全 志 科 技
Allwinner Technology

# A20 DMA 开发说明

**V1.0**

**2013-03-15**

# Revision History

| Version | Date | Changes compared to previous issue |
|---|---|---|
| v1.0 | 2013-03-15 | 初建版本 |
| | | |
| | | |
| | | |

# 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 A20 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．

A20 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．

A20 DMA 模块包含 16 个独立通道，分别有 8 个 dedicate 通道和 8 个 normal 通道．

## 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．

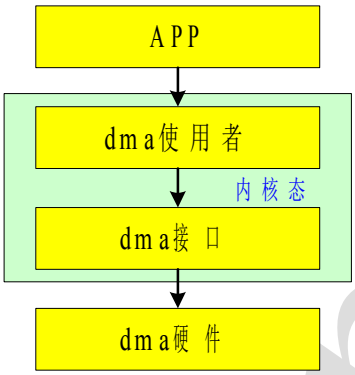
## 2.4. 源码结构介绍

DMA 驱动代码在\linux-3.3\arch\arm\mach-sun7i\dma 下:

DMA 导出接口在\linux-3.3\arch\arm\mach-sun7i\include\mach\下:

# 3. 模块体系结构描述

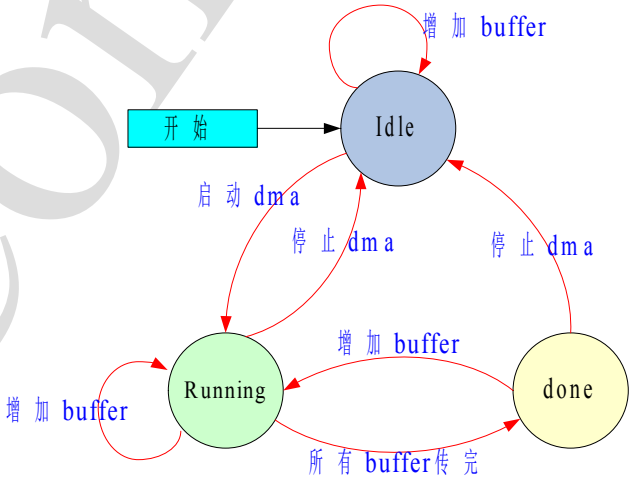
## 3.1. DMA 驱动架构图




(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.c: dma 模块初始化, dma 中断处理.
(2) Dma_core.c: 核心实现
(3) Dma_interface.c: 导出接口
(4) Dma_csp.c: dma 硬件操作函数.

## 3.2. DMA 软件状态




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.

# 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. chan_state_e

DMA 通道状态.

```
typedef enum {
    CHAN_STA_IDLE,      /* maybe before start or after stop */
    CHAN_STA_RUNING,  /* transferring */
    CHAN_STA_LAST_DONE /* the last buffer has done, in this state, sw_dma_enqueue will start dma */
}chan_state_e;
```

