A31 平台 DMA 开发说明文档

2013-2-22

版本历史

版本	时间	备注
V1.0	2012-11-1	建立初始版本
	2012-12-12	修改 demo.
	2013-2-22	增加框架结构说明.

目 录

1. 概述

1.1.编写目的

介绍 DMA 模块使用方法。

1.2.适用范围

适用于 A31 平台.

1.3.相关人员

DMA 开发人员。

2. 模块介绍

2.1.模块功能介绍

dma 即 Direct Memory Access(直接内存存取)，指数据不经 cpu，直接在设备和内存，内存和内存，设备和设备之间传输. 使用 DMA 可以减少 cpu 负担, cpu 可用于忙别的活，传输速度也比 cpu 搬运高得多.

A31 许多模块内置了 DMA，比如 sd, usb ehci, nand 等，目前只用两个模块用到 DMA 驱动，一是 usb otg，二是 audio codec. 当然用户可根据需要使用 DMA 驱动.

A31 DMA 模块包含 16 个独立通道，每个通道功能相同，无 dedicate 和 normal 之分.

2.2.相关术语介绍

2.2.1. DMA

Direct Memory Access，即直接内存存取，指数据不经 cpu，直接在设备和内存，内存和内存，设备和设备之间传输.

2.2.2. 描述符(des)

指能被 DMA 硬件解析的一段内存区域，其数据按一定的格式组织，包含源地址，目的地址，传输的数据长度等.

2.2.3. 散列传输

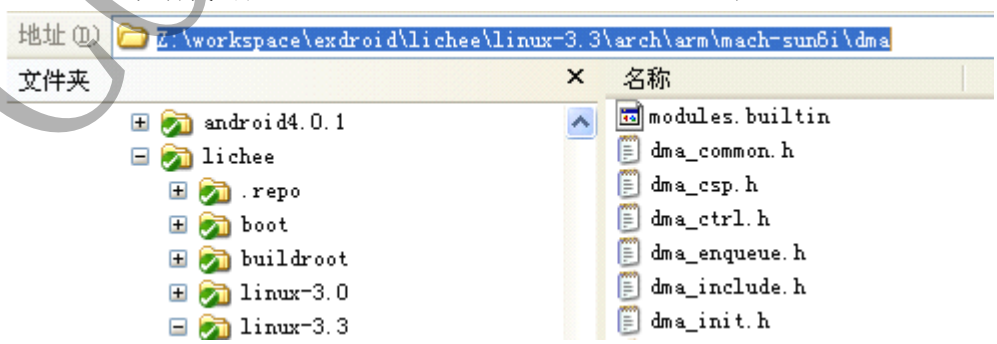
指只用启动 DMA 一次，就将多笔数据传完，即一次启动，批量传输.
软件上的散列传输，指前一笔数据传完后，由 DMA 驱动自动启动下一笔传输，硬件上还是每次传输一笔;
硬件上的散列传输，指硬件一次性将多笔传完，硬件能自动解析下一个数据块信息，这需要数据块描述符按一定的格式排布.

2.3.模块配置介绍

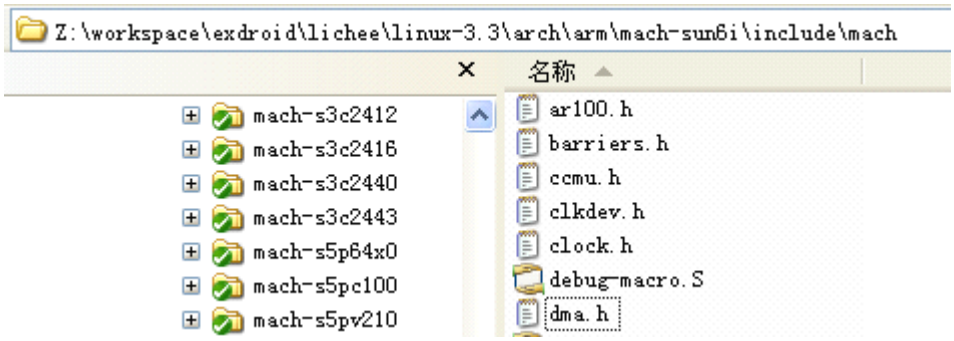
无.

2.4.源码结构介绍

DMA 驱动代码在\linux-3.3\arch\arm\mach-sun6i\dma 下:

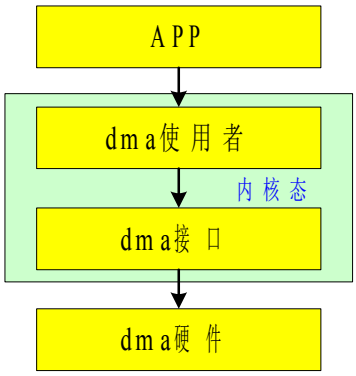


DMA 导出接口在\linux-3.3\arch\arm\mach-sun6i\include\mach\dma.h 下:



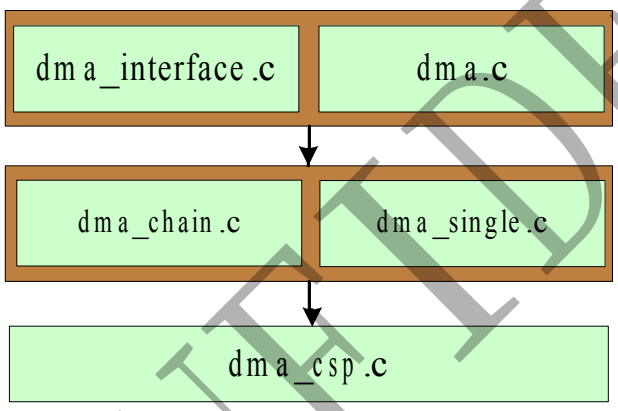
3. 模块体系结构描述

3.1. DMA 驱动架构图



(1) 应用程序 app 发起数据请求. 比如 audio 程序播放一段音乐.
(2) 内核层对应驱动响应应用层请求，调用 DMA 模块 API 进行数据传输. 比如 alsa 驱动.
(3) DMA 软件模块根据数据请求设置 DMA 硬件.
(4) DMA 硬件完成数据的实际传输.

DMA 驱动内部组织:



(1) dma_interface.c: dma 导出函数.
(2) Dma.c: dma 模块初始化, dma 中断处理.
(3) Dma_chain.c: chain 模式处理函数.
(4) Dma_single.c: single 模式处理函数.
(5) Dma_csp.c: dma 硬件操作函数.

3.2. DMA 的两种传输模式

A31 DMA 硬件上支持多包传输，即一次启动 DMA 后，将链上多个 buf 传完.
软件处理时，分两种情况:
(1) 使用硬件的多包传输的特性，即"多包工作模式"
(2) 不使用多包传输特性，即"单包工作模式"
虽然理论上多包模式效率更高，但实际使用时，单包模式更方便.
多包模式下，由于多个包共享一个中断 pending 位，因此软件上可能来不及单独处理每

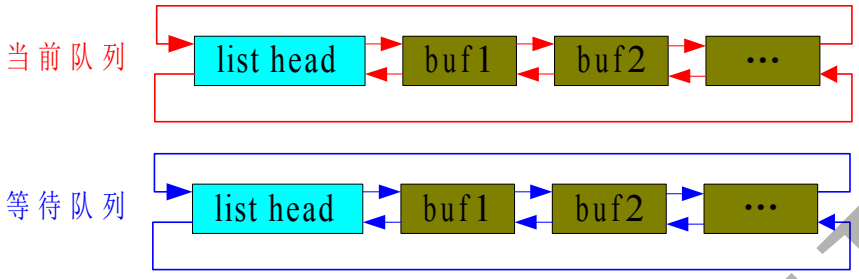
个包的中断；这在一些场合是不允许的，比如音频驱动，需要知道每个包的传输情况.

