



Hi3511/Hi3512 扩展接口驱动

API 参考

文档版本	02
发布日期	2008-09-05
BOM编码	N/A

深圳市海思半导体有限公司为客户提供全方位的技术支持，用户可与就近的海思办事处联系，也可直接与公司总部联系。

深圳市海思半导体有限公司

地址： 深圳市龙岗区坂田华为基地华为电气生产中心 邮编：518129

网址： <http://www.hisilicon.com>

客户服务电话： 0755-28788858

客户服务传真： 0755-28357515

客户服务邮箱： support@hisilicon.com.

版权所有 © 深圳市海思半导体有限公司 2008。 保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明



HISILICON、海思，均为深圳市海思半导体有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

秘密

版权所有 © 深圳市海思半导体有限公司



目 录

前 言.....	1
1 概述.....	1-1
1.1 Hi3511/Hi3512 扩展接口	1-2
1.2 函数描述方式	1-2
2 扩展接口驱动函数	2-1
2.1 I ² C接口驱动函数.....	2-2
2.2 SSP接口驱动函数	2-9
2.3 DMAC接口驱动函数	2-32
3 DMAC程序示例	3-1
3.1 链表方式的DMA传输.....	3-2
3.1.1 配置流程.....	3-2
3.1.2 实例代码.....	3-2
3.2 无链表方式的DMA传输.....	3-6
3.2.1 内存到内存	3-6
3.2.2 内存到外设.....	3-7
A 缩略语	A-1



前 言

概述

本文档主要介绍 Hi3511/Hi3512 扩展接口驱动函数。

产品版本

与本文档相对应的产品版本如下所示。

产品名称	产品版本
Hi3511 H.264 编解码处理器	V100
Hi3512 H.264 编解码处理器	V100

读者对象

本文档主要适用于以下工程师：

- 软件开发工程师
- 技术支持工程师

内容简介

本文档介绍 Hi3511/Hi3512 扩展接口驱动函数。本文档共分为 3 章和 1 个附录。




章节	内容
1 概述	介绍扩展接口及扩展接口驱动函数描述方式。
2 扩展接口驱动函数	按模块介绍扩展接口驱动函数。
3 DMAC 程序示例	举例说明 DMAC 接口使用。
A 缩略语	给出文中出现的缩略语及其全称。



约定

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明
 危险	以本标志开始的文本表示有高度潜在危险，如果不能避免，会导致人员死亡或严重伤害。
 警告	以本标志开始的文本表示有中度或低度潜在危险，如果不能避免，可能导致人员轻微或中等伤害。
 注意	以本标志开始的文本表示有潜在风险，如果忽视这些文本，可能导致设备或器件损坏、数据丢失、设备性能降低或不可预知的结果。
 窍门	以本标志开始的文本能帮助您解决某个问题或节省您的时间。
 说明	以本标志开始的文本是正文的附加信息，是对正文的强调和补充。

通用格式约定

格式	说明
宋体	正文采用宋体表示。
黑体	一级、二级、三级标题采用黑体。
楷体	警告、提示等内容一律用楷体，并且在内容前后增加线条与正文隔离。
“Terminal Display” 格式	“Terminal Display” 格式表示屏幕输出信息。此外，屏幕输出信息中夹杂的用户从终端输入的信息采用加粗字体表示。

命令行格式约定

格式	意义
粗体	命令行关键字（命令中保持不变、必须照输的部分）采用加粗字体表示。



格式	意义
<i>斜体</i>	命令行参数（命令中必须由实际值进行替代的部分）采用 <i>斜体</i> 表示。
[]	表示用“[]”括起来的部分在命令配置时是可选的。
{ x y ... }	表示从两个或多个选项选取一个。
[x y ...]	表示从两个或多个选项选取一个或者不选。
{ x y ... } *	表示从两个或多个选项选取多个，最少选取一个，最多选取所有选项。
[x y ...] *	表示从两个或多个选项选取多个或者不选。

图形界面元素引用约定

格式	意义
“ ”	带双引号“ ”的格式表示各类界面控件名称和数据表，如单击“确定”。
>	多级菜单用“>”隔开。如选择“文件 > 新建 > 文件夹”，表示选择“文件”菜单下的“新建”子菜单下的“文件夹”菜单项。

键盘操作约定

格式	意义
加“ ”的字符	表示键名。如“Enter”、“Tab”、“Backspace”、“a”等分别表示回车、制表、退格、小写字母a。
“键 1+键 2”	表示在键盘上同时按下几个键。如“Ctrl+Alt+A”表示同时按下“Ctrl”、“Alt”、“A”这三个键。
“键 1，键 2”	表示先按第一键，释放，再按第二键。如“Alt, F”表示先按“Alt”键，释放后再按“F”键。

鼠标操作约定

格式	意义
单击	快速按下并释放鼠标的的一个按钮。
双击	连续两次快速按下并释放鼠标的的一个按钮。



格式	意义
拖动	按住鼠标的按钮不放，移动鼠标。

修改记录

修改记录累积了每次文档更新的说明。最新版本的文档包含以前所有文档版本的更新内容。

修改日期	版本	修改说明
2008-08-25	02	增加 Hi3512 芯片信息。
2008-05-30	01	第 1 次版本发布。



1 概述

关于本章

本章描述内容如下表所示。

标题	内容
1.1 Hi3511/Hi3512 扩展接口	简单介绍 Hi3511/Hi3512 扩展接口。
1.2 函数描述方式	介绍扩展接口驱动函数的描述方式。



1.1 Hi3511/Hi3512 扩展接口

Hi3511/Hi3512 的扩展接口是芯片本身具有的接口，需要接外设才能使用。扩展接口主要有两方面作用：

- Hi3511/Hi3512 芯片通过扩展接口实现对外设的控制和数据传递。
- 用户通过扩展接口构建其他外围设备所需的驱动。

1.2 函数描述方式

Hi3511/Hi3512 扩展接口驱动函数使用以下域来描述。

参数域	作用
目的	简要描述扩展接口驱动函数的主要功能。
语法	给出扩展接口驱动函数的语法样式。
参数	列出扩展接口驱动函数的参数及参数说明。
返回值	列出扩展接口驱动函数的返回值及返回值说明。
需求	列出扩展接口驱动函数要包含的头文件和包含本函数的库文件。
注意	给出使用扩展接口驱动函数时应注意的事项。
举例	给出使用扩展接口驱动函数的实例。
相关主题	给出同本扩展接口驱动函数相关的其他信息。



2 扩展接口驱动函数

关于本章

本章描述内容如下表所示。

标题	内容
2.1 I²C 接口驱动函数	介绍 I ² C 接口驱动函数。
2.2 SSP 接口驱动函数	介绍 SSP 接口驱动函数。
2.3 DMAC 接口驱动函数	介绍 DMAC 接口驱动函数。