## 4.5. 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 */
};
```

## 4.6. 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.

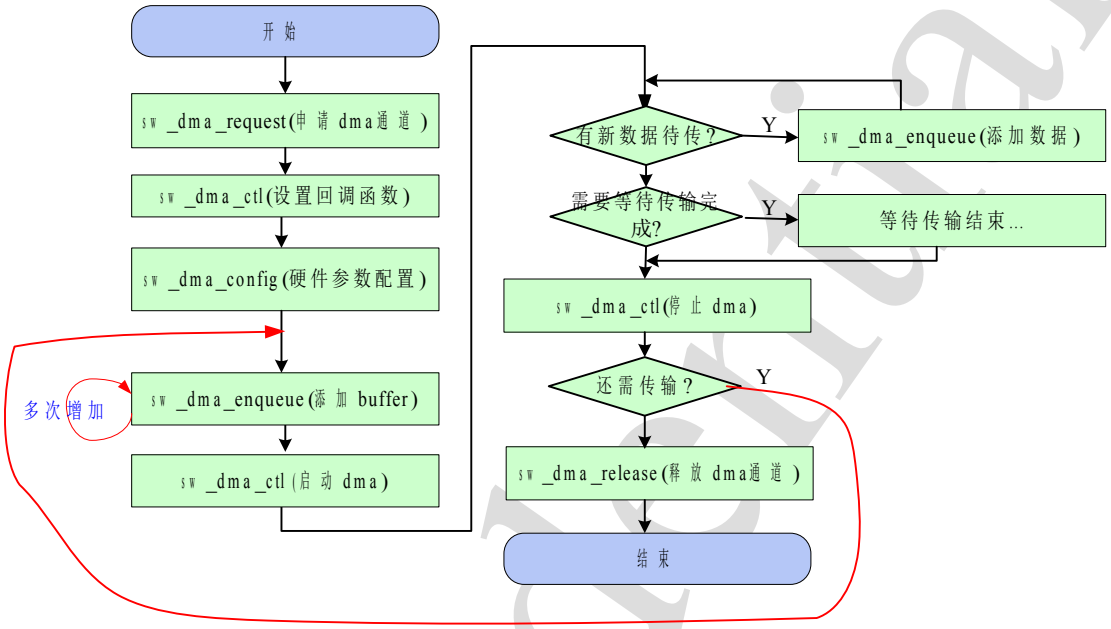
参数:

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

### 6.2.1. test_case_normal.c

```
#include "sun7i_dma_test.h"

/* src/dst start address */
static u32 g_src_addr = 0, g_dst_addr = 0;
```

```
/* cur buf index */
static atomic_t g_acur_cnt = ATOMIC_INIT(0);

/**
 * __cb_fd_normal - full done callback for case DTC_NORMAL
 * @dma_hdl:  dma handle
 * @parg:  args registerd with cb function
 *
 * Returns 0 if sucess, the err line number if failed.
 */
void __cb_fd_normal(dma_hdl_t dma_hdl, void *parg)
{
    u32     uret = 0;
    u32ucur_saddr = 0, ucur_daddr = 0;
    u32uloop_cnt = TOTAL_LEN_NORMAL / ONE_LEN_NORMAL;
    u32     ucur_cnt = 0;

    pr_info("%s: called!\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)    *
ONE_LEN_NORMAL;
        ucur_saddr = g_src_addr + ucur_cnt * ONE_LEN_NORMAL;
        ucur_daddr = g_dst_addr + ucur_cnt * ONE_LEN_NORMAL;
        if(0    !=    sw_dma_enqueue(dma_hdl,    ucur_saddr,    ucur_daddr,
ONE_LEN_NORMAL))
            printk("%s err, line %d\n", __func__, __LINE__);
    } else { /* buf enqueue complete */
        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_NORMAL]);
    }

    if(0 != uret)
```

```
        pr_err("%s err, line %d!\n", __func__, uret);
}

/**
 * __cb_hd_normal - half done callback for case DTC_NORMAL
 * @dma_hdl:  dma handle
 * @parg:  args registerd with cb function
 *
 * Returns 0 if sucess, the err line number if failed.
 */
void __cb_hd_normal(dma_hdl_t dma_hdl, void *parg)
{
    u32     uret = 0;
    u32ucur_saddr = 0, ucur_daddr = 0;
    u32uloop_cnt = TOTAL_LEN_NORMAL / ONE_LEN_NORMAL;
    u32     ucur_cnt = 0;

    pr_info("%s: called!\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)    *
ONE_LEN_NORMAL;
        ucur_saddr = g_src_addr + ucur_cnt * ONE_LEN_NORMAL;
        ucur_daddr = g_dst_addr + ucur_cnt * ONE_LEN_NORMAL;
        if(0    !=    sw_dma_enqueue(dma_hdl,    ucur_saddr,    ucur_daddr,
ONE_LEN_NORMAL))
            printk("%s err, line %d\n", __func__, __LINE__);
    }