所以目前实际用到的是单包模式.

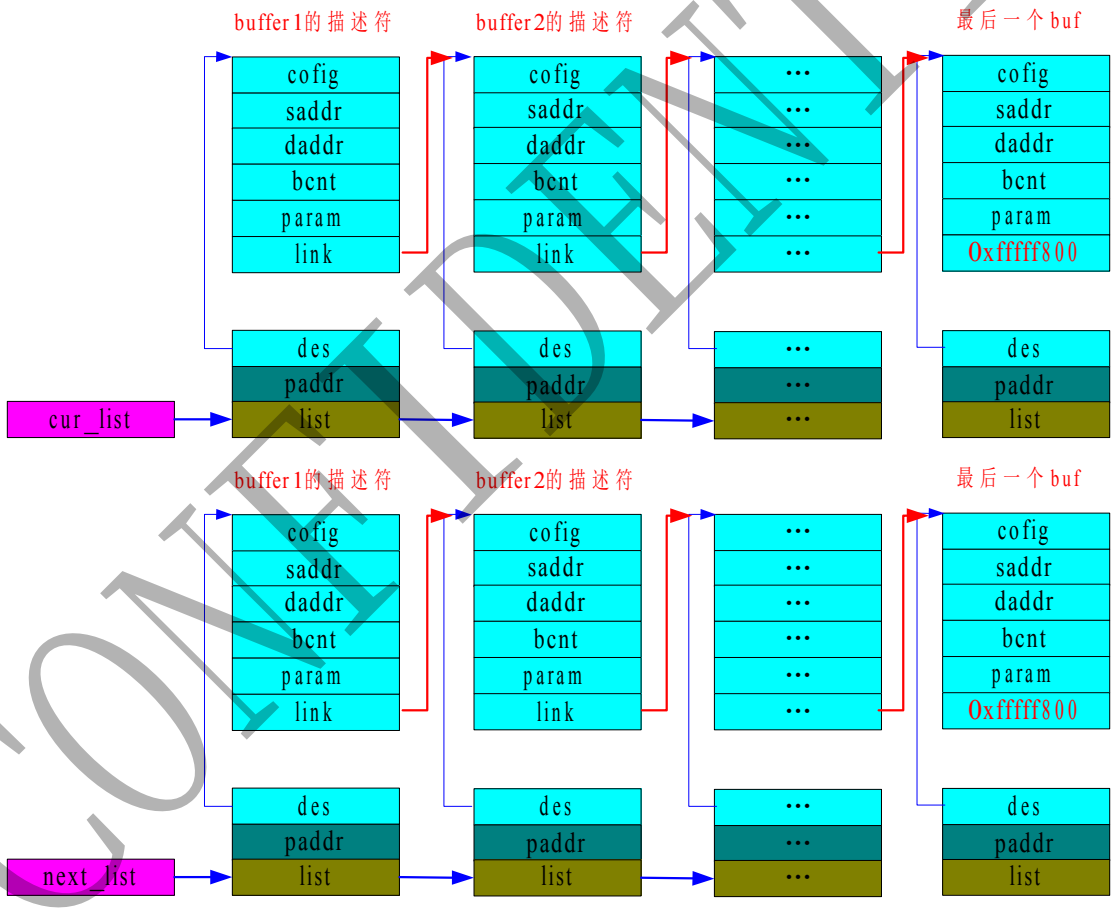
3.3.多包工作模式

3.3.1. Buf 队列管理

两个 buffer 链，一个正在传输，一个等待传输.



详细结构:



cur_list: 正在传输的队列;

next_list: 等当前队列传完后，下一步传输的队列;

为什么要弄两个队列?

因为 DMA 正在传输时，有可能有新包加进来，若只有一个队列，则需要暂停 DMA，将新包加到队列尾，再继续 DMA;

这样会导致数据传输中断，比如播音频时，可能卡顿;

而且当 start reg 为 0xfffff800 时，暂停操作是无效的，必须等 queue done 后，重现 start DMA，传输新包;

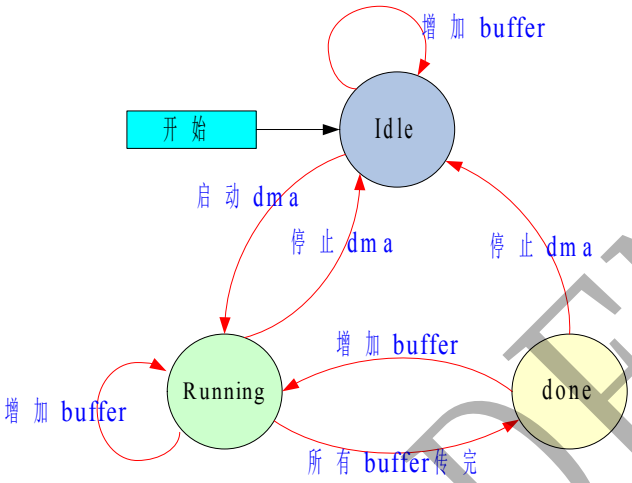
所以用两个队列更方便管理.

两个队列的使用情况:

(1) 启动 DMA 之前, buf 都加到当前队列，启动 DMA 之后，直至当前队列传完(queue done), 新的 buf 加到等待队列;

(2) 当前队列传完时，若等待队列非空，则将等待队列设为当前队列，将原当前队列清空，并开始传输当前队列；若等待队列为空，则状态机变为 done. 所有包已传完.

3.3.2. DMA 软件状态机



定义了三种软件状态:

(1) idle: 描述 DMA 硬件空闲的状态.

(2) running: 描述 DMA 正在传输的状态.

(3) done: 描述所有 buffer 传完的状态.

为什么 done 不能被 idle 替代?

虽然两种情况 DMA 硬件均空闲，但软件处理时，若在 idle 状态添加新包，不会自动 start DMA; 在 done 状态添加新包, DMA 驱动会自动启动传输.

实际使用时, DMA 的使用者很可能希望启动一次 DMA，然后只管添加新包，由 DMA 驱动将所有包传完.

关于状态变迁:

(1) 初始为 idle 状态; Idle 状态可以添加 buff;

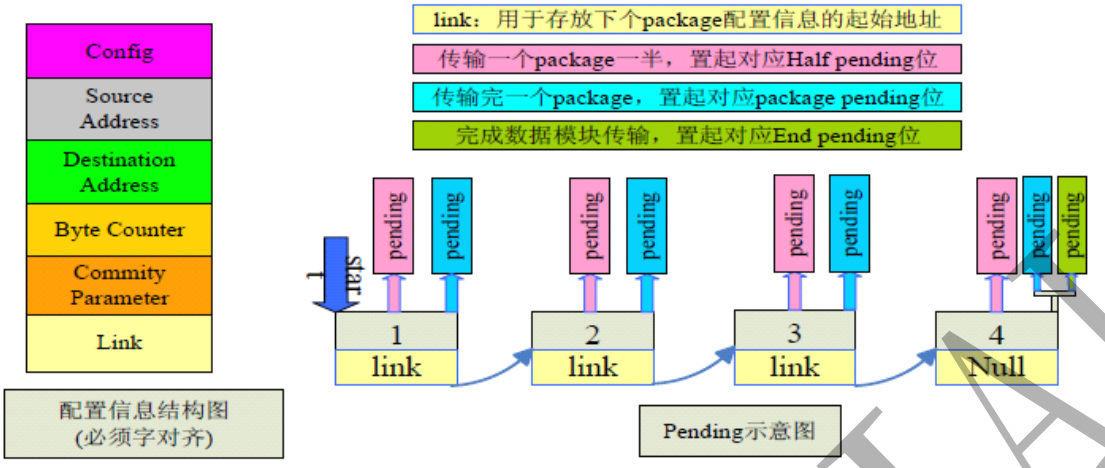
(2) Start DMA 之后，变为 running 状态;

Running 状态可以加 buff，此时 buf 加到等待队列;

(3) 当前队列传完时, DMA 中断函数若检查到等待队列非空，则将等待队列设为当前队列，将原当前队列清空，并传输当前队列.

