

音频驱动分析

何聪

2012-8-10

ac97 驱动，目前只实现了播放音的功能，大部分来源于 pmon，所以源码不详细说，只说一下我修改及添加的内容。

1. 重要的数据结构

```
static struct desc{
    u32 ordered;      //下一个描述符地址寄存器
    u32 saddr;        //内存地址寄存器
    u32 daddr;        //设备地址寄存器
    u32 length;       //长度寄存器
    u32 step_length;  //间隔长度寄存器
    u32 step_times;   //循环次数寄存器
    u32 cmd;          //控制寄存器
};
```

对应 1b 数据手册上 dma 描述符。

```
#define PAGE_SIZE 0x1000
#define QUEUE_MAX 2
#define PAGE_NR 16 /* dma buffer page number */
static unsigned char dma_pool[PAGE_SIZE * (PAGE_NR * (QUEUE_MAX + 1)*2)];
```

定义一个全局的内存空间，后面的 dma buffer 将从这个空间来分配。我们定义的 dma queue 为 2，总共两个队列，一个输入，一个输出（等下就会看到），每个 dma buffer 分配 64KB。当然，队列元素的个数，分配空间的页数，只要在此更改即可，不需要修改其他的代码。

```
typedef struct buffer_queue_s {
    int count; /* current usable buffer item */
    int id[QUEUE_MAX]; /* alloc array id */
    int lock;
} buffer_queue_t;
```

```
static struct desc *DMA_DESC_BASE;
```

buffer queue 分配，将利用这个结构。

```
static unsigned long out_dma_buf[QUEUE_MAX+1];
static unsigned long out_dma_pbuf[QUEUE_MAX+1];
static unsigned long out_dma_buf_data_count[QUEUE_MAX+1];
static unsigned long in_dma_buf[QUEUE_MAX+1];
static unsigned long in_dma_pbuf[QUEUE_MAX+1];
static unsigned long in_dma_buf_data_count[QUEUE_MAX+1];
```

buf 存放的是 dma 将操作的内存空间的虚拟地址，pbuf 是物理地址，dma_buf_data_count 记录 dma 传输的字节。

```
static buffer_queue_t out_empty_queue;
static buffer_queue_t out_full_queue;
static buffer_queue_t out_busy_queue;
```

```
static buffer_queue_t in_empty_queue;
static buffer_queue_t in_full_queue;
static buffer_queue_t in_busy_queue;
```

dma in 和 out 都定义了 empty、full、busy 三个 dma bufer 操作对象。详细的算法在后面说。

2. init_audio_data

```
DMA_DESC_BASE = (struct desc*)malloc(sizeof(struct desc) + 32);
DMA_DESC_BASE = ((u32)DMA_DESC_BASE+31) & (~31);
分配描述符空间，并且 4 字节对齐（1b 手册上说了要双字对齐）。
in_empty_queue.count = QUEUE_MAX;
out_empty_queue.count = QUEUE_MAX;

for (i=0;i<QUEUE_MAX;i++) {
    in_empty_queue.id[i] = i;
    out_empty_queue.id[i] = i;
}

in_full_queue.count = 0;
in_busy_queue.count = 0;
out_busy_queue.count = 0;
out_full_queue.count = 0;

/* Aligned in PAGE_SIZE */
p = (unsigned char *)(((unsigned int)p + PAGE_SIZE) & ~(PAGE_SIZE-1));

for (i = 0; i < QUEUE_MAX; i++) {
    out_dma_buf[i] = (unsigned long)p;
    p += PAGE_SIZE*PAGE_NR;
    out_dma_pbuf[i] = PHYADDR((unsigned int)out_dma_buf[i]);
    in_dma_buf[i] = (unsigned long)p;
    p += PAGE_SIZE*PAGE_NR;
    in_dma_pbuf[i] = PHYADDR((unsigned int)in_dma_buf[i]);
}
```

都是一些对前面定义的数据结构进行基本初始化，(in || out)_dma_buf 从 dma_pool 中取得了应有的空间。

```
ac97_config();
ac97 寄存器配置。

/* dma interrupt */
request_irq(LS1B_BOARD_DMA1_IRQ, ac97_dma_write_intr, NULL);
request_irq(LS1B_BOARD_DMA2_IRQ, ac97_dma_read_intr, NULL);
为 ac97 dma 申请读写中断。
```