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

/**
 * __waitdone_normal - wait dma transfer function for case DTC_NORMAL
 *
 * Returns 0 if sucess, the err line number if failed.
```

```
 */
u32 __waitdone_normal(void)
{
    long    ret = 0;
    long    timeout = 2 * HZ;

    /* wait dma done */
    ret = wait_event_interruptible_timeout(g_dtc_queue[DTC_NORMAL], \
        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;
    }
}

u32 __dtc_normal(void)
{
    u32     uret = 0/* , tmp = 0 */;
    void    *src_vaddr = NULL, *dst_vaddr = NULL;
    u32     src_paddr = 0, dst_paddr = 0;
    dma_hdl_t dma_hdl = (dma_hdl_t)NULL;
    dma_cb_t done_cb;
    dma_config_t dma_config;

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

    /* prepare the buffer and data */
    src_vaddr = dma_alloc_coherent(NULL, TOTAL_LEN_NORMAL,
(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, TOTAL_LEN_NORMAL,
(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, TOTAL_LEN_NORMAL);
    memset(dst_vaddr, 0x54, TOTAL_LEN_NORMAL);

    /* 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", CHAN_NORAML);
    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 full done callback */
    done_cb.func = __cb_fd_normal;
    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_normal;
    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__);

        /* config para */
        memset(&dma_config, 0, sizeof(dma_config));
        dma_config.xfer_type.src_data_width    = DATA_WIDTH_32BIT;
        dma_config.xfer_type.src_bst_len   = DATA_BRST_4;
        dma_config.xfer_type.dst_data_width    = DATA_WIDTH_32BIT;
        dma_config.xfer_type.dst_bst_len   = DATA_BRST_4;
        dma_config.address_type.src_addr_mode  =
NDMA_ADDR_INCREMENT;
        dma_config.address_type.dst_addr_mode  =
NDMA_ADDR_INCREMENT;
        dma_config.src_drq_type    = N_SRC_SDRAM;
        dma_config.dst_drq_type    = N_DST_SDRAM;
        dma_config.bconti_mode         = false;
        dma_config.irq_spt             = CHAN_IRQ_HD | CHAN_IRQ_FD;
        if(0 != sw_dma_config(dma_hdl, &dma_config)) {
            uret = __LINE__;
            goto end;
        }
        pr_info("%s: sw_dma_config success\n", __func__);

#if 0 /* add or not add, both ok, 2013-2-28 21:08 */
        /* set src/dst secu */
        tmp = SRC_SECU_DST_SECU;
        if(0 != sw_dma_ctl(dma_hdl, DMA_OP_SET_SECURITY, &tmp)) {
            uret = __LINE__;
            goto end;
        }
        pr_info("%s: DMA_OP_SET_SECURITY success\n", __func__);

        /* set wait state, ndma only */
        {
            u32 state = 0; /* 0~7, from spec */
            if(0 != sw_dma_ctl(dma_hdl, DMA_OP_SET_WAIT_STATE, &state)) {
                uret = __LINE__;
                goto end;
            }
            pr_info("%s: DMA_OP_SET_WAIT_STATE success\n", __func__);
```

```
    }
#endif

#if 0
    /* set para reg, ddma only */
    {
        dma_para_t para;
        para.src_blk_sz      = 0;
        para.src_wait_cyc    = 0;
        para.dst_blk_sz      = 0;
        para.dst_wait_cyc    = 0;
        if(0 != sw_dma_ctl(dma_hdl, DMA_OP_SET_PARA_REG, &tmp)) {
            uret = __LINE__;
            goto end;
        }
        pr_info("%s: DMA_OP_SET_PARA_REG success\n", __func__);
    }
#endif