2.1 I²C 接口驱动函数

I²C 接口驱动模块主要功能是读写标准 I²C 设备。该功能模块主要提供以下接口函数：

- [hi_i2c_read](#): 标准 I²C 读取函数。
- [hi_i2c_write](#): 标准 I²C 写入函数。
- [hi_i2c_muti_read](#): 标准 I²C 连续读取多个数据函数。
- [hi_i2c_muti_write](#): 标准 I²C 连续写入多个数据函数。



注意

- 在调用以上接口函数之前，必须先调用 `DECLARE_KCOM_HI_I2C()`宏函数声明，然后在调用模块的初始化函数中调用 `KCOM_HI_I2C_INIT()`宏函数后，才能正常使用。
- 在卸载此调用模块前也要在此调用模块的退出函数中调用 `KCOM_HI_I2C_EXIT()`宏函数。
- 每个宏函数在每个模块中只能被调用一次，多次调用可能会出错。

宏函数的应用举例如下：

```
DECLARE_KCOM_HI_I2C();
static int __init adv7179_init(void)
{
    int ret = 0;

    ret = KCOM_HI_I2C_INIT();
    if(ret)
    {
        printk("I2C module is not load.\n");
        return -1;
    }
    ret = misc_register(&adv7179_dev);
    if(ret)
    {
        KCOM_HI_I2C_EXIT();
        printk("could not register adv7179 devices. \n");
        return -1;
    }

    if(adv7179_device_init()<0)
    {
        misc_deregister(&adv7179_dev);
```



```
        KCOM_HI_I2C_EXIT();
        printk("adv7179 driver init fail for device init error!\n");
        return -1;
    }

    printk("adv7179 driver init successful!\n");
    return ret;
}

static void __exit adv7179_exit(void)
{
    misc_deregister(&adv7179_dev);
    KCOM_HI_I2C_EXIT();
    printk("adv7179 exit OK.\n");
}
```

2.1.1.1 hi_i2c_read

【目的】

读取 I²C 设备的某个寄存器的值。

【语法】

```
unsigned char hi_i2c_read(unsigned char devaddress,
                          unsigned char regaddress);
```

【参数】

参数名称	描述	输入/输出
devaddress	I ² C 设备的地址。	输入
regaddress	I ² C 设备的寄存器地址。	输入

【返回值】

读取的 I²C 设备的某个寄存器的值。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>



- `#include <linux/poll.h>`
- `#include <asm/hardware.h>`
- `#include <asm/bitops.h>`
- `#include <asm/uaccess.h>`
- `#include <asm/irq.h>`
- `#include <asm/io.h>`
- `#include <linux/moduleparam.h>`
- `#include <linux/types.h>`
- `#include <linux/fs.h>`
- `#include <linux/ioport.h>`
- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hi_i2c.h>`

【注意】

无。

【举例】

无。

【相关主题】

[hi_i2c_write](#)

2.1.1.2 hi_i2c_write

【目的】

向 I²C 设备的某个寄存器写入值。

【语法】

```
int hi_i2c_write(unsigned char devaddress,  
                 unsigned char regaddress,  
                 unsigned char data);
```

【参数】

参数名称	描述	输入/输出
devaddress	I ² C 设备的地址。	输入
regaddress	I ² C 设备的寄存器地址。	输入
data	写入寄存器的值。	输入

【返回值】



返回值	描述
0	成功。
-1	失败。

【需求】

头文件：

- `#include <linux/config.h>`
- `#include <linux/module.h>`
- `#include <linux/kernel.h>`
- `#include <linux/init.h>`
- `#include <linux/miscdevice.h>`
- `#include <linux/proc_fs.h>`
- `#include <linux/poll.h>`
- `#include <asm/hardware.h>`
- `#include <asm/bitops.h>`
- `#include <asm/uaccess.h>`
- `#include <asm/irq.h>`
- `#include <asm/io.h>`
- `#include <linux/moduleparam.h>`
- `#include <linux/types.h>`
- `#include <linux/fs.h>`
- `#include <linux/ioport.h>`
- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hi_i2c.h>`

【注意】

无。

【举例】

此例通过调用 I²C 的相关读写函数实现视频数模转换的初始化。

```
regvalue1 = hi_i2c_read(I2C_ADV7179, 0x07);
hi_i2c_write(I2C_ADV7179, 0x07, 0xa5);
regvalue2 = hi_i2c_read(I2C_ADV7179, 0x07);
if(regvalue2 != 0xa5)
{
    printk("read adv7179 register is %x\n", regvalue2);
    printk("check adv7179 error.\n");
    return -EFAULT;
}
```



```
hi_i2c_write(I2C_ADV7179, 0x07, regvalue1);  
close_vad();  
return 0;
```

【相关主题】[hi_i2c_read](#)

2.1.1.3 hi_i2c_muti_read

【目的】

从 I²C 设备某个子地址开始的位置连续读出多个数据。

【语法】

```
int hi_i2c_muti_read(unsigned char devaddress,  
                     unsigned int regaddress,  
                     int reg_addr_count,  
                     unsigned char *data,  
                     unsigned long count);
```

【参数】

参数名称	描述	输入/输出
Devaddress	I ² C 设备的地址。	输入
Regaddress	I ² C 设备的子地址。	输入
reg_addr_count	子地址长度（以字节为单位）。 1: 8 位子地址; 2: 16 位子地址; 3: 24 位子地址; 4: 32 位子地址。 其它值非法。	输入
Data	需要读取的数据将要被存放的首地址。	输入
Count	需要读取的数据个数。	输入

【返回值】

返回值	描述
0	成功。
-1	失败。



【需求】

头文件：

- `#include <linux/config.h>`
- `#include <linux/module.h>`
- `#include <linux/kernel.h>`
- `#include <linux/init.h>`
- `#include <linux/miscdevice.h>`
- `#include <linux/proc_fs.h>`
- `#include <linux/poll.h>`
- `#include <asm/hardware.h>`
- `#include <asm/bitops.h>`
- `#include <asm/uaccess.h>`
- `#include <asm/irq.h>`
- `#include <asm/io.h>`
- `#include <linux/moduleparam.h>`
- `#include <linux/types.h>`
- `#include <linux/fs.h>`
- `#include <linux/ioport.h>`
- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hi_i2c.h>`

【注意】

无。

【举例】

无。

【相关主题】

[hi_i2c_muti_write](#)

2.1.1.4 hi_i2c_muti_write

【目的】

向 I²C 设备某个子地址开始的位置连续写入多个数据。

【语法】

```
int hi_i2c_muti_write(unsigned char devaddress,  
                      unsigned int regaddress,  
                      int reg_addr_count,  
                      unsigned char *data,  
                      unsigned long count);
```