若 DMA 中断检查等待队列为空，则状态机变为 done. 所有包已传完.

3.3.3. 描述符结构



详见 4.2 说明.

关于描述符空间申请:

描述符空间是被硬件解析的，故不能有 cached 的数据，即对描述符空间的修改应及时更新到 dram，不能缓存到 cache. 两种办法:

(1) 申请 cached 空间，比如 kmem_cache_create, kmalloc 等. 然后每次 start DMA 之前，对描述符空间进行 cache flush，将 cache 里的数据刷到 dram 中.
此法效率比较低.

(2) 申请 uncached 空间，省去每次 start DMA 之前的 cache flush 操作.
申请 uncached 空间主要有 DMA_alloc_coherent, dma_pool_alloc 两类函数，前者用于申请较大空间，多用于仅一次申请释放的场合；后者用于动态申请释放的场合.

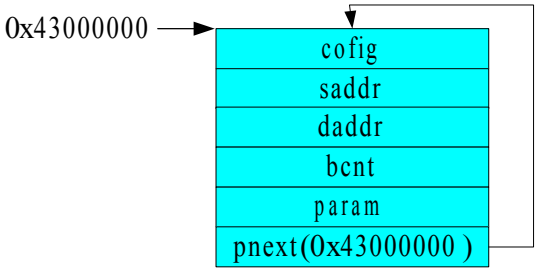
驱动采用的是 dmam_pool_alloc 方式.

3.3.4. continue 模式的处理

continue 模式是软件概念，指 DMA 传完最后一个 buffer 后，接着回过头传输第一个 buf，循环往复的过程.

要实现该功能，只需将最后一个 buff 的描述符的 link 指向第一个 buf 的描述符物理地址.

DMA 驱动支持 chain 模式下 continue 模式传输，目前只支持单 buff.



由于描述符的 link 域链在一起，DMA 硬件永远不可能取到 0xfffff800，因此永远没有 queue done 中断. 使用者需要注意.

3.4.DMA 单包工作模式

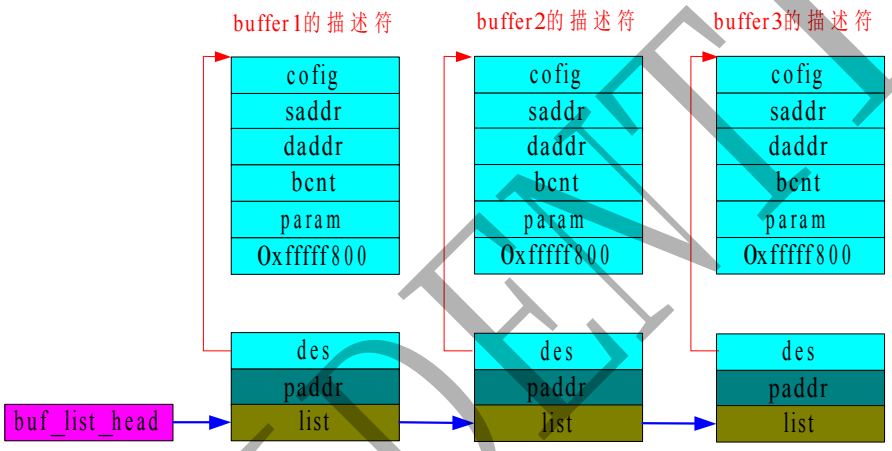
3.4.1. 概念

据上述，多包模式有可能软件上可能来不及单独处理每个包的中断，造成"丢"中断的后果，因此引入单包模式.

(1) 每个包的描述符 link 域均为 0xfffff800. 这样 DMA 传完该包会停下来，由软件启动下一包的传输.
(2) 软件上用链表管理每个包，形成一个队列.
(3) 每个包传完, half/full/queue irq pending 均会被置位.

3.4.2. buffer 队列管理

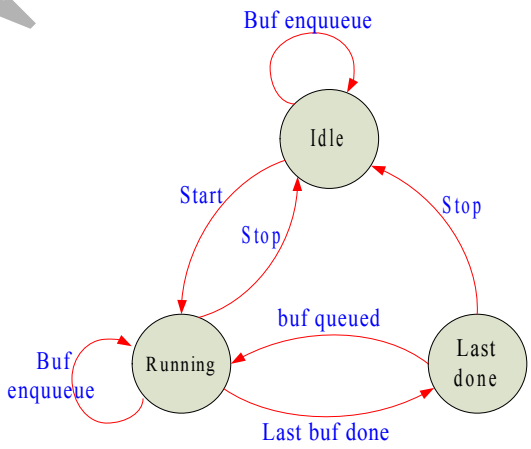
软件上将所有 buffer 的描述符链到一起.



当 buffer1 传完后，由于其描述符的 link 项为 0xfffff800，硬件没有下一个描述符可取，会停下来.
buf_list_head 为通道 buffer 链表，当程序得知 buffer1 传完(通过 queue done irq)后，会查找链上下一个 buffer，即 buffer2，并 start DMA(启动传输).

3.4.3. 软件状态机

单包模式的软件状态机如下:



只有三种状态:
(1) 申请 DMA 通道，初始状态为 idle

(2) 配置 DMA 通道, enqueue 第一个 buffer，状态还是 idle
(3) enqueue 另外一些 buffer，但未启动 DMA，状态还是 idle
(4) start DMA，状态变为 running
(5) buffer 传输过程中，状态还是 running
(6) 一个 buffer 传完, DMA 驱动在 queue done handle 中自动 start 链上另外一个 buffer，状态还是 running
(7) 链上所有 buffer 传完，没有新的 buffer 可以传输，状态变为 last done
(8) 又有新 buffer enqueue 进来，则 enuqeue 函数中自动 start 该 buffer，状态变为 running

以上任何状态下 stop DMA，则状态变为 idle，同时将剩余未传的 buffer 释放掉.

以下几点说明:

(1) 状态为 idle 时，只有 start 能使之启动，变为 running 状态.
(2) 不是每次 start 操作都能改变状态，比如 running 状态下, DMA 自动启动下一个链上 buffer，状态维持 running 不变.
(3) last done 状态主要用于以下情形：若链上所有 buffer 都传完了，最后一个 buffer 的 queue done 中断也处理了. 若有新的 buffer enquue，则必须由 enqueue 函数内部去 start dma. enqueue 函数检测到当前为 last done 状态时，才会 start DMA.
(4) 通过以上分析，以下几种情形会 start DMA:
   a) 初始时，由调用者启动 DMA，状态从 idle 变为 running.
   b) 一个 buffer 传完(done)时，由 queue done handle 函数启动下一个 buffer，状态 running 不变.
   c) 所有 buffer 传完,DMA 通道处于 last done 状态时，由 enqueue 函数启动新 buffer，同时更新状态到 running.

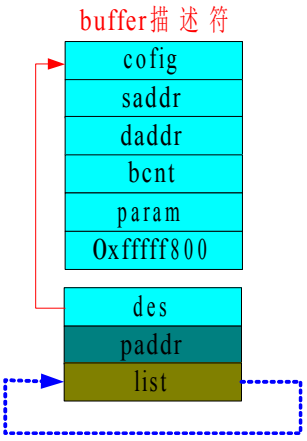
3.4.4. DMA 单包模式下的 continue 模式

某些驱动(比如音频)需要 DMA 在 continue 模式下工作，即 DMA 传完一个包后，自动重新传输该包，以避免每次 buffer done 后 enqueue 操作.

chain 模式下 continue 方式传输数据还是有可能丢中断，因为硬件在传完一包不会停下来，会自动重传.

