned long flags;
```

```
5.      LIST_HEAD(head);
6.
7.      spin_lock_irqsave(&vc->lock, flags);
8.      vchan_get_all_descriptors(vc, &head);
9.      list_for_each_entry(vd, &head, node)
10.         dmaengine_desc_clear_reuse(&vd->tx);
11.     spin_unlock_irqrestore(&vc->lock, flags);
12.
13.     vchan_dma_desc_free_list(vc, &head);
14. }
```

第8行将vc的desc_allocated/desc_submitted/desc_issued/desc_completed链表中的所有项都移到head中
第9行的for循环清楚所有txd的reuse标志，防止后面vchan_dma_desc_free_list将设置了reuse标志的txd又重新被添加到desc_allocated中。
第13行会释放head中所有txd以及txd->dsg_list中的dsg占用的内存资源

三、测试

启动开发板，加载驱动，向/dev/mem_dma_test写入字符串：

```
1.  [root@tq2440 mnt]# insmod memory_dma_v4.ko
2.  [  278.180266] memory_dma 4b000000.tq2440_mem_dma_demo_v4: memory_dma_probe
3.  [root@tq2440 mnt]# echo "pengdonglin" > /dev/mem_dma_test
4.  [  290.620588] memory_dma 4b000000.tq2440_mem_dma_demo_v4: mem_dma_write en
5.  [  290.622109] memory_dma 4b000000.tq2440_mem_dma_demo_v4: from user: pengd
6.  [  290.622109]
7.  [  290.634830] memory_dma 4b000000.tq2440_mem_dma_demo_v4: transfer complet
8.  [  290.638067] memory_dma 4b000000.tq2440_mem_dma_demo_v4: we get: pengdong
9.  [  290.638067]
```

四、其他

1、cookie （drivers/dma/dmaengine.h）

- 初始化

在dma device添加虚拟DMA通道的时候在vchan_init中调用dma_cookie_init对

```
1.  /**
2.   * dma_cookie_init - initialize the cookies for a DMA channel
3.   * @chan: dma channel to initialize
4.   */
5.  static inline void dma_cookie_init(struct dma_chan *chan)
6.  {
7.      chan->cookie = DMA_MIN_COOKIE;
```

```
    chan->completed_cookie = DMA_MIN_COOKIE;
}
```

- 分配 （cookie用于标示一个txd）

dmaengine_submit --> vchan_tx_submit --> dma_cookie_assign

```
/**
 * dma_cookie_assign - assign a DMA engine cookie to the descriptor
 * @tx: descriptor needing cookie
 *
 * Assign a unique non-zero per-channel cookie to the descriptor.
 * Note: caller is expected to hold a lock to prevent concurrency.
 */
static inline dma_cookie_t dma_cookie_assign(struct dma_async_tx_descriptor
{
    struct dma_chan *chan = tx->chan;
    dma_cookie_t cookie;

    cookie = chan->cookie + 1;
    if (cookie < DMA_MIN_COOKIE)
        cookie = DMA_MIN_COOKIE;
    tx->cookie = chan->cookie = cookie;

    return cookie;
}
```

从这里看到，一个chan下可以挂多个tx，每一个tx都有唯一的cookie，而chan->cookie记录的最新的被submit的tx的cookie

- 完成

s3c24xx_dma_irq --> vchan_cookie_complete --> dma_cookie_complete

```
/**
 * dma_cookie_complete - complete a descriptor
 * @tx: descriptor to complete
 *
 * Mark this descriptor complete by updating the channels completed
 * cookie marker.  Zero the descriptors cookie to prevent accidental
 * repeated completions.
 *
 * Note: caller is expected to hold a lock to prevent concurrency.
 */
static inline void dma_cookie_complete(struct dma_async_tx_descriptor *tx)
{
    BUG_ON(tx->cookie < DMA_MIN_COOKIE);
    tx->chan->completed_cookie = tx->cookie;
    tx->cookie = 0;
}
```

将刚刚处理完毕的txd的cookie存放到completed_cookie中

- 获得cookie标示的txd的状态，是否处理完毕

s3c24xx_dma_tx_status --> dma_cookie_status

```
/**
 * dma_cookie_status - report cookie status
 * @chan: dma channel
 * @cookie: cookie we are interested in
 * @state: dma_tx_state structure to return last/used cookies
 *
 * Report the status of the cookie, filling in the state structure if
 * non-NULL.  No locking is required.
 */
static inline enum dma_status dma_cookie_status(struct dma_chan *chan,
    dma_cookie_t cookie, struct dma_tx_state *state)
{
    dma_cookie_t used, complete;

    used = chan->cookie;
    complete = chan->completed_cookie;
    barrier();
    if (state) {
        state->last = complete;
        state->used = used;
        state->residue = 0;
    }
    return dma_async_is_complete(cookie, complete, used);
}
```

used记录的是最近被submit的txd的cookie，complete记录的是最新地被处理完毕的txd的cookie

```
/**
 * dma_async_is_complete - test a cookie against chan state
 * @cookie: transaction identifier to test status of
 * @last_complete: last know completed transaction
 * @last_used: last cookie value handed out
 *
 * dma_async_is_complete() is used in dma_async_is_tx_complete()
 * the test logic is separated for lightweight testing of multiple cookies
 */
static inline enum dma_status dma_async_is_complete(dma_cookie_t cookie,
            dma_cookie_t last_complete, dma_cookie_t last_used)
{
    if (last_complete <= last_used) {
        if ((cookie <= last_complete) || (cookie > last_used))
            return DMA_COMPLETE;
    } else {
        if ((cookie <= last_complete) && (cookie > last_used))
            return DMA_COMPLETE;
    }
    return DMA_IN_PROGRESS;
}
```

从上面可以看到判断一个txd是否传输完成的方法： 我们知道，一个虚拟DMA通道下面可以添加多个txd，这些txd会被按照添加的顺序先后进行串行处理，而且后添加的txd的cookie一定比前面添加的cookie大，有了这个前提，就很容易理解上面的算法.
从前面我们的分析，第13行的if判断一定会成立，剩下的就是第14行的(cookie <= last_complete)，如果这个条件成立，那么cookie标示的txd一定被处理完了，否则的话，有可能还没有处理，也可能正在处理。

完。