【参数】

参数名称	描述	输入/输出
Devaddress	I ² C 设备的地址。	输入
Regaddress	I ² C 设备的子地址。	输入
reg_addr_count	子地址长度（以字节为单位）。 1: 8 位子地址; 2: 16 位子地址; 3: 24 位子地址; 4: 32 位子地址。	输入
Data	将要写入的数据的首地址。	输入
Count	将要写入的数据的个数。	输入

【返回值】

返回值	描述
0	成功。
-1	失败。

【需求】

头文件:

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>



- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hi_i2c.h>`

【注意】

无。

【举例】

此例通过调用 I²C 的连续读写函数实现 e2prom 的读写访问。

```
int e2prom_read(unsigned int sub_addr,
                int sub_addr_count,
                unsigned char *pdata_out,
                int count)
{
    return hi_i2c_muti_read(E2PROM_DEV_ADDRESS,
                           sub_addr,
                           sub_addr_count,
                           pdata_out,
                           count);
}

int e2prom_write (unsigned int sub_addr,
                  int sub_addr_count,
                  unsigned char *data,
                  int count)
{
    return hi_i2c_muti_write(E2PROM_DEV_ADDRESS,
                             sub_addr,
                             sub_addr_count,
                             data,
                             count);
}
```

【相关主题】

[hi_i2c_muti_read](#)

2.2 SSP 接口驱动函数

SSP 接口驱动模块提供如下功能：

- 对 SSP 进行初始化配置。
- 通过 CPU 进行数据读写。
- 结合 DMAC 实现 DMA 数据传输。

该功能模块提供以下接口函数：



- `hi_ssp_enable`: 使能 SSP。
- `hi_ssp_disable`: 禁止 SSP。
- `hi_ssp_set_frameform`: 设置 SSP 的帧格式。
- `hi_ssp_set_serialclock`: 设置 SSP 的串行时钟。
- `hi_ssp_set_inturrupt`: 设置 SSP 的中断。
- `hi_ssp_interrupt_clear`: 清除 SSP 的中断。
- `hi_ssp_dmac_enable`: 使能 SSP 的 DMA 传输模式。
- `hi_ssp_dmac_disable`: 禁止 SSP 的 DMA 传输模式。
- `hi_ssp_busystate_check`: 检测 SSP 是否处于忙状态。
- `hi_ssp_readdata`: 读取 SSP 的 FIFO 中的数据。
- `hi_ssp_writedata`: 向 SSP 的 FIFO 中写入数据。
- `hi_ssp_dmac_init`: 初始化 SSP 的 DMA 传输模式。
- `hi_ssp_dmac_transfer`: 配置 SSP 的 DMA 传输模式所需传输的数据。
- `hi_ssp_dmac_exit`: 退出 SSP 的 DMA 传输模式。



注意

- 在调用以上接口函数之前，必须先调用 `DECLARE_KCOM_HI_SSP()`宏函数声明，然后在调用模块的初始化函数中调用 `KCOM_HI_SSP_INIT()`宏函数后，才能正常使用。
- 在卸载此调用模块前也要在此调用模块的退出函数中调用 `KCOM_HI_SSP_EXIT()`宏函数。
- 每个宏函数在每个模块中只能被调用一次，多次调用可能会出错。调用方式和 I²C 模块相同。

2.2.1.1 hi_ssp_enable

【目的】

使能 SSP。

【语法】

```
void hi_ssp_enable(void);
```

【参数】

无

【返回值】

无

【需求】

头文件：



- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hi_ssp.h>

【注意】

无。

【举例】

无。

【相关主题】

无。

2.2.1.2 hi_ssp_disable

【目的】

禁止 SSP。

【语法】

```
void hi_ssp_disable(void);
```

【参数】

无。

【返回值】

无。

【需求】



头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hi_ssp.h>

【注意】

当初始化 SSP 时，在配置帧格式、位宽、时钟等之前，需要先调用此函数保证 SSP 处于禁止状态。

【举例】

无。

【相关主题】

无。

2.2.1.3 hi_ssp_set_frameform

【目的】

设置 SSP 的帧格式。

【语法】

```
int hi_ssp_set_frameform(unsigned char framemode,  
                          unsigned char spo,  
                          unsigned char sph,  
                          unsigned char datawidth);
```

【参数】



参数名称	描述	输入/输出
framemode	SSP 帧格式。 0: MOTOROLA SPI 模式。 1: TI 同步串行模式。 2: National microware 模式。 3~255: 保留。	输入
spo	SSP 处于稳定状态时的时钟极性。 0: 低。 1: 高。 注意: 此参数只在 MOTOROLA SPI 模式下有效。	输入
sph	SSP 采集数据的时候是使用时钟的上升沿还是下降沿。 0: 上升沿。 1: 下降沿。 注意: 此参数只在 MOTOROLA SPI 模式下有效。	输入
datawidth	数据位宽, 单位为 bit, 取值范围为 4~16。	输入

【返回值】

返回值	描述
0	成功。
-1	失败。

【需求】

头文件:

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>



- `#include <linux/moduleparam.h>`
- `#include <linux/types.h>`
- `#include <linux/fs.h>`
- `#include <linux/ioport.h>`
- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hi_ssp.h>`

【注意】

无。

【举例】

此例通过调用 SSP 的使能、禁止函数实现对 SD 卡的初始化。

```
void ssp_init_for_sdcard(void)
{
    hi_ssp_disable();
    if(hi_ssp_set_frameform(SSP_CR0_FRF_MOT,
                           SSP_CR0_SPO,
                           SSP_CR0_SPH,
                           SSP_CR0_DSS_8))
    {
        printf("hi_ssp_set_frameform error.\n");
    }
    if(hi_ssp_set_serialclock(SSP_CR0_SCR_DFLT, SSP_CPSR_DFLT))
    {
        printf("hi_ssp_set_serialclock error.\n");
    }
    hi_ssp_set_inturrupt(0);
#ifdef CONFIG_SSP_DMA
    hi_ssp_dmac_enable();
#else
    hi_ssp_dmac_disable();
#endif
    hi_ssp_enable();
    hi_ssp_interrupt_clear();
}
```

【相关主题】

无。

2.2.1.4 hi_ssp_set_serialclock

【目的】



设置 SSP 的串行时钟。

【语法】

```
int hi_ssp_set_serialclock(unsigned char scr, unsigned char cpsdvsr);
```

【参数】

参数名称	描述	输入/输出
scr	SSP 串行时钟率，取值范围为 0~255。 SSP 接口工作时钟计算公式如下： $F_{SSPCLK} / (CPSDVSR \times (1 + SCR))$ 其中 FSSPCLK 是系统时钟，CPSDVSR 和 SCR 分别由参数 cpsdvsr 和 scr 指定。	输入
cpsdvsr	SSP 预分频参数，必须设为 2~254 之间的偶数。	输入