因此, DMA 驱动增加了 single 模式下的 continue 模式.基本原理是:

(1) 当前仅有一个 buffer 在传.
(2) buffer 传完(queue done 中断产生)后，在 qd handler 中重新 start 这个 buffer.



4. 模块数据结构描述

4.1.dma_channel_t

DMA 通道信息，即申请 DMA 得到的句柄.

```
struct dma_channel_t {
    u32     used;           /* 1 used, 0 unuse */
    u32     id;         /* channel id, 0~15 */
    char            owner[MAX_OWNER_NAME_LEN];      /* dma chnnnel owner name */
    u32     reg_base;   /* regs base addr */
    u32     bconti_mode;    /* cotinue mode */
    u32             irq_spt;  /* channel irq supprot type, used for irq handler only enabled
then can call irq callback */
    struct dma_cb_t         hd_cb;          /* half done call back func */
    struct dma_cb_t         fd_cb;          /* full done call back func */
    struct dma_cb_t         qd_cb;          /* queue done call back func */
    struct dma_op_cb_t      op_cb;          /* dma operation call back func */
    struct des_save_info_t      des_info_save;   /* save the prev buf para, used by
sw_dma_enqueue */
    enum dma_work_mode_e    work_mode;
    union dma_chan_sta_u    state;          /* channel state for chain/single mode */
    spinlock_t          lock;           /* dma channel lock */
    /*
     * for chain mode only
     */
    struct list_head    cur_list;  /* buf list which is being tranferring */
    struct list_head    next_list;/* buf list bkup for next tranfer */
    /*
     * for single mode only
     */
    des_item            *pcur_des;    /* cur buffer which is transferring */
    struct list_head    buf_list_head;
};
```

4.2.cofig_des_t

dma 配置描述符结构.

```
struct cofig_des_t {
    u32     cofig;          /* dma configuration reg */
    u32     saddr;          /* dma src phys addr reg */
    u32     daddr;          /* dma dst phys addr reg */
    u32     bcnt;           /* dma byte cnt reg */
    u32     param;          /* dma param reg */
    struct cofig_des_t *pnext;          /* next descriptor address */
```

```
};
```

4.3.des_item

描述符管理单元.

```
typedef struct __des_item {
    struct cofig_des_t des;     /* descriptor that will be set to hw */
    u32             paddr;      /* physical addr of this __des_item struct */
    struct list_head    list;   /* list node */
}des_item;
```

4.4. des_save_info_t

DMA 描述符配置参数备份. sw_dma_enqueue 不带 config 和 param 参数, 所以需要沿用上一次的 config 和 param 参数.

```
struct des_save_info_t {
    u32     cofig;          /* dma configuration reg */
    u32     param;          /* dma param reg */
    u32     bconti_mode;    /* if dma transfer in continue mode */
};
```

4.5.dma_chan_sta_u

DMA 通道软件状态, chain 和 single 模式分别定义.

```
enum st_md_chain_e {
    DMA_CHAN_STA_IDLE,      /* maybe before start or after stop */
    DMA_CHAN_STA_RUNING,    /* transferring */
    DMA_CHAN_STA_DONE  /* all buffer has transfer done, hw idle, des queue is empty
*/
};

/* dam channel state for single mode */
enum st_md_single_e {
    SINGLE_STA_IDLE,        /* maybe before start or after stop */
    SINGLE_STA_RUNING,      /* transferring */
    SINGLE_STA_LAST_DONE    /* the last buffer has done,
                * in this state, any enqueueing will start dma
                */
};

union dma_chan_sta_u {
    enum st_md_chain_e      st_md_ch;   /* channel state for chain mode */
    enum st_md_single_e     st_md_sg;   /* channel state for single mode */
};
```

4.6.dma_cb_t

dma half/full/queue done 回调函数.

```
typedef u32 (* dma_cb)(dm_hdl_t dma_hdl, void *parg, enum dma_cb_cause_e cause);

struct dma_cb_t {
    dma_cb      func;    /* 函数指针 */
    void        *parg;   /* func 的参数 */
};
```

4.7.dma_op_cb_t

dma operation 回调函数.

```
typedef u32 (* dma_op_cb)(dm_hdl_t dma_hdl, void *parg, enum dma_op_type_e op);

struct dma_op_cb_t {
    dma_op_cb   func;    /* 函数指针 */
    void        *parg;   /* func 的参数 */
};
```

4.8.dma_op_type_e

dma 操作类型.

```
enum dma_op_type_e {
    DMA_OP_START,               /* start dma */
    DMA_OP_PAUSE,               /* pause transferring */
    DMA_OP_RESUME,              /* resume transferring */
    DMA_OP_STOP,                /* stop dma */

    DMA_OP_GET_STATUS,          /* get channel status: idle/busy */
    DMA_OP_GET_CUR_SRC_ADDR,    /* get current src address */
    DMA_OP_GET_CUR_DST_ADDR,    /* get current dst address */
    DMA_OP_GET_BYTECNT_LEFT,    /* get byte cnt left */

    DMA_OP_SET_OP_CB,           /* set operation callback */
    DMA_OP_SET_HD_CB,           /* set half done callback */
    DMA_OP_SET_FD_CB,           /* set full done callback */
    DMA_OP_SET_QD_CB,           /* set queue done callback */
};
```

5. 模块接口描述

5.1.sw_dma_request

原型: dm_hdl_t sw_dma_request(char * name, enum dma_work_mode_e work_mode);
功能: 申请 dma 通道.
参数:

name: dma 通道名，由调用者取，可以为 NULL，但不能与已有的冲突.

work_mode: dma 工作模式，DMA_WORK_MODE_CHAIN 表示 chain 模式，DMA_WORK_MODE_SINGLE 表示 single 模式，其他值无效. 一般用 single 模式.

返回: 成功返回句柄，失败返回 NULL.

5.2.sw_dma_release

原型: u32 sw_dma_release(dm_hdl_t dma_hdl);
功能: 释放 dma 通道
参数:

dma_hdl: dma 通道句柄

返回: 成功返回 0，失败返回出错的行号.

5.3.sw_dma_ctl

原型: u32 sw_dma_ctl(dm_hdl_t dma_hdl, enum dma_op_type_e op, void *parg);
功能: dma 控制函数，用于启动 dma，停止 dma，获取数据传输状态，设置回调函数等.
参数:

dma_hdl: dma 通道句柄

op: 操作类型

```
enum dma_op_type_e {
        DMA_OP_START,                           /* start dma */
        DMA_OP_PAUSE,                           /* pause transferring */
        DMA_OP_RESUME,                          /* resume transferring */
        DMA_OP_STOP,                    /* stop dma */

        DMA_OP_GET_STATUS,      /* get channel status: idle/busy */
        DMA_OP_GET_CUR_SRC_ADDR,    /* get current src address */
        DMA_OP_GET_CUR_DST_ADDR,    /* get current dst address */
        DMA_OP_GET_BYTECNT_LEFT,    /* get byte cnt left */

        DMA_OP_SET_OP_CB,               /* set operation callback */
        DMA_OP_SET_HD_CB,               /* set half done callback */
        DMA_OP_SET_FD_CB,               /* set full done callback */
        DMA_OP_SET_QD_CB,               /* set queue done callback */
};
```

parg: 操作所带参数，不同 op 参数意义不同

返回: 成功返回 0, 失败返回出错的行号.

