H616-dma 接口使用说明书

1.0

2019.12.10

文档履历

版本号	日期	制/修订人	内容描述
1.0	2019.12.10	AWA0991	
		AWA1440	

目录

1. 概述

1.1 编写目的

介绍 DMA Engine 模块及其接口使用方法：

1. dma driver framework
2. API 接口介绍
3. 使用范例及注意事项

1.2 适用范围

本文档适用于 Linux-4.9 内核。

1.3 相关人员

- DMA 模块使用者
- 驱动模块负责人

2. DMA engine 框架

2.1 基本概述

DMA engine 是 linux 内核 dma 驱动框架，针对 DMA 驱动的混乱局面内核社区提出了一个全新的框架驱动，目标在统一 dma API 让各个模块使用 DMA 时不用关心硬件细节，同时代码复用提高。并且实现异步的数据传输，降低机器负载。

2.1.1 术语约定

- DMA: Direct Memory Access(直接内存存取)
- Channel: DMA 通道
- Slave: 从通道，一般指设备通道
- Master: 主通道，一般指内存

2.1.2 功能简介

DMA engine 向使用者提供统一的接口，不同的模式下使用不同的 DMA 接口，省去使用过多的关注硬件接口。

2.2 基本结构

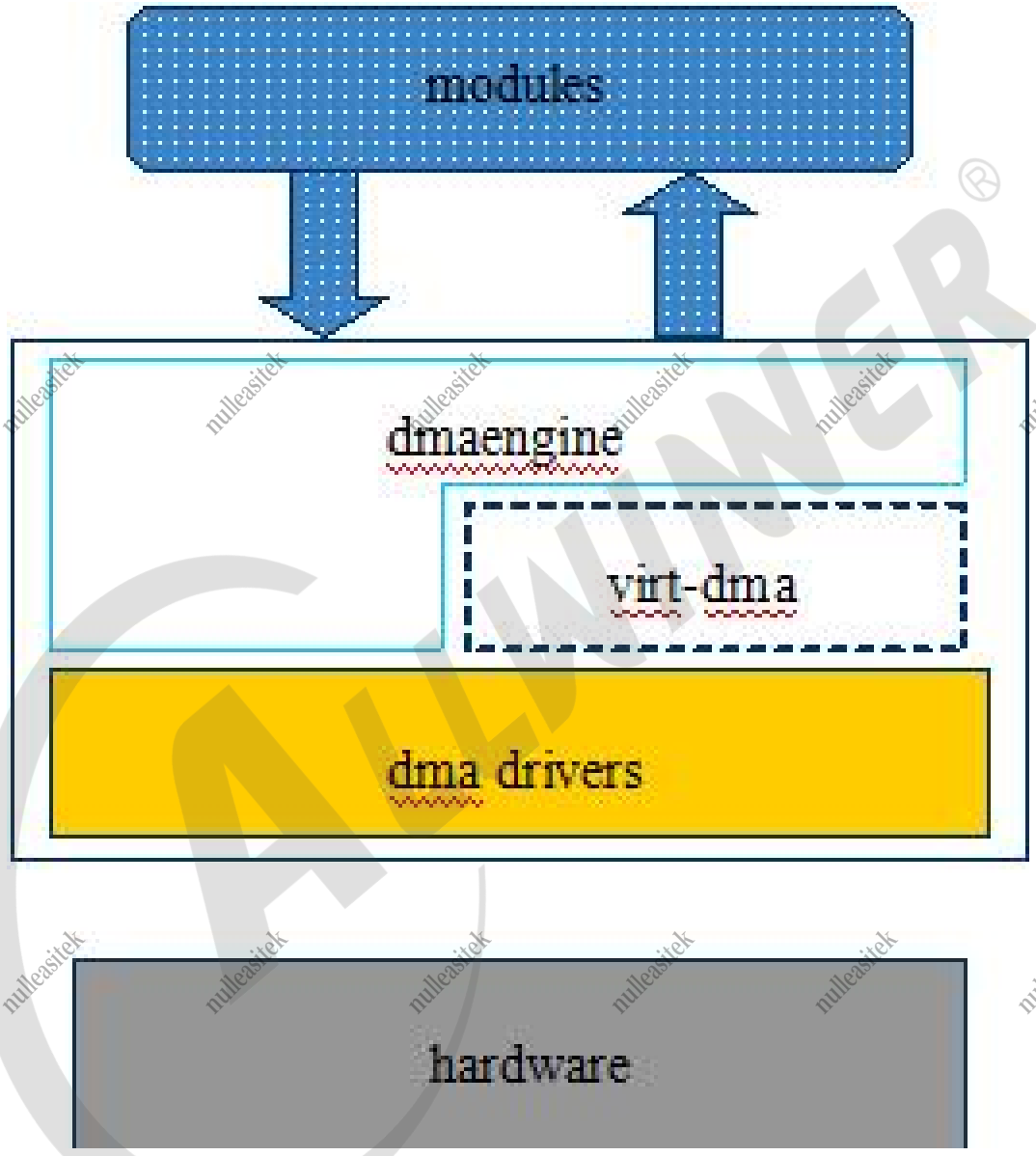

图 1: DMA engine 框架图

2.3 模式

2.3.1 内存拷贝

纯粹地内存拷贝，即从指定的源地址拷贝到指定的目的地址。传输完毕会发生一个中断，并调用回函数。

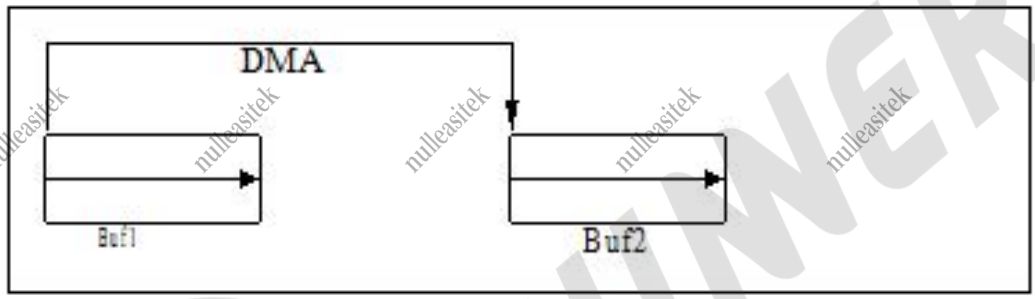

图 2: DMA engine 内存拷贝示意图

2.3.2 散列表

散列模式是把不连续的内存块直接传输到指定的目的地址。当传输完毕会发生一个中断，并调用回调函数。

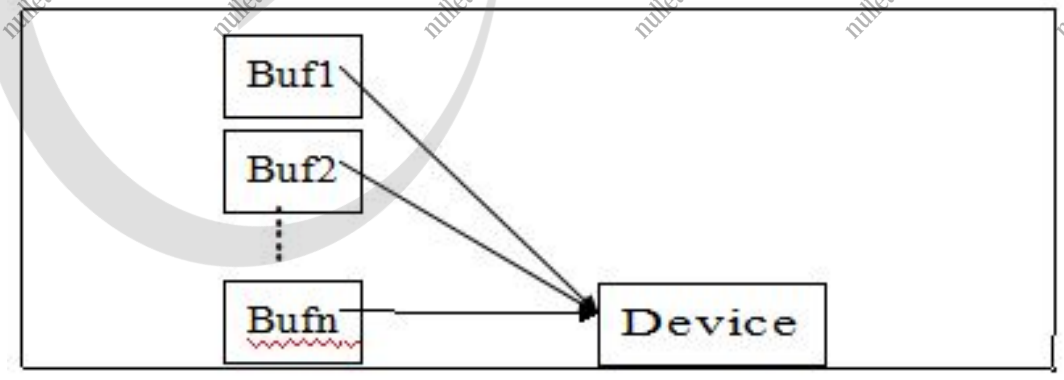

图 3: DMA engine 散列拷贝示意图 (slave 与 master)

上述的散列拷贝操作是针对于 Slave 设备而言的，它支持的是 Slave 与 Master 之间的拷贝，还有另一散列拷贝是专门对内存进行操作的，即 Master 与 Master 之间进行操作，具体形式图如下：

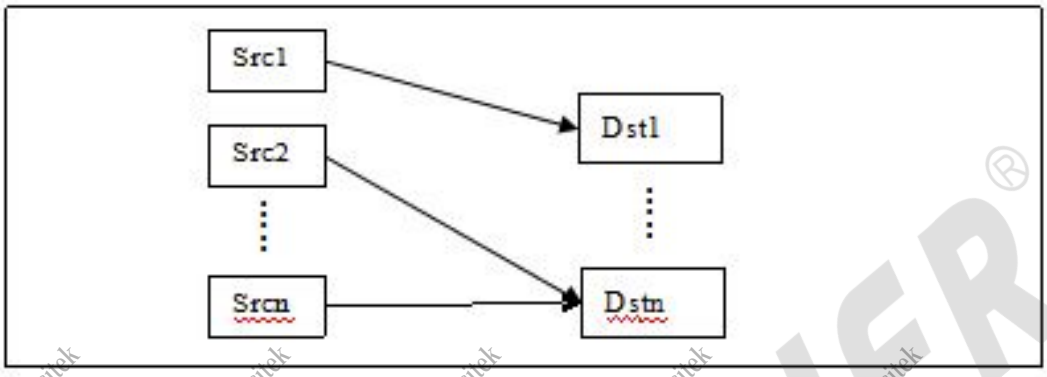

图 4: DMA engine 散列拷贝示意图 (master 与 master)

