# 作者

彭东林
pengdonglin137@163.com

# 平台

TQ2440
Linux-4.9

# 概述

上一篇博客分析了DMA控制器的寄存器，循序渐进，下面我们直接操作DMA控制器的寄存器实现一个mem2mem的DMA传输。

# 正文

## 一、修改设备树

为了避免kernel自带的s3c2440 dma驱动的干扰，修改设备树，将DMA控制器的节点disable，同时最好把声卡相关的节点也关闭，因为声卡的注册会依赖DMA。然后添加我们demo驱动要用的设备树节点。

```
    dma: s3c2410-dma@4B000000 {
        compatible = "tq2440, 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>;
        status = "disabled"; // 原先的DMA控制器节点，为防止干扰，关闭
    };

    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";
        status = "disabled";  // 在注册的时候会申请DMA资源，所以需要关闭
```

```
    };

    s3c24xx_uda134x {
        compatible = "s3c24xx_uda134x";
        clocks = <&clocks MPLL>, <&clocks PCLK_I2S>;
        clock-names = "mpll", "iis";
        status = "disabled";  // 由于iis没有注册，alsa的machine驱动注册一定会失败，所以
    };

    tq2440_mem_dma_demo_v3 { // 这是我们这节的demo驱动需要的节点，只需reg属性即可
        compatible = "tq2440,memory_dma_v3";
        reg = <0x4B000000 0x1000>;
    };
```

## 二、驱动

```
#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>
#include <linux/delay.h>

//缓冲区大小，将来会在两块4K大小的连续物理内存之间进行DMA传输
#define BUFSIZE  (4*1024)

typedef struct {
    volatile u32 DISRC;
    volatile u32 DISRCC;
    volatile u32 DIDST;
    volatile u32 DIDSTC;
    volatile u32 DCON;
    volatile u32 DSTAT;
    volatile u32 DCSRC;
    volatile u32 DCDST;
    volatile u32 DMASKTRIG;
} dma_reg_t;

// 我们以DMAC0为例，这里是起始地址
static volatile dma_reg_t *ch0_base;

// 由于DMA传输时直接跟系统总线通信，不经过MMU，所以需要获得物理地址
static dma_addr_t src, dst;

// 初始化这两块内存需要的时虚拟地址
static char *src_virt, *dst_virt;
static struct device *dev;