5.4.sw_dma_config

原型: u32 sw_dma_config(dm_hdl_t dma_hdl, dma_config_t *pcfg, dma_enque_phase_e phase);
功能: 用于启动 dma 之前, 配置 dma 硬件参数, 添加第一个 buffer.
参数:

dma_hdl: dma 通道句柄
pcfg: buffer 配置信息
phase: 添加 buffer 的阶段. 该参数现无意义, 请固定设成 ENQUE_PHASE_NORMAL.

```
enum dma_enque_phase_e {
        ENQUE_PHASE_NORMAL, 一般的添加, 即非以下三种情形.
        ENQUE_PHASE_HD,          dma 的 half done 回调函数中添加
        ENQUE_PHASE_FD,          dma 的 full done 回调函数中添加
        ENQUE_PHASE_QD           dma 的 queue done 回调函数中添加
};
```

返回: 成功返回 0, 失败返回出错的行号.

5.5.sw_dma_enqueue

原型: u32 sw_dma_enqueue(dm_hdl_t dma_hdl, u32 src_addr, u32 dst_addr, u32 byte_cnt,
dma_enque_phase_e phase);
功能: 添加 buffer 到队列.
参数:

dma_hdl: dma 通道句柄
src_addr: 源物理地址
dst_addr: 源物理地址
byte_cnt: 传输字节数
phase: 传输阶段. 该参数现无意义, 请固定设成 ENQUE_PHASE_NORMAL.

返回: 成功返回 0, 失败返回出错的行号.

5.6.sw_dma_getposition

原型: int sw_dma_getposition(dm_hdl_t dma_hdl, u32 *pSrc, u32 *pDst);
功能: 获取当前传输位置信息. 注: 仅音频模块(spdif/i2s/hdmi audio/pcm)用到, 其他模块别用.
参数:

dma_hdl: dma 通道句柄
pSrc: 存放获取的 src addr 寄存器值
pDst: 存放获取的 dst addr 寄存器值

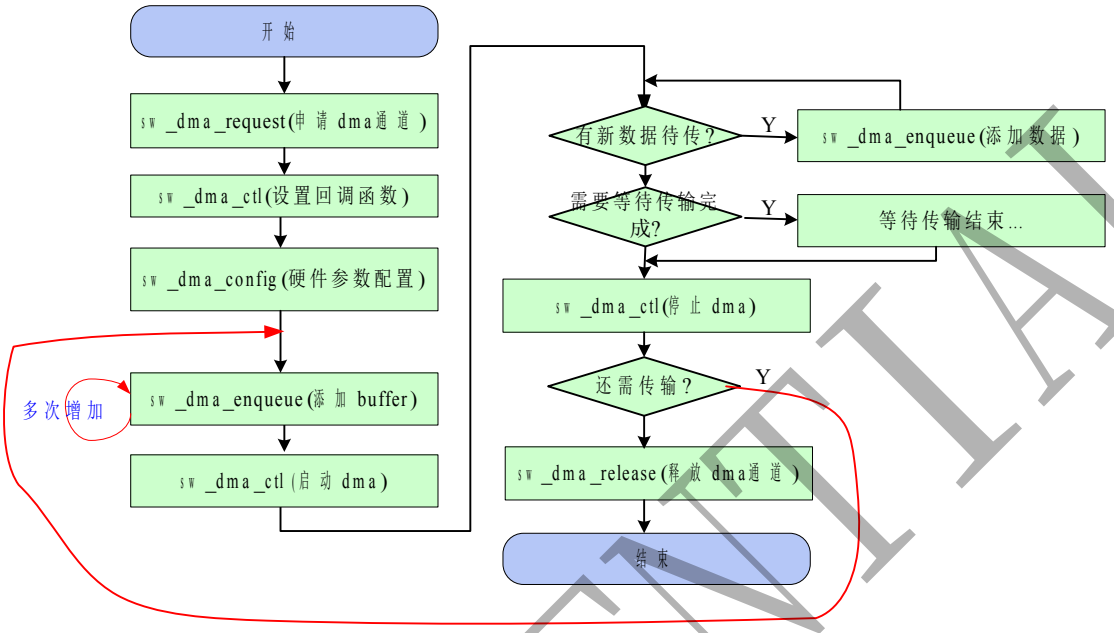
返回: 成功返回 0, 失败返回出错的行号.

5.7.sw_dma_dump_chan

原型: void sw_dma_dump_chan(dm_hdl_t dma_hdl);
功能: 打印通道信息函数. 用于调试.
参数:
dma_hdl: dma 通道句柄
返回: 无.

6. 模块开发 DEMO

6.1. DMA 使用流程图



(1) 申请 dma 通道.

(2) 设置回调函数: hd_cb(des 队列中某个 buffer 传输一半时回调), fd_cb(des 队列中某个 buffer 传输完时回调), qd_cb(des 队列中所有 buffer 传输完时回调)

(3) 配置 dma 参数，如 src drq type(源地址类型), burst length(burst 长度)；添加第一个 buffer.

(4) 若有新 buffer 加进来，则添加.

(5) 启动 dma.

(6) 若有新 buffer 加进来，则添加.

(7) 等待 buffer 传输完成. 或直接 stop(根据需要).

(8) 停止 dma.

(9) 释放 dma 通道.

6.2. demo 程序

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/fcntl.h>
#include <linux/gfp.h>
#include <linux/interrupt.h>
#include <linux/init.h>
#include <linux/ioport.h>
#include <linux/in.h>
#include <linux/string.h>
```

```
#include <linux/delay.h>
#include <linux/errno.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/skbuff.h>
#include <linux/platform_device.h>
#include <linux/dma-mapping.h>
#include <linux/slab.h>
#include <asm/io.h>
#include <asm/pgtable.h>
#include <asm/dma.h>
#include <linux/kthread.h>
#include <linux/delay.h>
#include <asm/dma-mapping.h>
#include <linux/wait.h>
#include <linux/random.h>

#include <mach/dma.h>

/*
 * dma test case id
 */
enum dma_test_case_e {
    DTC_1T_MEM_2_MEM,   /* dma test case one-thread from memory to memory */
    DTC_SINGLE_MODE,    /* dma test case for single mode */
    DTC_SINGLE_CONT_MODE,  /* dma test case for single mode & continue mode */
    DTC_2T_MEM_2_MEM,   /* dma test case two-thread from memory to memory,
                * memory range should not be conflict, eg: thread one
                * from memory-A to memory-B, thread two from C to D.
                */
    DTC_1TM2M_MANY_ENQ,    /* dma test case one-thread memory to memory,
many enqueue */
    DTC_1TM2M_CONTI_MOD,   /* dma test case one-thread memory to memory,
continue mode */
    DTC_1T_ENQ_AFT_DONE,   /* check if dma driver can continue transfer new
enqueued buffer after done */
    DTC_1T_CMD_STOP,    /* stop when dma running */
    DTC_MAX
};

/* total length and each transfer length */
#define DTC_1T_TOTAL_LEN  SIZE_512K
#define DTC_1T_ONE_LEN          SIZE_4K
```

```
/* src/dst start address */
static u32 g_src_addr = 0, g_dst_addr = 0;
/* cur package index */
static atomic_t g_acur_cnt = ATOMIC_INIT(0);