    /* enqueue first buf */
    if(0      !=      sw_dma_enqueue(dma_hdl,      g_src_addr,      g_dst_addr,
ONE_LEN_NORMAL)) {
        uret = __LINE__;
        goto end;
    }
    pr_info("%s: sw_dma_enqueue first buf 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;
        u32uloop_cnt = TOTAL_LEN_NORMAL / ONE_LEN_NORMAL;
        while((ucur_cnt = atomic_add_return(1, &g_acur_cnt)) < uloop_cnt) {
            ucur_saddr = g_src_addr + ucur_cnt * ONE_LEN_NORMAL;
```

```
                ucur_daddr = g_dst_addr + ucur_cnt * ONE_LEN_NORMAL;
                if(0 != sw_dma_enqueue(dma_hdl, ucur_saddr, ucur_daddr,
ONE_LEN_NORMAL))
                    printk("%s err, line %d\n", __func__, __LINE__);
            }
        }
        pr_info("%s, line %d\n", __func__, __LINE__);

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

        /*
         * NOTE: must sleep here, becase when __waitdone_normal return, buffer
enqueue complete, but
         * data might not transfer complete, 2012-11-14
         */
        msleep(1200);

        /* check if data ok */
        if(0 == memcmp(src_vaddr, dst_vaddr, TOTAL_LEN_NORMAL))
            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 = (dma_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((dma_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_info("%s, line %d!\n", __func__, __LINE__);

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

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

### 6.2.2. test_case_normal.h

```
#ifndef __TEST_CASE_NORMAL
#define __TEST_CASE_NORMAL

/* total length and one buf length */
#define TOTAL_LEN_NORMAL      SZ_128K
#define ONE_LEN_NORMAL        SZ_8K

u32 __dtc_normal(void);
u32 __dtc_normal_conti(void);
```

```
u32 __dtc_app_cb_eque(void);
u32 __dtc_case_enq_aftdone(void);


#endif /* __TEST_CASE_NORMAL */
```

### 6.2.3. sun7i_dma_test.h

```
#ifndef __SUN7I_DMA_TEST_H
#define __SUN7I_DMA_TEST_H

#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>
```

```
#include "test_case_normal.h"

/*
 * dma test case id
 */
enum dma_test_case_e {
    DTC_NORMAL = 0,          /* case for normal channel */
    DTC_NORMAL_CONT_MODE,    /* case for normal channel continue mode */
    DTC_DEDICATE,        /* case for dedicate channel */
    DTC_DEDICATE_CONT_MODE,  /* case for dedicate channel continue mode */
    /*
     * for dedicate below
     */
    DTC_ENQ_AFT_DONE,    /* enqueued buffer after dma last done, to see if can cotinue auto start */
    DTC_MANY_ENQ,            /* many buffer enqueued, function test */
    DTC_CMD_STOP,       /* stop when dma running */
    DTC_M2M_TWO_THREAD,    /* two-thread run simutalously, pressure test and memory leak test */
    DTC_MAX
};

extern wait_queue_head_t g_dtc_queue[];
extern atomic_t g_adma_done;

#endif /* __SUN7I_DMA_TEST_H */
```