【返回值】

返回值	描述
0	成功。
-1	失败。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>



- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hi_ssp.h>`

【注意】

无。

【举例】

无。

【相关主题】

无。

2.2.1.5 hi_ssp_set_inturrupt

【目的】

设置 SSP 的中断。

【语法】

```
void hi_ssp_set_inturrupt(unsigned char regvalue);
```

【参数】

参数名称	描述	输入/输出
regvalue	SSP 的各中断使能禁止标志。 仅低 4 位有效： <ul style="list-style-type: none">• bit[0]: RORIM（接收溢出中断）。• bit[1]: RTIM（接收超时中断）。• bit[2]: RXIM（接收 FIFO 中断）。• bit[3]: TXIM（发送 FIFO 中断）。 各 bit 含义如下： 0: 禁止中断； 1: 使能中断。	输入

【返回值】

无。

【需求】

头文件：

- `#include <linux/config.h>`
- `#include <linux/module.h>`
- `#include <linux/kernel.h>`



- `#include <linux/init.h>`
- `#include <linux/miscdevice.h>`
- `#include <linux/proc_fs.h>`
- `#include <linux/poll.h>`
- `#include <asm/hardware.h>`
- `#include <asm/bitops.h>`
- `#include <asm/uaccess.h>`
- `#include <asm/irq.h>`
- `#include <asm/io.h>`
- `#include <linux/moduleparam.h>`
- `#include <linux/types.h>`
- `#include <linux/fs.h>`
- `#include <linux/ioport.h>`
- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hi_ssp.h>`

【注意】

通过本函数可以使能或者禁止各种组合的中断：

- `hi_ssp_set_inturrupt(0x00)`
禁止所有的中断。
- `hi_ssp_set_inturrupt(0x0F)`
使能所有中断。
- `hi_ssp_set_inturrupt(0x03)`
使能 RORIM 和 RTIM 中断。

【举例】

无。

【相关主题】

无。

2.2.1.6 hi_ssp_interrupt_clear

【目的】

清除 SSP 的中断。

【语法】

```
void hi_ssp_interrupt_clear(void);
```

【参数】

无。

**【返回值】**

无。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hi_ssp.h>

【注意】

调用本函数清除所有的中断状态位，并非中止中断。

【举例】

无。

【相关主题】

无。

2.2.1.7 hi_ssp_dmac_enable

【目的】

使能 SSP 的 DMA 传输模式。

【语法】

```
void hi_ssp_dmac_enable(void);
```

【参数】



无。

【返回值】

无。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hi_ssp.h>

【注意】

调用该函数前，要先调用 [hi_ssp_dmac_init](#)。

【举例】

无。

【相关主题】

无。

2.2.1.8 hi_ssp_dmac_disable

【目的】

禁止 SSP 的 DMA 传输模式。

【语法】

```
void hi_ssp_dmac_disable(void);
```

**【参数】**

无。

【返回值】

无。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hi_ssp.h>

【注意】

调用该函数前，要先调用 [hi_ssp_dmac_init](#)。

【举例】

无。

【相关主题】

无。

2.2.1.9 hi_ssp_busystate_check

【目的】

检测 SSP 是否处于忙状态。

【语法】



```
unsigned int hi_ssp_bustystate_check(void);
```

【参数】

无。

【返回值】

返回值	描述
0	SSP 闲。
1	SSP 忙。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hi_ssp.h>

【注意】

无。

【举例】

此例通过调用 SSP 的忙状态检测函数和 FIFO 的读写函数实现对 SD 卡的读操作。

```
unsigned char sdcard_read_byte(void)
{
    unsigned int loopi = 0;
```



```
while(loopi<1000)
{
    if(hi_ssp_bustystate_check())
        loopi++;
    else
        break;
}

if(loopi == 1000) /*timeout*/
{
    return -1;
}
hi_ssp_writedata(0xff);

loopi = 0;
while(loopi<1000)
{
    if(hi_ssp_bustystate_check())
        loopi++;
    else
        break;
}
if(loopi == 1000) /*timeout*/
{
    return -1;
}
return(hi_ssp_readdata());
}
```

【相关主题】

无。

2.2.1.10 hi_ssp_readdata

【目的】

读取 SSP 的 FIFO 中的数据。

【语法】

```
int hi_ssp_readdata(void);
```

【参数】

无。



【返回值】

从 FIFO 中读到的数据。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hi_ssp.h>

【注意】

无。

【举例】

无。

【相关主题】

无。

2.2.1.11 hi_ssp_writedata

【目的】

向 SSP 的 FIFO 中写入数据。

【语法】

```
void hi_ssp_writedata(unsigned short sdata);
```

【参数】



参数名称	描述	输入/输出
sdata	向 FIFO 中写入的数据。	输入

【返回值】

无。

【需求】

头文件：

- `#include <linux/config.h>`
- `#include <linux/module.h>`
- `#include <linux/kernel.h>`
- `#include <linux/init.h>`
- `#include <linux/miscdevice.h>`
- `#include <linux/proc_fs.h>`
- `#include <linux/poll.h>`
- `#include <asm/hardware.h>`
- `#include <asm/bitops.h>`
- `#include <asm/uaccess.h>`
- `#include <asm/irq.h>`
- `#include <asm/io.h>`
- `#include <linux/moduleparam.h>`
- `#include <linux/types.h>`
- `#include <linux/fs.h>`
- `#include <linux/ioport.h>`
- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hi_ssp.h>`

【注意】

无。

【举例】

无。

【相关主题】

无。

2.2.1.12 hi_ssp_dmac_init

【目的】

初始化 SSP 的 DMA 传输模式。



【语法】

```
int hi_ssp_dmac_init(void *prx_dmac_hook, void *ptx_dmac_hook);
```

【参数】

参数名称	描述	输入/输出
prx_dmac_hook	DMA 传输模式接收中断回调函数指针。	输入
ptx_dmac_hook	DMA 传输模式发送中断回调函数指针。	输入

【返回值】

返回值	描述
0	成功。
-1	失败。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hi_ssp.h>