```

```c
static int mem_dma_open(struct inode *inode, struct file *file)
{
    // 设置源地址
    ch0_base->DISRC = src;
    // AHB 递增
    ch0_base->DISRCC = (0<<1) | (0<<0);

    // 设置目的地址
    ch0_base->DIDST = dst;
    // AHB 递增
    ch0_base->DIDSTC = (0<<1) | (0<<0);

    // Demand模式，跟HCLK同步，关闭中断，transfer size设置为1
    // Whole Service模式，软件触发，关闭auto reload，data size设置为1字节
    ch0_base->DCON = (0<<31) | (1<<30) | (0<<29) | (0<<28) | (1<<27) \
             | (0<<23) | (1<<22) | (0<<20);

    // 清除上次的数据，这里使用的是虚拟地址，不能使用物理地址
    memset(src_virt, 0x0, BUFSIZE);
    memset(dst_virt, 0x0, BUFSIZE);

    return 0;
}

static ssize_t mem_dma_write(struct file *file, const char *data,
                size_t len, loff_t *ppos)
{
    dev_info(dev, "%s enter.\n", __func__);

    // 将用户的数据拷贝到源内存，使用的也是虚拟地址
    copy_from_user(src_virt, data, len);
    dev_info(dev, "from user: %s\n", src_virt);

    // 由于transfer和data size都是1，所以这里的len可以直接写入DCON[19:0]
    // 需要注意的是不要越界，只保留len的[19:0]即可
    (ch0_base->DCON) |= (len & ((1<<20)-1));

    //清除STOP位，打开DMA通道，软件触发
    ch0_base->DMASKTRIG = (0<<2) | (1<<1) | (1<<0);

    // 等待DMA空闲，也就是DMA传输完毕
    while ((ch0_base->DSTAT>>20)&0x3)
        msleep(10);

    // 将目的地址内存的内容打印出来，看看跟源地址内存内容是否相同
    dev_info(dev, "transfer end!\n");
    dev_info(dev, "we got: %s\n", dst_virt);

    return len;
}

static const struct file_operations mem_dma_fops = {
    .owner      = THIS_MODULE,
    .open       = mem_dma_open,
```

```
90.     .write      = mem_dma_write,
91. };
92. 
93. static struct miscdevice mem_dma_miscdev = {
94.     .minor  = MISC_DYNAMIC_MINOR,
95.     .name   = "mem_dma_test",
96.     .fops   = &mem_dma_fops,
97. };
98. 
99. static int memory_dma_probe(struct platform_device *pdev) {
100.     int ret = 0;
101.     struct resource *res;
102. 
103.     dev = &pdev->dev;
104.     dev_info(dev, "%s enter.\n", __func__);
105. 
106.     if (!dev->of_node) {
107.         dev_err(dev, "no platform data.\n");
108.         return -EINVAL;
109.     }
110. 
111.     res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
112.     // 映射DMAC的寄存器资源，此时ch0_base指向DMAC0的起始地址
113.     ch0_base = devm_ioremap_resource(dev, res);
114. 
115.     // 分配一块4k大小的连续的物理地址，对应的物理地址存放在src中
116.     // 返回值是对应的虚拟地址
117.     src_virt = dma_zalloc_coherent(dev, BUFSIZE, &src, GFP_KERNEL);
118.     if (!src_virt) {
119.         dev_err(dev, "%s: alloc src failed.\n", __func__);
120.         ret = -ENOMEM;
121.         goto err1;
122.     }
123. 
124.     dst_virt = dma_zalloc_coherent(dev, BUFSIZE, &dst, GFP_KERNEL);
125.     if (!dst_virt) {
126.         dev_err(dev, "%s: alloc dst failed.\n", __func__);
127.         ret = -ENOMEM;
128.         goto err2;
129.     }
130. 
131.     ret = misc_register(&mem_dma_miscdev);
132.     if (ret < 0) {
133.         pr_err("failed to register mem_dma device\n");
134.         goto err3;
135.     }
136. 
137.     return 0;
138. 
139. err3:
140.     dma_free_coherent(dev, BUFSIZE, dst_virt, dst);
141. err2:
142.     dma_free_coherent(dev, BUFSIZE, src_virt, src);
143. err1:
```

```
144.     return ret;
145. 
146. }
147. 
148. static int memory_dma_remove(struct platform_device *pdev) {
149. 
150.     dev_info(dev, "%s enter.\n", __func__);
151.     misc_deregister(&mem_dma_miscdev);
152. 
153.     dma_free_coherent(dev, BUFSIZE, dst_virt, dst);
154.     dma_free_coherent(dev, BUFSIZE, src_virt, src);
155. 
156.     return 0;
157. }
158. 
159. static const struct of_device_id memory_dma_dt_ids[] = {
160.     { .compatible = "tq2440,memory_dma_v3", },
161.     {},
162. };
163. 
164. MODULE_DEVICE_TABLE(of, memory_dma_dt_ids);
165. 
166. static struct platform_driver memory_dma_driver = {
167.     .driver        = {
168.         .name    = "memory_dma",
169.         .of_match_table    = of_match_ptr(memory_dma_dt_ids),
170.     },
171.     .probe        = memory_dma_probe,
172.     .remove        = memory_dma_remove,
173. };
174. 
175. static int __init memory_dma_init(void)
176. {
177.     int ret;
178. 
179.     ret = platform_driver_register(&memory_dma_driver);
180.     if (ret)
181.         printk(KERN_ERR "memory_dma: probe failed: %d\n", ret);
182. 
183.     return ret;
184. }
185. module_init(memory_dma_init);
186. 
187. static void __exit memory_dma_exit(void)
188. {
189.     platform_driver_unregister(&memory_dma_driver);
190. }
191. module_exit(memory_dma_exit);
192. 
193. MODULE_LICENSE("GPL");
```

## 三、测试

加载驱动后，会在/dev下面生成一个名为mem_dma_test的设备节点，我们可以往其中echo内容：

```
[root@tq2440 mnt]# insmod memory_dma_v3.ko
[  283.494939] memory_dma 4b000000.tq2440_mem_dma_demo_v3: memory_dma_probe enter.
[root@tq2440 mnt]# echo "pengdonglin137@163.com" > /dev/mem_dma_test
[  298.176085] memory_dma 4b000000.tq2440_mem_dma_demo_v3: mem_dma_write enter.
[  298.177619] memory_dma 4b000000.tq2440_mem_dma_demo_v3: from user: pengdonglin13
[  298.177619]
[  298.204600] memory_dma 4b000000.tq2440_mem_dma_demo_v3: transfer end!
[  298.205513] memory_dma 4b000000.tq2440_mem_dma_demo_v3: we got: pengdonglin137@1
[  298.205513]
```

可以看到DMA传输成功。

# 四、其他

## 1、dma_zalloc_coherent

参考内核文档：
Documentation/DMA-API.txt
Documentation/DMA-API-HOWTO.txt
Linux-DMA-In-Device-Drivers

这个函数是对dma_alloc_coherent的wrapper，分配DMA传输用的内存之所以不用kmalloc是因为用kmalloc分配的内存会cache，而
用dma_alloc_coherent分配的内存却不用担心这一问题，正如内核文档所说：

**Consistent memory is memory for which a write by either the device or the processor can immediately be read by the processor or device without having to worry about caching effects. (You may however need to make sure to flush the processor's write buffers before telling devices to read that memory.)**

完。