### 6.2.4. sun7i_dma_test.c

```
#include "sun7i_dma_test.h"

/* wait dma done queue, used for wait dma done */
wait_queue_head_tg_dtc_queue[DTC_MAX];
atomic_t          g_adma_done = ATOMIC_INIT(0);  /* dma done flag */

const char *case_name[] = {
    "DTC_NORMAL",
    "DTC_NORMAL_CONT_MODE",
```

```
    "DTC_DEDICATE",
    "DTC_DEDICATE_CONT_MODE",
    "DTC_ENQ_AFT_DONE",
    "DTC_MANY_ENQ",
    "DTC_CMD_STOP",
    "DTC_M2M_TWO_THREAD",
};

/**
 * __dma_test_init_waitqueue - init dma wait queue
 */
static void __dma_test_init_waitqueue(void)
{
    u32 i = (u32)DTC_MAX;

    while(i--)
        init_waitqueue_head(&g_dtc_queue[i]);
}

static int dma_test_main(int id)
{
    enum dma_test_case_e cur_test = (enum dma_test_case_e)id;
    u32 ret = 0;

    switch(cur_test) {
    case DTC_NORMAL:
        ret = __dtc_normal();
        break;
    case DTC_NORMAL_CONT_MODE:
        //ret = __dtc_normal_conti();
        break;
    case DTC_DEDICATE:
        //ret = __dtc_dedicate();
        break;
    case DTC_DEDICATE_CONT_MODE:
        //ret = __dtc_dedicate_conti();
        break;
    case DTC_ENQ_AFT_DONE:
        //ret = __dtc_enq_aftdone();
        break;
    case DTC_MANY_ENQ:
        //ret = __dtc_many_enq();
```

```
            break;
        case DTC_CMD_STOP:
            //ret = __dtc_stop();
            break;
        case DTC_M2M_TWO_THREAD:
            //ret = __dtc_two_thread();
            break;
        default:
            ret = __LINE__;
            break;
        }

        if(0 == ret)
            printk("%s: test success!\n", __func__);
        else
            printk("%s: test failed!\n", __func__);
        return ret;
}

ssize_t test_store(struct class *class, struct class_attribute *attr,
                const char *buf, size_t size)
{
        int id = 0;

        /* get test id */
        if(strict_strtoul(buf, 10, (long unsigned int *)&id)) {
            pr_err("%s: invalid string %s\n", __func__, buf);
            return -EINVAL;
        }
        pr_info("%s: string %s, test case %s\n", __func__, buf, case_name[id]);

        if(0 != dma_test_main(id))
            pr_err("%s: dma_test_main failed! id %d\n", __func__, id);
        else
            pr_info("%s: dma_test_main success! id %d\n", __func__, id);
        return size;
}

ssize_t help_show(struct class *class, struct class_attribute *attr, char *buf)
{
        ssize_t cnt = 0;
```

```
    cnt += sprintf(buf + cnt, "usage: echo id > test\n");
    cnt += sprintf(buf + cnt, "        id for case DTC_NORMAL                  is
%d\n", (int)DTC_NORMAL);
    cnt += sprintf(buf + cnt, "        id for case DTC_NORMAL_CONT_MODE
is %d\n", (int)DTC_NORMAL_CONT_MODE);
    cnt += sprintf(buf + cnt, "        id for case DTC_DEDICATE                is
%d\n", (int)DTC_DEDICATE);
    cnt += sprintf(buf + cnt, "        id for case DTC_DEDICATE_CONT_MODE
is %d\n", (int)DTC_DEDICATE_CONT_MODE);
    cnt += sprintf(buf + cnt, "          id for case DTC_ENQ_AFT_DONE
is %d\n", (int)DTC_ENQ_AFT_DONE);
    cnt += sprintf(buf + cnt, "        id for case DTC_MANY_ENQ                is
%d\n", (int)DTC_MANY_ENQ);
    cnt += sprintf(buf + cnt, "        id for case DTC_CMD_STOP                is
%d\n", (int)DTC_CMD_STOP);
    cnt += sprintf(buf + cnt, "          id for case DTC_M2M_TWO_THREAD
is %d\n", (int)DTC_M2M_TWO_THREAD);
    cnt += sprintf(buf + cnt, "case description:\n");
    cnt += sprintf(buf + cnt, "         DTC_NORMAL:                    case for
normal channel\n");
    cnt += sprintf(buf + cnt, "        DTC_NORMAL_CONT_MODE:    case for
normal channel continue mode\n");
    cnt += sprintf(buf + cnt, "         DTC_DEDICATE:                  case for
dedicate channel\n");
    cnt += sprintf(buf + cnt, "         DTC_DEDICATE_CONT_MODE: case for
dedicate channel continue mode\n");
    cnt += sprintf(buf + cnt, "    below is for dedicate:\n");
    cnt += sprintf(buf + cnt, "        DTC_ENQ_AFT_DONE:          enqueued
buffer after dma last done, to see if can cotinue auto start\n");
    cnt += sprintf(buf + cnt, "        DTC_MANY_ENQ:                many buffer
enqueued, function test\n");
    cnt += sprintf(buf + cnt, "        DTC_CMD_STOP:                stop when
dma running\n");
    cnt += sprintf(buf + cnt, "                DTC_M2M_TWO_THREAD:
two-thread run simutalously, pressure test and memory leak test\n");
    return cnt;
}

static struct class_attribute dma_test_class_attrs[] = {
    __ATTR(test, 0220, NULL, test_store), /* not 222, for CTS, other group
cannot have write permission, 2013-1-11 */
    __ATTR(help, 0444, help_show, NULL),
```

```
    __ATTR_NULL,
};

static struct class dma_test_class = {
    .name        = "sunxi_dma_test",
    .owner       = THIS_MODULE,
    .class_attrs = dma_test_class_attrs,
};

static int __init sw_dma_test_init(void)
{
    int  status;

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

    /* init dma wait queue */
    __dma_test_init_waitqueue();

    /* register sys class */
    status = class_register(&dma_test_class);
    if(status < 0)
        pr_info("%s err, status %d\n", __func__, status);
    else
        pr_info("%s success\n", __func__);
    return 0;
}

/**
 * sw_dma_test_exit - exit the dma test module
 */
static void __exit sw_dma_test_exit(void)
{
    pr_info("sw_dma_test_exit: enter\n");
}

#ifdef MODULE
module_init(sw_dma_test_init);
module_exit(sw_dma_test_exit);
MODULE_LICENSE ("GPL");
MODULE_AUTHOR ("liugang");
MODULE_DESCRIPTION ("sun7i Dma Test driver code");
#else
```

```
__initcall(sw_dma_test_init);
#endif /* MODULE */
```