【注意】

无。

**【举例】**

此例通过调用 SSP 的 DMA 传输模式初始化函数来实现 SD 卡的 DMA 传输模式的初始化。例子中以 hil_mmb 开头的一些函数是海思公司为内存管理增加的特有函数。

```
int dmactssp_init(unsigned int *readbuf, unsigned int *writebuf)
{
    hil_mmb_t *pBufAddr;

    memset(&dmactssp_s.ssp_txchbufaddress, 0, sizeof(dmactssp_s));
    dmactssp_temp_s.txbufaddress=0;
    dmactssp_temp_s.rxbufaddress=0;

    hi_ssp_dmac_init(rx_dmach_hook, tx_dmach_hook);
    pBufAddr =hil_mmb_alloc("SSP_TEMP_TXBUF",
                            PAGE_SIZE,
                            0,
                            HIL_MMZ_GFP_DDR,
                            NULL);
    if(NULL == pBufAddr)
    {
        printk("SSP_TEMP_TXBUF: cannot get mem\n");
        goto ssp_mmap_out;
    }
    dmactssp_temp_s.txbufaddress= hil_mmb_phys(pBufAddr);
    dmactssp_temp_s.txbufmapaddress =
        (unsigned int)ioremap_nocache(dmactssp_temp_s.txbufaddress,
                                      PAGE_SIZE);
    memset(dmactssp_temp_s.txbufmapaddress, 0xff, PAGE_SIZE);

    pBufAddr =hil_mmb_alloc("SSP_TEMP_RXBUF",
                            PAGE_SIZE,
                            0,
                            HIL_MMZ_GFP_DDR,
                            NULL);
    if(NULL == pBufAddr)
    {
        printk("SSP_TEMP_RXBUF: cannot get mem\n");
        goto ssp_mmap_out;
    }
    dmactssp_temp_s.rxbufaddress= hil_mmb_phys(pBufAddr);
    dmactssp_temp_s.rxbufmapaddress =
        (unsigned int)ioremap_nocache(dmactssp_temp_s.rxbufaddress,
                                      PAGE_SIZE);
```



```
pBufAddr =hil_mmb_alloc("SSP_TXBUF1",
                        DMAC_SSP_SIZE,
                        0,
                        HIL_MMZ_GFP_DDR,
                        NULL);

if(NULL == pBufAddr)
{
    printk("SSP_TXBUF1: cannot get mem\n");
    goto ssp_mmap_out;
}

dmac_ssp_s.ssptxchbufaddress= hil_mmb_phys(pBufAddr);
dmac_ssp_s.ssptxchbufmapaddress =
(unsigned int)ioremap_nocache(dmac_ssp_s.ssptxchbufaddress,
                              DMAC_SSP_SIZE);

pBufAddr= hil_mmb_alloc("SSP_RXBUF2",
                        DMAC_SSP_SIZE,
                        0,
                        HIL_MMZ_GFP_DDR,
                        NULL);

if (NULL == pBufAddr)
{
    printk("SSP_RXBUF2: cannot get mem\n");
    goto ssp_mmap_out;
}

dmac_ssp_s.ssprxchbufaddress =hil_mmb_phys(pBufAddr);
dmac_ssp_s.ssprxchbufmapaddress =
(unsigned int)ioremap_nocache(dmac_ssp_s.ssprxchbufaddress,
                              DMAC_SSP_SIZE);

*readbuf =dmac_ssp_s.ssprxchbufmapaddress;
*writebuf=dmac_ssp_s.ssptxchbufmapaddress;
memset((char *)dmac_ssp_s.ssprxchbufmapaddress,0,DMAC_SSP_SIZE);
memset((char *)dmac_ssp_s.ssptxchbufmapaddress,0,DMAC_SSP_SIZE);

return 0;

ssp_mmap_out:
if(dmac_ssp_temp_s.txbufaddress)
    hil_mmb_freeby_phys(dmac_ssp_temp_s.txbufaddress);
if(dmac_ssp_temp_s.rxbufaddress)
    hil_mmb_freeby_phys(dmac_ssp_temp_s.rxbufaddress);
if(dmac_ssp_s.ssptxchbufaddress)
    hil_mmb_freeby_phys(dmac_ssp_s.ssptxchbufaddress);
```



```
if(dmac_ssp_s.ssprxchbufaddress)
    hil_mmb_freeby_phys(dmac_ssp_s.ssprxchbufaddress);
return 1;
}
```

【相关主题】

无。

2.2.1.13 hi_ssp_dmac_transfer

【目的】

配置 SSP 的 DMA 传输模式所需传输的数据。

【语法】

```
int hi_ssp_dmac_transfer(unsigned int phy_rxbufaddr,
                        unsigned int phy_txbufaddr,
                        unsigned int transfersize);
```

【参数】

参数名称	描述	输入/输出
phy_rxbufaddr	DMA 传输模式接收缓冲物理地址。	输入
phy_txbufaddr	DMA 传输模式发送缓冲物理地址。	输入
transfersize	DMA 传输模式传输数据长度。	输入

【返回值】

返回值	描述
0	成功。
-EINVAL	失败。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>



- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hi_ssp.h>

【注意】

无。

【举例】

此例通过调用 SSP 的 DMA 传输模式数据传输配置函数实现对 SD 卡的 DMA 传输模式数据传输配置。

```
#ifdef CONFIG_SSP_DMA
    unsigned int sdcard_read(unsigned int addr,
                              unsigned int off,
                              unsigned int block_num)
#else
    unsigned int sdcard_read(unsigned int addr,unsigned char *buf,
                              unsigned int block_num)
#endif
{
    unsigned char CMD18[] = {0x52,0x00,0x00,0x00,0x00,0xFF};
    unsigned char CMD12[] = {0x4c,0x00,0x00,0x00,0x00,0xFF};
    unsigned int i=0,num=0,timeout=0;
    unsigned char temp=0;

    CMD18[1] = ((addr & 0xFF000000) >> 24);
    CMD18[2] = ((addr & 0x00FF0000) >> 16);
    CMD18[3] = ((addr & 0x0000FF00) >> 8);

    sdcard_disable();
    for (i=0;i<64;i++)
    {
        sdcard_read_byte();
    }
}
```



```
temp=sdcard_command2(CMD18);
if (temp!=0)
{
    return 1;
}

for(num=0;num<block_num;num++)
{
    timeout=0;
    do{
        temp=sdcard_read_byte();
        if(temp == 15)
            return(2);
        if(timeout++ > 10000)
            return(3);
    }while (temp != 0xff);
    timeout=0;

    do{
        temp=sdcard_read_byte();
        if(timeout++ > 10000)
            return(4);
    }while(temp != 0xfe);

#ifdef CONFIG_SSP_DMA
    if(hi_ssp_dmac_transfer(
        (dmac_ssp_s.ssprxchbufaddress +off+512*num),
        dmac_ssp_temp_s.txbufaddress,
        0x200) !=0)

        return 10;
    udelay(400);
#else

    for(i=0;i<512;i++)
    {
        *(buf+i+512*num) = sdcard_read_byte();
    }
#endif

    sdcard_read_byte();
    sdcard_read_byte();
}
```



```
        timeout=0;
        do{
            temp=sdcard_command2(CMD12);
            if(timeout++ >10000)
            {
                return (5);
            }
        }while(temp!=0);

        timeout=0;
        do{
            temp=sdcard_read_byte();
            if(timeout++ > 10000)
            {
                return(6);
            }
        }while (temp != 0xff);

        sdcard_disable();
        for(i=0;i<64;i++)
        {
            sdcard_read_byte();
        }

        return 0;
    }
```