2.3.3 循环缓存

循环模式就是把一块 Ring buffer 切成若干片，周而复始的传输，每传完一个片会发生一个中断，同时调用回调函数。

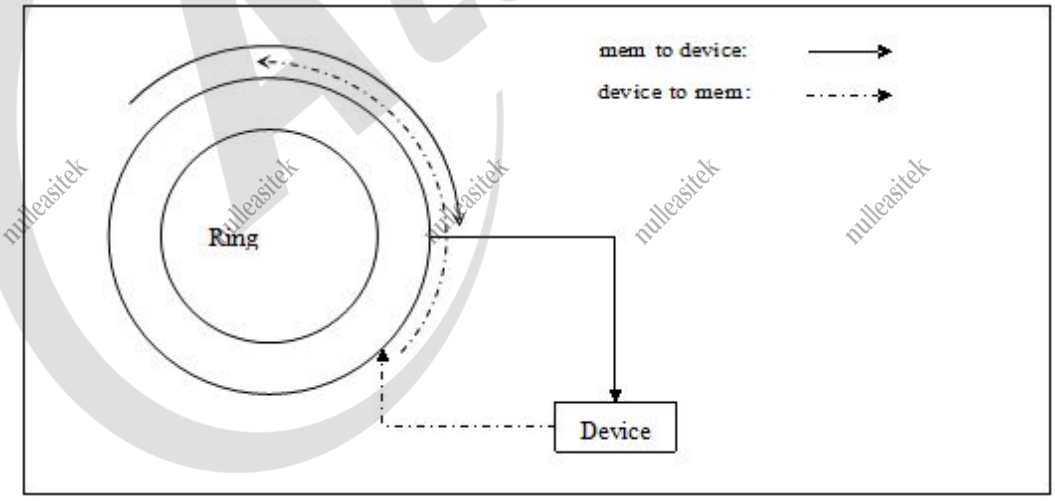

图 5: DMA engine 循环拷贝示意图

3. 接口介绍

3.1 通道相关

```
struct dma_chan *dma_request_channel(dma_cap_mask_t *mask, dma_filter_fn fn, void *fn_param);
```

功能：申请一个可用通道

参数：

mask: 所有申请的传输类型的掩码

fn: DMA 驱动私有的过滤函数

fn_param: 传入的私有参数

返回：返回一个通道数据指针

```
void dma_release_channel(struct dma_chan *chan)
```

功能：释放一个通道

参数：所需要释放的通道指针

3.2 配置相关

```
int dmaengine_slave_config(struct dma_chan *chan, struct dma_slave_config *config)
```

功能：配置一个从通道传输

参数：

chan: 通道结构指针

config: 配置数据指针

返回：非零表示失败，零表示成功

```
struct dma_slave_config {
    enum dma_transfer_direction direction;
    dma_addr_t src_addr;
    dma_addr_t dst_addr;
    enum dma_slave_buswidth src_addr_width;
    enum dma_slave_buswidth dst_addr_width;
    u32 src_maxburst;
    u32 dst_maxburst;
    bool device_fc;``
    unsigned int slave_id;
};
```

direction: 传输方向，取值 MEM_TO_DEV DEV_TO_MEM MEM_TO_MEM DEV_TO_DEV

src_addr: 源地址，必须是物理地址

dst_addr: 目的地址，必须是物理地址

src_addr_width: 源数据宽度，byte 整数倍，取值 1，2，4，8

dst_addr_width: 目的数据宽度，取值同上

src_max_burst: 源突发长度，取值 1，4，8

dst_max_burst: 目的突发长度，取值同上

slave_id: 从通道 id 号，此处用作 DRQ 的设置，使用 sunxi_slave_id(d, s) 宏设置，具体取值参照 include/linux/sunxi-dma.h 和 include/linux/dma/sunxi/dma-sun8iw18.h 里使用

3.3 传输相关

```
struct dma_async_tx_descriptor *dmaengine_prep_slave_single(struct dma_chan *chan,
 dma_addr_t buf, size_t len, enum dma_transfer_direction dir,
 unsigned long flags)
```