/**
 * __cb_qd_1t_mem_2_mem - queue done callback for case DTC_1T_MEM_2_MEM
 * @dma_hdl:     dma handle
 * @parg:     args registerd with cb function
 * @cause:   case for this cb, DMA_CB_OK means data transfer OK,
 *          DMA_CB_ABORT means stopped before transfer complete
 *
 * Returns 0 if sucess, the err line number if failed.
 */
u32 __cb_qd_1t_mem_2_mem(dm_hdl_t dma_hdl, void *parg, enum dma_cb_cause_e
cause)
{
      u32       uret = 0;
      u32 ucur_saddr = 0, ucur_daddr = 0;
      u32 uloop_cnt = DTC_1T_TOTAL_LEN / DTC_1T_ONE_LEN;
      u32       ucur_cnt = 0;

      pr_info("%s: called!\n", __func__);
      switch(cause) {
      case DMA_CB_OK:
            pr_info("%s: DMA_CB_OK!\n", __func__);
            /* enqueue if not done */
            ucur_cnt = atomic_add_return(1, &g_acur_cnt);
            if(ucur_cnt < uloop_cnt) {
                  printk("%s, line %d\n", __func__, __LINE__);
                  /* NOTE: fatal err, when read here, g_acur_cnt has changed by other place,
2012-12-2 */
                  //ucur_saddr      =      g_src_addr      +      atomic_read(&g_acur_cnt)      *
DTC_1T_ONE_LEN;
                  ucur_saddr = g_src_addr + ucur_cnt * DTC_1T_ONE_LEN;
                  ucur_daddr = g_dst_addr + ucur_cnt * DTC_1T_ONE_LEN;
                  if(0      !=      sw_dma_enqueue(dma_hdl,      ucur_saddr,      ucur_daddr,
DTC_1T_ONE_LEN, ENQUE_PHASE_QD))
                        printk("%s err, line %d\n", __func__, __LINE__);
#if 0
            /*
             * we have complete enqueueing, but not means it's the last qd irq,
             * in test, we found sometimes never meet if(ucur_cnt == uloop_cnt...
             * that is, enqueue complete during hd/fd callback.
```

```
             */
            } else if(ucur_cnt == uloop_cnt){
                printk("%s, line %d\n", __func__, __LINE__);
                sw_dma_dump_chan(dma_hdl); /* for debug */

                /* maybe it's the last irq; or next will be the last irq, need think about */
                atomic_set(&g_adma_done, 1);
                wake_up_interruptible(&g_dtc_queue[DTC_1T_MEM_2_MEM]);
#endif
            } else {
                printk("%s, line %d\n", __func__, __LINE__);
                sw_dma_dump_chan(dma_hdl); /* for debug */

                /* maybe it's the last irq */
                atomic_set(&g_adma_done, 1);
                wake_up_interruptible(&g_dtc_queue[DTC_1T_MEM_2_MEM]);
            }
            break;
        case DMA_CB_ABORT:
            pr_info("%s: DMA_CB_ABORT!\n", __func__);
            break;
        default:
            uret = __LINE__;
            goto end;
        }

end:
        if(0 != uret)
            pr_err("%s err, line %d!\n", __func__, uret);
        return uret;
}

/**
 * __cb_fd_1t_mem_2_mem - full done callback for case DTC_1T_MEM_2_MEM
 * @dma_hdl:     dma handle
 * @parg:     args registerd with cb function
 * @cause:   case for this cb, DMA_CB_OK means data transfer OK,
 *           DMA_CB_ABORT means stopped before transfer complete
 *
 * Returns 0 if sucess, the err line number if failed.
 */
u32 __cb_fd_1t_mem_2_mem(dm_hdl_t dma_hdl, void *parg, enum dma_cb_cause_e
cause)
{
```

```
    u32       uret = 0;
    u32 ucur_saddr = 0, ucur_daddr = 0;
    u32 uloop_cnt = DTC_1T_TOTAL_LEN / DTC_1T_ONE_LEN;
    u32       ucur_cnt = 0;

    pr_info("%s: called!\n", __func__);
    switch(cause) {
    case DMA_CB_OK:
        pr_info("%s: DMA_CB_OK!\n", __func__);
        /* enqueue if not done */
        ucur_cnt = atomic_add_return(1, &g_acur_cnt);
        if(ucur_cnt < uloop_cnt){
            printk("%s, line %d\n", __func__, __LINE__);
            ucur_saddr = g_src_addr + ucur_cnt * DTC_1T_ONE_LEN;
            ucur_daddr = g_dst_addr + ucur_cnt * DTC_1T_ONE_LEN;
            if(0      !=      sw_dma_enqueue(dma_hdl,      ucur_saddr,      ucur_daddr,
DTC_1T_ONE_LEN, ENQUE_PHASE_FD))
                printk("%s err, line %d\n", __func__, __LINE__);
        } else /* do nothing */
            printk("%s, line %d\n", __func__, __LINE__);
        break;
    case DMA_CB_ABORT:
        pr_info("%s: DMA_CB_ABORT!\n", __func__);
        break;
    default:
        uret = __LINE__;
        goto end;
    }

end:
    if(0 != uret)
        pr_err("%s err, line %d!\n", __func__, uret);
    return uret;
}

/**
 * __cb_hd_1t_mem_2_mem - half done callback for case DTC_1T_MEM_2_MEM
 * @dma_hdl:    dma handle
 * @parg:    args registerd with cb function
 * @cause:   case for this cb, DMA_CB_OK means data transfer OK,
 *           DMA_CB_ABORT means stopped before transfer complete
 *
 * Returns 0 if sucess, the err line number if failed.
 */
```

```
u32 __cb_hd_1t_mem_2_mem(dm_hdl_t dma_hdl, void *parg, enum dma_cb_cause_e cause)
{
	u32	uret = 0;

	pr_info("%s: called!\n", __func__);
	switch(cause) {
	case DMA_CB_OK:
		pr_info("%s: DMA_CB_OK!\n", __func__);
		break;
	case DMA_CB_ABORT:
		pr_info("%s: DMA_CB_ABORT!\n", __func__);
		break;
	default:
		uret = __LINE__;
		goto end;
	}

end:
	if(0 != uret)
		pr_err("%s err, line %d!\n", __func__, uret);
	return uret;
}

/**
 * __cb_op_1t_mem_2_mem - operation callback for case DTC_1T_MEM_2_MEM
 * @dma_hdl:	dma handle
 * @parg:	args registerd with cb function
 * @op:		the operation type
 *
 * Returns 0 if sucess, the err line number if failed.
 */
u32 __cb_op_1t_mem_2_mem(dm_hdl_t dma_hdl, void *parg, enum dma_op_type_e op)
{
	pr_info("%s: called!\n", __func__);

	switch(op) {
	case DMA_OP_START:
		pr_info("%s: op DMA_OP_START!\n", __func__);
		atomic_set(&g_adma_done, 0);
		break;
	case DMA_OP_STOP:
		pr_info("%s: op DMA_OP_STOP!\n", __func__);
		break;
```

```
        case DMA_OP_SET_HD_CB:
            pr_info("%s: op DMA_OP_SET_HD_CB!\n", __func__);
            break;
        case DMA_OP_SET_FD_CB:
            pr_info("%s: op DMA_OP_SET_FD_CB!\n", __func__);
            break;
        case DMA_OP_SET_OP_CB:
            pr_info("%s: op DMA_OP_SET_OP_CB!\n", __func__);
            break;
        default:
            printk("%s, line %d\n", __func__, __LINE__);
            break;
        }

        return 0;
}

/**
 *    __waitdone_1t_mem_2_mem    -    wait    dma    transfer    function    for    case
DTC_1T_MEM_2_MEM
 *
 * Returns 0 if sucess, the err line number if failed.
 */
u32 __waitdone_1t_mem_2_mem(void)
{
        long        ret = 0;
        long        timeout = 10 * HZ; /* 10s */

        /* wait dma done */
        ret = wait_event_interruptible_timeout(g_dtc_queue[DTC_1T_MEM_2_MEM], \
            atomic_read(&g_adma_done)== 1, timeout);
        /* reset dma done flag to 0 */
        atomic_set(&g_adma_done, 0);

        if(-ERESTARTSYS == ret) {
            pr_info("%s success!\n", __func__);
            return 0;
        } else if(0 == ret) {
            pr_info("%s err, time out!\n", __func__);
            return __LINE__;
        } else {
            pr_info("%s success with condition match, ret %d!\n", __func__, (int)ret);
            return 0;
        }
```