# 7. Android 系统支持

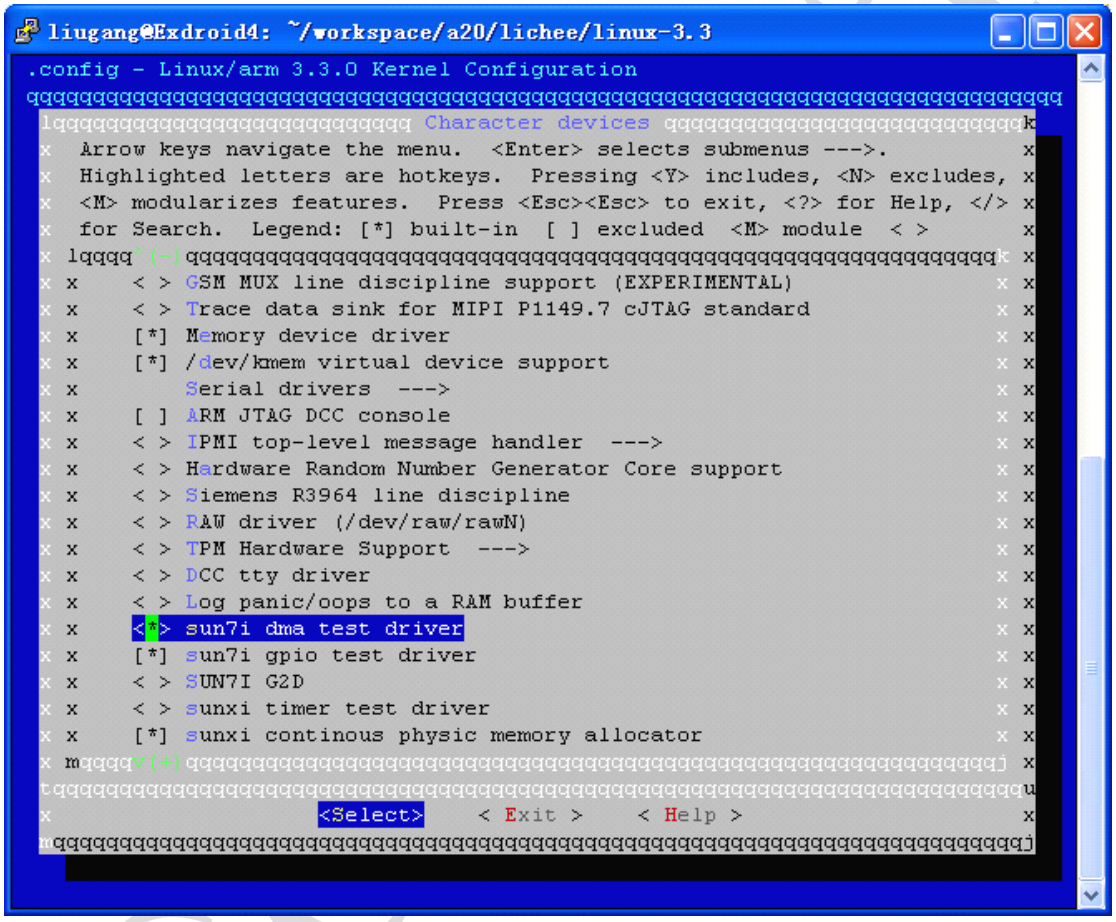
dma 属 linux 内核模块， 和 android 无直接关系.