功能：准备一次单包传输

参数：

chan: 通道指针

buf: 需要传输地址

dir: 传输方向，此处为 DMA_MEM_TO_DEV，DMA_DEV_TO_MEM

flags: 传输标志

返回：返回一个传输描述符指针

```
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)
```

功能：准备一次多包传输，散列形式 (slave 模式）

参数：

chan: 通道指针

sgl: 散列表地址，此散列表传输之前需要建立

sg_len: 散列表内 buffer 的个数

dir: 传输方向，此处为 DMA_MEM_TO_DEV，DMA_DEV_TO_MEM

flags: 传输标志

返回：返回一个传输描述符指针

```
struct dma_async_tx_descriptor *device_prep_dma_sg(
struct dma_chan *chan,
struct scatterlist *dst_sg, unsigned int dst_nents,
struct scatterlist *src_sg, unsigned int src_nents,
unsigned long flags);
```

功能：准备一次多包传输，散列形式 (Master 模式)

参数：

chan: 通道指针

dst_sg: 目的散列表地址，此散列表传输之前需要建立

dst_nents: 散列表内 buffer 的个数

src_sg: 源散列表地址，此散列表传输之前需要建立

src_nents: 散列表内 buffer 的个数

flags；传输标志

返回：返回一个传输描述符指针

```
struct dma_async_tx_descriptor *dmaengine_prep_dma_cyclic(struct dma_chan *chan,
 dma_addr_t buf_addr, size_t buf_len, size_t period_len,
 enum dma_transfer_direction dir, unsigned long flags)
```

功能：准备一次环形 buffer 传输

参数：

chan：通道指针

buf_addr：环形 buffer 起始地址，必须为物理地址

buf_len：环形 buffer 的长度

period_len：每一小片 buffer 的长度

dir：传输方向，此处为 DMA_MEM_TO_DEV，DMA_DEV_TO_MEM

flags：传输标志

返回：返回一个传输描述符指针

```
struct dma_async_tx_descriptor {
 dma_cookie_t cookie;
 enum dma_ctrl_flags flags; /* not a 'long' to pack with cookie */
 dma_addr_t phys;
 struct dma_chan *chan;
 dma_cookie_t (*tx_submit)(struct dma_async_tx_descriptor *tx);
 dma_async_tx_callback callback;
 void *callback_param;
};
```

cookie：本次传输的 cookie，在此通道上唯一

tx_submit：本次传输的提交执行函数

callback：传输完成后的回调函数

callback_param：回调函数的参数

```
dma_cookie_t dmaengine_submit(struct dma_async_tx_descriptor *desc)
```

功能：提交已经做好准备的传输

参数：

desc：传输描述符，由前面准备函数获得

返回：返回一个 cookie，小于零为失败

```
void dma_async_issue_pending(struct dma_chan *chan)
```

功能：启动通道传输

参数：

chan：通道指针

3.4 其他

```
int dmaengine_terminate_all(struct dma_chan *chan)
```

功能：停止通道上的传输

参数：

chan：通道指针

返回：非零失败，零成功

```
int dmaengine_pause(struct dma_chan *chan)
```

功能：暂停某通道的传输

参数：

chan：通道指针

返回：非零失败，零成功

```
int dmaengine_resume(struct dma_chan *chan)
```

功能：恢复某通道的传输

参数：

chan：通道指针

返回：非零失败，零成功

```
enum dma_status dmaengine_tx_status(struct dma_chan *chan, dma_cookie_t cookie,
    struct dma_tx_state *state)
```