```
}

/**
 * __dtc_1t_mem_2_mem - dma test case one-thread from memory to memory
 *
 * Returns 0 if success, the err line number if failed.
 */
u32 __dtc_1t_mem_2_mem(void)
{
	u32		uret = 0;
	void		*src_vaddr = NULL, *dst_vaddr = NULL;
	u32		src_paddr = 0, dst_paddr = 0;
	dm_hdl_t dma_hdl = (dm_hdl_t)NULL;
	struct dma_cb_t done_cb;
	struct dma_op_cb_t op_cb;
	struct dma_config_t dma_config;

	pr_info("%s enter\n", __func__);

	/* prepare the buffer and data */
	src_vaddr = dma_alloc_coherent(NULL, DTC_1T_TOTAL_LEN, (dma_addr_t
*)&src_paddr, GFP_KERNEL);
	if(NULL == src_vaddr) {
		uret = __LINE__;
		goto end;
	}
	pr_info("%s: src_vaddr 0x%08x, src_paddr 0x%08x\n", __func__, (u32)src_vaddr,
src_paddr);
	dst_vaddr = dma_alloc_coherent(NULL, DTC_1T_TOTAL_LEN, (dma_addr_t
*)&dst_paddr, GFP_KERNEL);
	if(NULL == dst_vaddr) {
		uret = __LINE__;
		goto end;
	}
	pr_info("%s: dst_vaddr 0x%08x, dst_paddr 0x%08x\n", __func__, (u32)dst_vaddr,
dst_paddr);

	/* init src buffer */
	get_random_bytes(src_vaddr, DTC_1T_TOTAL_LEN);
	memset(dst_vaddr, 0x54, DTC_1T_TOTAL_LEN);

	/* init loop para */
	atomic_set(&g_acur_cnt, 0);
	g_src_addr = src_paddr;
```

```
    g_dst_addr = dst_paddr;

    /* request dma channel */
    dma_hdl = sw_dma_request("m2m_dma", DMA_WORK_MODE_CHAIN);
    if(NULL == dma_hdl) {
        uret = __LINE__;
        goto end;
    }
    pr_info("%s: sw_dma_request success, dma_hdl 0x%08x\n", __func__,
(u32)dma_hdl);

    /* set queue done callback */
    memset(&done_cb, 0, sizeof(done_cb));
    memset(&op_cb, 0, sizeof(op_cb));
    done_cb.func = __cb_qd_1t_mem_2_mem;
    done_cb.parg = NULL;
    if(0 != sw_dma_ctl(dma_hdl, DMA_OP_SET_QD_CB, (void *)&done_cb)) {
        uret = __LINE__;
        goto end;
    }
    pr_info("%s: set queuedone_cb success\n", __func__);
    /* set full done callback */
    done_cb.func = __cb_fd_1t_mem_2_mem;
    done_cb.parg = NULL;
    if(0 != sw_dma_ctl(dma_hdl, DMA_OP_SET_FD_CB, (void *)&done_cb)) {
        uret = __LINE__;
        goto end;
    }
    pr_info("%s: set fulldone_cb success\n", __func__);
    /* set half done callback */
    done_cb.func = __cb_hd_1t_mem_2_mem;
    done_cb.parg = NULL;
    if(0 != sw_dma_ctl(dma_hdl, DMA_OP_SET_HD_CB, (void *)&done_cb)) {
        uret = __LINE__;
        goto end;
    }
    pr_info("%s: set halfdone_cb success\n", __func__);
    /* set operation done callback */
    op_cb.func = __cb_op_1t_mem_2_mem;
    op_cb.parg = NULL;
    if(0 != sw_dma_ctl(dma_hdl, DMA_OP_SET_OP_CB, (void *)&op_cb)) {
        uret = __LINE__;
        goto end;
    }
```

```
    pr_info("%s: set op_cb success\n", __func__);

    /* set config para */
    memset(&dma_config, 0, sizeof(dma_config));
    dma_config.xfer_type       = DMAXFER_D_BWORD_S_BWORD;
    dma_config.address_type = DMAADDRT_D_LN_S_LN;
    dma_config.para  = 0;
    dma_config.irq_spt     = CHAN_IRQ_HD | CHAN_IRQ_FD | CHAN_IRQ_QD;
    dma_config.src_addr  = src_paddr;
    dma_config.dst_addr  = dst_paddr;
    dma_config.byte_cnt   = DTC_1T_ONE_LEN;
    //dma_config.conti_mode = 1;
    dma_config.bconti_mode = false;
    dma_config.src_drq_type = DRQSRC_SDRAM;
    dma_config.dst_drq_type = DRQDST_SDRAM;
    /* enqueue buffer */
    if(0 != sw_dma_config(dma_hdl, &dma_config, ENQUE_PHASE_NORMAL)) {
        uret = __LINE__;
        goto end;
    }
    pr_info("%s: sw_dma_config success\n", __func__);
    /* dump chain */
    sw_dma_dump_chan(dma_hdl);

    /* start dma */
    if(0 != sw_dma_ctl(dma_hdl, DMA_OP_START, NULL)) {
        uret = __LINE__;
        goto end;
    }

    /* enqueue other buffer, with callback enqueue simutanously */
    {
        u32      ucur_cnt = 0, ucur_saddr = 0, ucur_daddr = 0;
        u32 uloop_cnt = DTC_1T_TOTAL_LEN / DTC_1T_ONE_LEN;
        while((ucur_cnt = atomic_add_return(1, &g_acur_cnt)) < uloop_cnt) {
            ucur_saddr = g_src_addr + ucur_cnt * DTC_1T_ONE_LEN;
            ucur_daddr = g_dst_addr + ucur_cnt * DTC_1T_ONE_LEN;
            if(0      !=      sw_dma_enqueue(dma_hdl,      ucur_saddr,      ucur_daddr,
DTC_1T_ONE_LEN, ENQUE_PHASE_NORMAL))
                printk("%s err, line %d\n", __func__, __LINE__);
        }
    }
    pr_info("%s, line %d\n", __func__, __LINE__);
```

```
    /* wait dma done */
    if(0 != __waitdone_1t_mem_2_mem()) {
        uret = __LINE__;
        goto end;
    }
    pr_info("%s: __waitdone_1t_mem_2_mem sucess\n", __func__);

    /*
     * NOTE: must sleep here, becase when __waitdone_1t_mem_2_mem return, buffer
enqueue complete, but
     * data might not transfer complete, 2012-11-14
     */
    msleep(1000);

    /* check if data ok */
    if(0 == memcmp(src_vaddr, dst_vaddr, DTC_1T_TOTAL_LEN))
        pr_info("%s: data check ok!\n", __func__);
    else {
        pr_err("%s: data check err!\n", __func__);
        uret = __LINE__; /* return err */
        goto end;
    }

    /* stop and release dma channel */
    if(0 != sw_dma_ctl(dma_hdl, DMA_OP_STOP, NULL)) {
        uret = __LINE__;
        goto end;
    }
    pr_info("%s: sw_dma_stop success\n", __func__);
    if(0 != sw_dma_release(dma_hdl)) {
        uret = __LINE__;
        goto end;
    }
    dma_hdl = (dm_hdl_t)NULL;
    pr_info("%s: sw_dma_release success\n", __func__);

end:
    if(0 != uret)
        pr_err("%s err, line %d!\n", __func__, uret); /* print err line */
    else
        pr_info("%s, success!\n", __func__);