3. get_id_buffer、put_id_buffer 和 elements_in_queue

首先分析下一些操作函数：

```
static inline int get_buffer_id(struct buffer_queue_s *q)
{
    int r;
    unsigned long flags;
    int i;
    flags = spin_lock_irqsave();
    if (q->count == 0) return -1;
    r = q->id[0];
    for (i=0; i < q->count-1; i++)
        q->id[i] = q->id[i+1];
    q->count --;
    spin_unlock_irqrestore(flags);
    return r;
}
```

假设 QUEUE_MAX=5（实际代码中是 2，大点好分析点），看上面的初始化函数中，q->count=QUEUE_MAX，q->id[i]=i，所以初始 id 如下：

i	0	1	2	3	4
id	0	1	2	3	4
count	5				
r	0				

get_buffer_id 连续操作多次。id 的变化：

第一次

i	0	1	2	3	4
id	1	2	3	4	4
count	4				
r	0				

第二次

i	0	1	2	3	4
id	2	3	4	4	4
count	3				
r	1				

第三次

i	0	1	2	3	4
id	3	4	4	4	4
count	2				
r	2				

第四次

i	0	1	2	3	4
id	4	4	4	4	4
count	1				
r	3				

第五次

i	0	1	2	3	4
id	4	4	4	4	4
count	0				
r	4				

count 已经到 0，再 get 将返回错误的-1.

```
static inline void put_buffer_id(struct buffer_queue_s *q, int id)
{
    unsigned long flags;
    flags = spin_lock_irqsave();
    q->id[q->count] = id;
    q->count ++;
    spin_unlock_irqrestore(flags);
}
```

从上面随便选一种情况来分析，就选第五次 get 后的结果，put_buffer_id 传入的 id 可能 0 到 4 的任何值，假设是 3，那么运行后，结果如下：

i	0	1	3	4	5
id	3	4	4	4	4
count	1				

继续传入 1，结果如下：

i	0	1	2	3	4
id	3	1	4	4	4
count	2				

```
static inline int elements_in_queue(struct buffer_queue_s *q)
{
    int r;
    unsigned long flags;
    flags=spin_lock_irqsave();
    r = q->count;
    spin_unlock_irqrestore(flags);
    return r;
}
```

该函数很简单，检查队列中是否还有可用元素.

4. pcm_read

```
if (first_record_call) {
    first_record_call = 0;
    if ((id = get_buffer_id(&in_empty_queue)) >= 0) {
        put_buffer_id(&in_busy_queue, id);
        in_dma_buf_data_count[id] = copy_count;
        /* __ac97_enable_receive_dma(); */
        /* __ac97_enable_record(); */
        dma_start(in_dma_pbuf[id], in_dma_buf_data_count[id]);
        OSSemPend(rx_sem, 0, &err);
    }
}
```

```

    }
}

```

这段代码只会被运行一次，就是在第一次调用 `pcm_read` 的时候。`read` 和 `write` 分别操作三个 `buffer` 数组，`empty`、`full` 和 `busy`。`empty` 表示可用的 `buffer` 数，`full` 表示 `read` 完成 DMA 传输后，存放有效数据的 `buffer`，表示 `write` 之前填充好数据的 `buffer`，`busy` 表示当前在传输的 `buffer`。`pcm_read` 后面的代码就很容易理解了。

5. `ac97_dma_read_intr`

```
while(read_reg(confreg_base + 0x1160)&4);
```

确定是否操作 dma 传输完成。

```

    id1 = get_buffer_id(&in_busy_queue);
    put_buffer_id(&in_full_queue, id1);
    OSSemPost(rx_sem);

```

获取传输的 `buffer id`，然后放到 `full` 数组里面，表示有完成的 `buffer`。

```

    if ((id2 = get_buffer_id(&in_empty_queue)) >= 0) {
        put_buffer_id(&in_busy_queue, id2);
        in_dma_buf_data_count[id2] = in_dma_buf_data_count[id1];
        dma_start(in_dma_pbuf[id2], in_dma_buf_data_count[id2]);
    } else
        in_busy_queue.count = 0;

```

从新发起一个 DMA 传输，如果没有可用的 `empty buffer`，就把 `busy queue` 的个数置 0。结合 `pcm_read` 来看，就能清楚明白。

`pcm_read` 和 `ac97_dma_write_intr` 和读操作的差不多，就不赘述了。

6. 总结

对别 `pmon`，在 `pmon` 的基础上添加了中断的支持，添加队列操作，这样能够同时有多个 `buffer` 可供操作，来满足 DMA 传输和对内存的拷贝同时在运行，