【相关主题】

无。

2.2.1.14 hi_ssp_dmac_exit

【目的】

退出 SSP 的 DMA 传输模式。

【语法】

```
void hi_ssp_dmac_exit(void);
```

【参数】

无。

【返回值】

无。

【需求】



头文件：

- `#include <linux/config.h>`
- `#include <linux/module.h>`
- `#include <linux/kernel.h>`
- `#include <linux/init.h>`
- `#include <linux/miscdevice.h>`
- `#include <linux/proc_fs.h>`
- `#include <linux/poll.h>`
- `#include <asm/hardware.h>`
- `#include <asm/bitops.h>`
- `#include <asm/uaccess.h>`
- `#include <asm/irq.h>`
- `#include <asm/io.h>`
- `#include <linux/moduleparam.h>`
- `#include <linux/types.h>`
- `#include <linux/fs.h>`
- `#include <linux/ioport.h>`
- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hi_ssp.h>`

【注意】

无。

【举例】

无。

【相关主题】

无。

2.3 DMAC 接口驱动函数

DMAC 接口驱动模块提供以下功能：

- 执行 DMA 操作所需要的资源申请。
- DMAC 配置。
- 完成通道的获取、释放，链表空间的开辟、释放及链表的初始化。
- 执行 Memory 到 Memory、Memory 到外设、外设到 Memory DMA 传输的配置。

DMAC 接口驱动模块提供以下接口函数：

- [`dmac_channel_allocate`](#)：动态分配 DMA 通道。
- [`dmac_channel_free`](#)：释放指定的 DMA 通道。



- `dmac_register_isr`: 使用固定通道时, 注册中断回调函数。
- `allocate_dmali_space`: 申请 DMA 传输链表存储空间。
- `free_dmali_space`: 释放 DMA 传输链表存储空间。
- `dmac_start_m2m`: 配置 Memory 到 Memory 无链表传输模式的 DMAC 寄存器。
- `dmac_start_m2p`: 配置 Memory 到外设的无链表传输模式的 DMAC 寄存器。
- `dmac_buildllim2m`: 建立 Memory 到 Memory 的寄存器配置链表。
- `dmac_buildllim2p`: 建立 Memory 到外设的寄存器配置链表。
- `dmac_start_llim2m`: 配置 Memory 到 Memory 的链表传输模式的 DMAC 寄存器。
- `dmac_start_llim2p`: 配置 Memory 到外设链表传输模式的 DMAC 寄存器。
- `dmac_channelstart`: 启动 DMA 传输通道。
- `dmac_channelclose`: 关闭 DMA 传输通道。
- `dmac_wait`: 等待某个通道的一次 DMA 传输完成函数。



注意

- 在调用以上接口函数之前, 必须先调用 `DECLARE_KCOM_HI_DMAL`宏函数声明, 然后在调用模块的初始化函数中调用 `KCOM_HI_DMAL_INIT`宏函数后, 才能正常使用。
- 在卸载此调用模块前也要在此调用模块的退出函数中调用 `KCOM_HI_DMAL_EXIT`宏函数。
- 每个宏函数在每个模块中只能被调用一次, 多次调用可能会出错。调用方式和 I²C 模块相同。

2.3.1.1 dmac_channel_allocate

【目的】

动态分配 DMA 通道。

【语法】

```
int dmac_channel_allocate(void *pISR);
```

【参数】



参数名称	描述	输入/输出
pisr	<p>指向中断处理回调函数的指针。</p> <p>此函数指针所指向的函数原型如下：</p> <pre>typedef void REG_ISR(int *p_dma_chn,int *p_dma_status); REG_ISR *function[DMAC_MAX_CHANNELS];</pre> <p>其中：</p> <ul style="list-style-type: none">• p_dma_chn 表示指向产生中断的 DMA 通道号的指针。• p_dma_status 表示指向该通道状态的指针。	输入

【返回值】

返回值	描述
通道号	分配成功的通道号，范围为 0~7。
-EFAULT	获取通道失败，没有空闲通道。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hidmac.h>



【注意】

如果不需要中断处理函数，则输入指针为 NULL。

【举例】

无。

【相关主题】

无。

2.3.1.2 dmac_channel_free

【目的】

释放指定的 DMA 通道。

【语法】

```
int dmac_channel_free(unsigned int channel);
```

【参数】

参数名称	描述	输入/输出
channel	释放的 DMA 通道序号，范围为 0~7。	输入

【返回值】

返回值	描述
0	释放通道成功。
-EFAULT	释放通道失败。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>



- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hidmac.h>

【注意】

无。

【举例】

无。

【相关主题】

无。

2.3.1.3 dmac_register_isr

【目的】

当使用固定的 DMA 通道时，使用此函数注册中断回调函数。

【语法】

```
int dmac_register_isr(unsigned int channel,void *pISR);
```

【参数】

参数名称	描述	输入/输出
channel	DMA 传输的通道号，范围是 0~7。	输入
pISR	中断处理回调函数指针。此函数指针所指向的函数原型如下： <pre>typedef void REG_ISR(int *p_dma_chn,int *p_dma_status); REG_ISR *function[DMAC_MAX_CHANNELS];</pre> <ul style="list-style-type: none">• p_dma_chn 表示指向产生中断的 DMA 通道号的指针。• p_dma_status 表示指向该通道状态的指针。	输入

【返回值】



返回值	描述
0	成功。
-1	失败。

【需求】

头文件：

- `#include <linux/config.h>`
- `#include <linux/module.h>`
- `#include <linux/kernel.h>`
- `#include <linux/init.h>`
- `#include <linux/miscdevice.h>`
- `#include <linux/proc_fs.h>`
- `#include <linux/poll.h>`
- `#include <asm/hardware.h>`
- `#include <asm/bitops.h>`
- `#include <asm/uaccess.h>`
- `#include <asm/irq.h>`
- `#include <asm/io.h>`
- `#include <linux/moduleparam.h>`
- `#include <linux/types.h>`
- `#include <linux/fs.h>`
- `#include <linux/ioport.h>`
- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hidmac.h>`

【注意】

无。

【举例】

无。

【相关主题】

无。

2.3.1.4 allocate_dmali_space

【目的】

申请 DMA 传输链表存储空间。

【语法】



```
int allocate_dmalli_space(unsigned int *ppheadlli, unsigned char page_num);
```

【参数】

参数名称	描述	输入/输出
ppheadlli	用于存储链表结构的首地址。	输入
page_num	分配存储空间大小的系数。 每个存储空间大小为 PAGE_SIZE，总的存储空间大小为 page_num × PAGE_SIZE。	输入