功能：查询某次提交的状态

参数：

chan：通道指针

cookie：提交返回的 id

state：用于获取状态的变量地址

返回：返回当前的状态，取值如下：

DMA_SUCCESS：传输成功完成

DMA_IN_PROGRESS：提交尚未处理或处理中

DMA_PAUSE：传输已经暂停

DMA_ERROR：此次传输失败

4. DMA engine 使用流程

本章节主要是讲解 DMA engine 的使用流程，以及注意事项

4.1 基本流程

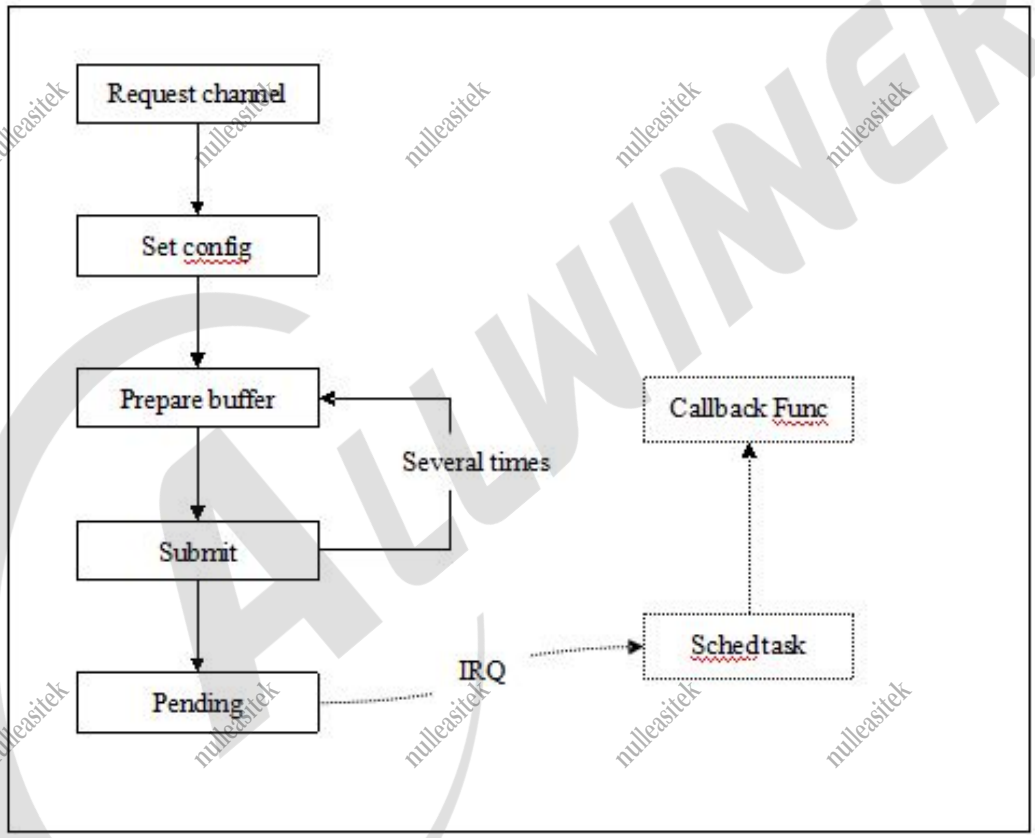

图 6: DMA engine 使用流程

4.2 注意事项

- 回调函数里不允许休眠，以及调度
- 回调函数时间不宜过长

- Pending 并不是立即传输而是等待软中断的到来，cyclic 模式除外
- 在 dma_slave_config 中的 slave_id 对于 deivces 必须要指定

5. 使用范例

5.1 范例

```
struct dma_chan *chan;
dma_cap_mask_t mask;
dma_cookie_t cookie;
struct dma_slave_config config;
struct dma_tx_state state;
struct dma_async_tx_descriptor *tx = NULL;
void *src_buf;
dma_addr_t src_dma;

dma_cap_zero(mask);
dma_cap_set(DMA_SLAVE, mask);
dma_cap_set(DMA_CYCLIC, mask);

/* 申请一个可用通道 */
chan = dma_request_channel(dt->mask, NULL, NULL);
if (!chan){
    return -EINVAL;
}

src_buf = kmalloc(1024*4, GFP_KERNEL);
if (!src_buf) {
 dma_release_channel(chan);
 return -EINVAL;
}

/* 映射地址用DMA访问 */
src_dma = dma_map_single(NULL, src_buf, 1024*4, DMA_TO_DEVICE);

config.direction = DMA_MEM_TO_DEV;
config.src_addr = src_dma;
config.dst_addr = 0x01c;
config.src_addr_width = DMA_SLAVE_BUSWIDTH_2_BYTES;
config.dst_addr_width = DMA_SLAVE_BUSWIDTH_2_BYTES;
config.src_maxburst = 1;
config.dst_maxburst = 1;
config.slave_id = sunxi_slave_id(DRQDST_AUDIO_CODEC, DRQSRC_SDRAM);

dmaengine_slave_config(chan, &config);

tx = dmaengine_pre_dma_cyclic(chan, scr_dma, 1024*4, 1024, DMA_MEM_TO_DEV,
 DMA_PREP_INTERRUPT | DMA_CTRL_ACK);
```

```
/* 设置回调函数 */
tx->callback = dma_callback;
tx->callback = NULL;

/* 提交及启动传输 */
cookie = dmaengine_submit(tx);
dma_async_issue_pending(chan);
```

6. Declaration

This document 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 document neither states nor implies warranty of any kind, including fitness for any particular application.tates nor implies warranty of any kind, including fitness for any particular application.