# 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.

## 8.1. menuconfig 的配置

device drivers -> character devices -> sun7i dma test driver:




## 8.2. 测试用例选择

```
enum dma_test_case_e {
    DTC_NORMAL = 0,          /* case for normal channel */
    DTC_NORMAL_CONT_MODE,    /* case for normal channel continue
mode */
    DTC_DEDICATE,        /* case for dedicate channel */
    DTC_DEDICATE_CONT_MODE,  /* case for dedicate channel continue
```

```
mode */
    /*
     * for dedicate below
     */
    DTC_ENQ_AFT_DONE,    /* enqueued buffer after dma last done, to see
if can cotinue auto start */
    DTC_MANY_ENQ,        /* many buffer enqueued, function test */
    DTC_CMD_STOP,        /* stop when dma running */
    DTC_M2M_TWO_THREAD,  /* two-thread run simutalously, pressure
test and memory leak test */
    DTC_MAX
};
```

DTC_NORMAL: normal 通道测试;
DTC_NORMAL_CONT_MODE: normal 通道 continue 模式测试;
DTC_DEDICATE: dedicate 通道测试;
DTC_DEDICATE_CONT_MODE: dedicate 通道 continue 模式测试;
DTC_ENQ_AFT_DONE: 测试所有 buff 传完后，增加新的 buff(此时正常应该自动启动 dma)情形;
DTC_MANY_ENQ: 测试大量 enqueue buffer 情形;
DTC_CMD_STOP: 测试传输中途手动 stop 情形，查看剩余 buff 是否正常释放;
DTC_M2M_TWO_THREAD: 两个线程同时传输情形;

## 8.3. 测试操作步骤

(1) 启动样机，接串口;
(2) 进入 sysfs 节点: cd /sys/class/sunxi_dma_test/;
(3) 选择测试用例，进行测试: (比如测试 normal 通道)echo 0 > test;

# 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.

DMA 测试驱动提供了 sysfs 节点的方式，供测试 dma 硬件.

# 10. Declaration

This(A20 DMA 开发说明) is the original work and copyrighted property of Allwinner Technology ("Allwinner"). Reproduction in whole or in part must obtain the written approval of Allwinner and give clear acknowledgement to the copyright owner.

The information furnished by Allwinner is believed to be accurate and reliable. Allwinner reserves the right to make changes in circuit design and/or specifications at any time without notice. Allwinner does not assume any responsibility and liability for its use. Nor for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of Allwinner. This datasheet neither states nor implies warranty of any kind, including fitness for any particular application.