【返回值】

返回值	描述
0	成功。
-1	失败。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hidmac.h>

【注意】



无。

【举例】

无。

【相关主题】

无。

2.3.1.5 free_dmali_space

【目的】

释放 DMA 传输链表存储空间。

【语法】

```
int free_dmali_space(unsigned int *ppheadlli, unsigned char page_num);
```

【参数】

参数名称	描述	输入/输出
ppheadlli	用于存储待释放的链表结构的首地址。	输入
page_num	待释放存储空间大小的系数。 每个存储空间大小为 PAGE_SIZE，总的存储空间大小为 page_num×PAGE_SIZE。	输入

【返回值】

返回值	描述
0	成功。
-1	失败。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>



- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hidmac.h>

【注意】

无。

【举例】

无。

【相关主题】

无。

2.3.1.6 dmac_start_m2m

【目的】

配置 Memory 到 Memory 无链表传输模式的 DMAC 寄存器。

【语法】

```
int dmac_start_m2m (unsigned int channel,  
                    unsigned int psource,  
                    unsigned int pdest,  
                    unsigned int uwnumtransfers);
```

【参数】

参数名称	描述	输入/输出
channel	DMA 传输的通道号，范围为 0~7。	输入
psource	源 Memory 区域首地址，为物理地址。	输入
pdest	目的 Memory 区域首地址，为物理地址。	输入
uwnumtransfers	需要传输的字节数。	输入

【返回值】



返回值	描述
0	DMAC 寄存器配置成功。
-EINVAL	DMAC 寄存器配置失败，输入参数不合法。

【需求】

头文件：

- `#include <linux/config.h>`
- `#include <linux/module.h>`
- `#include <linux/kernel.h>`
- `#include <linux/init.h>`
- `#include <linux/miscdevice.h>`
- `#include <linux/proc_fs.h>`
- `#include <linux/poll.h>`
- `#include <asm/hardware.h>`
- `#include <asm/bitops.h>`
- `#include <asm/uaccess.h>`
- `#include <asm/irq.h>`
- `#include <asm/io.h>`
- `#include <linux/moduleparam.h>`
- `#include <linux/types.h>`
- `#include <linux/fs.h>`
- `#include <linux/ioport.h>`
- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hidmac.h>`

【注意】

无。

【举例】

无。

【相关主题】

无。

2.3.1.7 dmac_start_m2p

【目的】

配置 Memory 到外设的无链表传输模式的 DMAC 寄存器。

【语法】



```
int dmac_start_m2p(unsigned int channel,
                  unsigned int pmemaddr,
                  unsigned int uwperipheralid,
                  unsigned int uwnumtransfers,
                  unsigned int next_ll_i_addr)
```

【参数】

参数名称	描述	输入/输出
channel	DMA 传输通道的序号，范围是 0~7。	输入
pmemaddr	Memory 的区域首地址，为物理地址。	输入
uwperipheralid	外设请求线序号，范围是 0~15。	输入
uwnumtransfers	要传输的数据字节数。	输入
next_ll_i_addr	下一个链表的首地址，如果没有下一个链表，则赋值为 0。	输入

【返回值】

返回值	描述
0	DMAC 寄存器配置成功。
-EINVAL	DMAC 寄存器配置失败，输入参数不合法。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>



- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hidmac.h>

【注意】

无。

【举例】

无。

【相关主题】

无。

2.3.1.8 dmac_buildllim2m

【目的】

建立 Memory 到 Memory 的寄存器配置链表。

【语法】

```
int dmac_buildllim2m(unsigned int *ppheadlli,  
                     unsigned int psource,  
                     unsigned int pdest,  
                     unsigned int totaltransfersize,  
                     unsigned int uwnumtransfers);
```

【参数】

参数名称	描述	输入/输出
ppheadlli	链表结构的首地址。	输入
psource	源 Memory 区域首地址，为物理地址。	输入
pdest	目的 Memory 区域首地址，为物理地址。	输入
totaltransfersize	传输的字节数。	输入
uwnumtransfers	一次 DMA 传输的字节数。	输入

【返回值】

返回值	描述
0	链表建立成功。



返回值	描述
-EINVAL	链表建立失败。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hidmac.h>

【注意】

无。

【举例】

无。

【相关主题】

无。

2.3.1.9 dmac_buildllim2p

【目的】

建立 Memory 到外设的寄存器配置链表。

【语法】

```
int dmac_buildllim2p(unsigned int *ppheadlli,
```



```
unsigned int *pmemaddr,  
unsigned int uwperipheralid,  
unsigned int totaltransfersize,  
unsigned int uwnumtransfers,  
unsigned int burstsize);
```

【参数】

参数名称	描述	输入/输出
ppheadlli	链表结构的首地址。	输入
pmemaddr	Memory 的区域首地址，为物理地址。 注意： 该指针是一个数组指针，数组下标为 2。	输入
uwperipheralid	外设请求线序号，范围是 0~15。	输入
totaltransfersize	传输的字节数。	输入
uwnumtransfers	一次 DMA 传输的字节数。	输入
burstsize	一次 Burst 传输的字节数。	输入

【返回值】

返回值	描述
0	链表建立成功。
-EINVAL	链表建立失败。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>



- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hidmac.h>

【注意】

无。

【举例】

无。

【相关主题】

无。

2.3.1.10 dmac_start_llim2m

【目的】

配置 Memory 到 Memory 的链表传输模式的 DMAC 寄存器。

【语法】

```
int dmac_start_llim2m(unsigned int channel, unsigned int *pfirst_lli);
```

【参数】

参数名称	描述	输入/输出
channel	DMA 传输通道的序号，范围为 0~7。	输入
pfirst_lli	指向链表结构（dmac_lli）的指针。	输入

链表结构 dmac_lli 定义如下：

```
typedef struct dmac_lli
{
    unsigned int src_addr;           /*source address*/
    unsigned int dst_addr;          /*destination address*/
    unsigned int next_lli;          /*pointer to next LLI*/
    unsigned int lli_transfer_ctrl; /*control word*/
}dmac_lli;
```

【返回值】



返回值	描述
0	DMAC 寄存器配置成功。
-EINVAL	DMAC 寄存器配置失败，输入参数不合法。

【需求】

头文件：

- `#include <linux/config.h>`
- `#include <linux/module.h>`
- `#include <linux/kernel.h>`
- `#include <linux/init.h>`
- `#include <linux/miscdevice.h>`
- `#include <linux/proc_fs.h>`
- `#include <linux/poll.h>`
- `#include <asm/hardware.h>`
- `#include <asm/bitops.h>`
- `#include <asm/uaccess.h>`
- `#include <asm/irq.h>`
- `#include <asm/io.h>`
- `#include <linux/moduleparam.h>`
- `#include <linux/types.h>`
- `#include <linux/fs.h>`
- `#include <linux/ioport.h>`
- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hidmac.h>`