    /* stop and free dma channel, if need */
    if((dm_hdl_t)NULL != dma_hdl) {
```

```
            pr_err("%s, stop and release dma handle now!\n", __func__);
            if(0 != sw_dma_ctl(dma_hdl, DMA_OP_STOP, NULL))
                pr_err("%s err, line %d!\n", __func__, __LINE__);
            if(0 != sw_dma_release(dma_hdl))
                pr_err("%s err, line %d!\n", __func__, __LINE__);
        }
        pr_err("%s, line %d!\n", __func__, __LINE__);

        /* free dma memory */
        if(NULL != src_vaddr)
            dma_free_coherent(NULL, DTC_1T_TOTAL_LEN, src_vaddr, src_paddr);
        if(NULL != dst_vaddr)
            dma_free_coherent(NULL, DTC_1T_TOTAL_LEN, dst_vaddr, dst_paddr);

        pr_err("%s, end!\n", __func__);
        return uret;
}
```

7. Android 系统支持

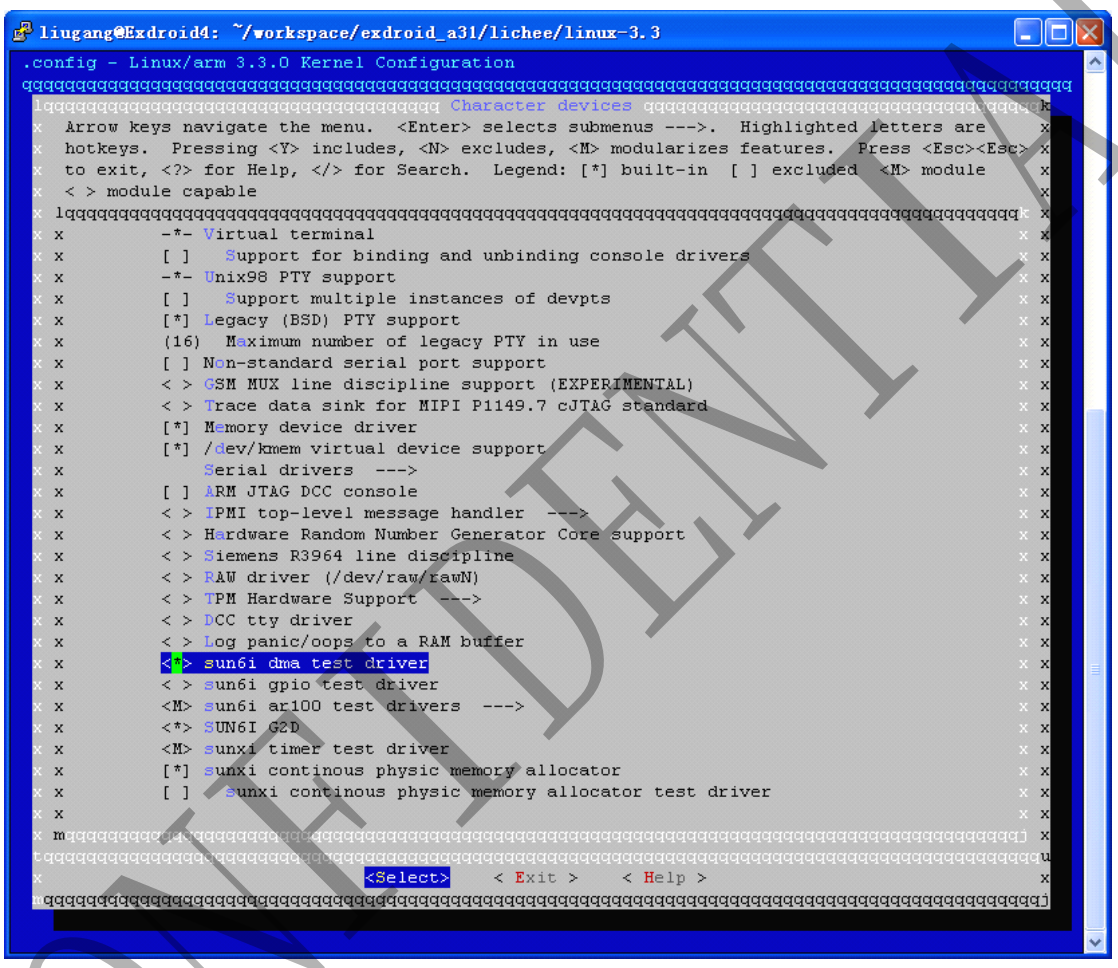
dma 属 linux 内核模块， 和 android 无直接关系.

8. 模块调试

dma 是 buildin 的模块，不用加载，调试方法是，在 menuconfig 中选择 dma_test 驱动，设置测试用例，然后编译 linux 镜像启动看打印．若打印 success 表明用例执行成功；打印 fail/err 表明失败．

menuconfig 的配置:

device drivers -> character devices -> sun6i dma test driver:



测试用例选择:

修改 drivers/char/dma_test/sun6i_dma_test.c 的 g_cur_test_case 变量:

```
static enum dma_test_case_e g_cur_test_case = DTC_SINGLE_MODE;

enum dma_test_case_e {
    DTC_1T_MEM_2_MEM,    /* dma test case one-thread from memory to memory */
    DTC_SINGLE_MODE,     /* dma test case for single mode */
    DTC_SINGLE_CONT_MODE, /* dma test case for single mode & continue mode */
    DTC_2T_MEM_2_MEM,    /* dma test case two-thread from memory to memory,
                          * memory range should not be conflict, eg: thread one
                          * from memory-A to memory-B, thread two from C to D.
```

```
                    */
    DTC_1TM2M_MANY_ENQ,    /* dma test case one-thread memory to memory, many enqueue */
    DTC_1TM2M_CONTI_MOD,   /* dma test case one-thread memory to memory, continue mode */
    DTC_1T_ENQ_AFT_DONE,   /* check if dma driver can continue transfer new enqueued buffer after done */
    DTC_1T_CMD_STOP,       /* stop when dma running */
    DTC_MAX
};
```

(1) DTC_SINGLE_MODE: 测试 dma 模块在 single 模式下是否能正常工作.
(2) DTC_SINGLE_CONT_MODE: 测试 dma 模块在 single 模式下, continue 模式是否正常.
(3) DTC_1T_MEM_2_MEM: 测试单线程通过 dma 从 memory 到 memory 拷贝数据情形. 最简单的情形.
(4) DTC_2T_MEM_2_MEM: 测试两个线程同时通过 dma 从 memory 到 memory 拷贝数据情形.
(5) DTC_1TM2M_MANY_ENQ: 测试描述符空间的动态申请和释放是否正常，有没有内存泄漏.
(6) DTC_1TM2M_CONTI_MOD: 测试单线程通过 dma 从 memory 到 memory 拷贝数据情形. continue mode 的情形.
(7) DTC_1T_ENQ_AFT_DONE: 测试单线程通过 dma 从 memory 到 memory 拷贝数据情形. 在中途传完所有 buffer 后，再添加新的 buffer，看 dma 模块能否自动续传.
(8) DTC_1T_CMD_STOP: 测试单线程通过 dma 从 memory 到 memory 拷贝数据情形. 测试 stop 命令时，能否将剩余的描述符空间释放.

9. 总结

DMA 驱动主要用来统一管理系统的 DMA 资源，使用之前要先申请，用完释放，以便别的模块用.

支持两种使用模式, chain 模式和 single 模式, single 模式下硬件一次传一个包, chain 模式可传多个包. 虽然 chain 模式效率更高，但可能软件来不及单步处理每个包的中断，因此目前主要用 single 模式.

Chain 模式采用两个队列管理 buf，当前队列和等待队列，正在传输时，新的包将添加到等待队列.