【注意】

无。

【举例】

无。

【相关主题】

无。

2.3.1.11 dmact_start_llim2p

【目的】

配置 Memory 到外设链表传输模式的 DMAC 寄存器。

【语法】



```
int dmac_start_llim2p(unsigned int channel,  
                     unsigned int *pfirst_lli,  
                     unsigned int uwperipheralid);
```

【参数】

参数名称	描述	输入/输出
channel	DMA 传输通道的序号，范围为 0~7。	输入
pfirst_lli	指向链表结构（dmac_lli）的指针。	输入
uwperipheralid	外设请求线序号，范围是 0~15。	输入

【返回值】

返回值	描述
0	DMAC 寄存器配置成功。
-EINVAL	DMAC 寄存器配置失败，输入参数不合法。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>
- #include <asm/irq.h>
- #include <asm/io.h>
- #include <linux/moduleparam.h>
- #include <linux/types.h>
- #include <linux/fs.h>
- #include <linux/ioport.h>
- #include <linux/interrupt.h>
- #include <linux/kcom.h>
- #include <kcom/hidmac.h>



【注意】

无。

【举例】

无。

【相关主题】

无。

2.3.1.12 dmac_channelstart

【目的】

启动 DMA 传输通道。

【语法】

```
int dmac_channelstart(unsigned int u32channel);
```

【参数】

参数名称	描述	输入/输出
u32channel	使能通道的序号，范围为 0~7。	输入

【返回值】

返回值	描述
0	通道使能成功。
-EINVAL	通道使能失败，通道序号不合法。

【需求】

头文件：

- #include <linux/config.h>
- #include <linux/module.h>
- #include <linux/kernel.h>
- #include <linux/init.h>
- #include <linux/miscdevice.h>
- #include <linux/proc_fs.h>
- #include <linux/poll.h>
- #include <asm/hardware.h>
- #include <asm/bitops.h>
- #include <asm/uaccess.h>



- `#include <asm/irq.h>`
- `#include <asm/io.h>`
- `#include <linux/moduleparam.h>`
- `#include <linux/types.h>`
- `#include <linux/fs.h>`
- `#include <linux/ioport.h>`
- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hidmac.h>`

【注意】

无。

【举例】

无。

【相关主题】

无。

2.3.1.13 `dmac_channelclose`

【目的】

关闭 DMA 传输通道。

【语法】

```
int dmac_channelclose(unsigned int channel);
```

【参数】

参数名称	描述	输入/输出
<code>u32channel</code>	关闭通道的序号，范围为 0~7。	输入

【返回值】

返回值	描述
0	通道关闭成功。
-EINVAL	通道关闭失败，通道序号不合法。

【需求】

头文件：

- `#include <linux/config.h>`



- `#include <linux/module.h>`
- `#include <linux/kernel.h>`
- `#include <linux/init.h>`
- `#include <linux/miscdevice.h>`
- `#include <linux/proc_fs.h>`
- `#include <linux/poll.h>`
- `#include <asm/hardware.h>`
- `#include <asm/bitops.h>`
- `#include <asm/uaccess.h>`
- `#include <asm/irq.h>`
- `#include <asm/io.h>`
- `#include <linux/moduleparam.h>`
- `#include <linux/types.h>`
- `#include <linux/fs.h>`
- `#include <linux/ioport.h>`
- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hidmac.h>`

【注意】

无。

【举例】

无。

【相关主题】

无。

2.3.1.14 dmac_wait

【目的】

当 DMAC 开始传输后，若不在中断方式进行下次传输而采用查询方式时，调用此函数等待此次 DMA 传输完成。

【语法】

```
int dmac_wait(unsigned int channel);
```

【参数】

参数名称	描述	输入/输出
channel	DMA 传输的通道号，范围为 0~7。	输入

【返回值】



返回值	描述
0	成功。
-1	失败。

【需求】

头文件：

- `#include <linux/config.h>`
- `#include <linux/module.h>`
- `#include <linux/kernel.h>`
- `#include <linux/init.h>`
- `#include <linux/miscdevice.h>`
- `#include <linux/proc_fs.h>`
- `#include <linux/poll.h>`
- `#include <asm/hardware.h>`
- `#include <asm/bitops.h>`
- `#include <asm/uaccess.h>`
- `#include <asm/irq.h>`
- `#include <asm/io.h>`
- `#include <linux/moduleparam.h>`
- `#include <linux/types.h>`
- `#include <linux/fs.h>`
- `#include <linux/ioport.h>`
- `#include <linux/interrupt.h>`
- `#include <linux/kcom.h>`
- `#include <kcom/hidmac.h>`

【注意】

无。

【举例】

无。

【相关主题】

无。



3 DMAC 程序示例

关于本章

本章描述内容如下表所示。

标题	内容
3.1 链表方式的 DMA 传输	详细描述链表方式的 DMA 传输。
3.2 无链表方式的 DMA 传输	详细描述无链表方式的 DMA 传输。



3.1 链表方式的 DMA 传输

3.1.1 配置流程

链表方式的 DMA 传输配置流程如下：

- 步骤 1 申请 DMA 通道。
- 步骤 2 申请链表空间。
- 步骤 3 建立链表。
- 步骤 4 执行一次传输。
- 步骤 5 释放链表空间。
- 步骤 6 关闭并释放 DMA 通道。

----结束

3.1.2 实例代码

本节通过调用 DMAC 接口函数演示如何采用链表方式进行 AI 的数据传输。



说明

这里只是演示了 AI 采用 DMAC 的链表方式进行数据传输调用 DMAC 接口函数部分的代码，并不是 AI 的整个代码！

```
/* 1 申请DMA通道*/
HI_S32 AI_TRANS_Init(AUDIO_DEV AudioDevId)
{
    SIO_DMA_S *pAIDMA = &g_stAIDMA[AudioDevId];
    char acName[] = "AIDMA";
    HI_S32 s32Ret = HI_SUCCESS;

    /*the max ai buf size,include dma size(ping pong buff),channel buffer
and aec buffer*/
    HI_U32 u32Maxsize = 2*sizeof(SIO_FIFO_U)*MAX_AUDIO_POINT_NUM +
        2*4*(MAX_SIO_CHN*MAX_AUDIO_POINT_NUM*MAX_AUDIO_FRAME_NUM);

    /*alloc dma buf and all channel buf*/
    pAIDMA->astDMABuf[0].u32DMABufAddr = g_pAIMmb->
new_mmb(acName,u32Maxsize,0,NULL);
    if (MMB_ADDR_INVALID == pAIDMA->astDMABuf[0].u32DMABufAddr)
    {
        HI_TRACE_SIO(HI_DBG_ERR,"alloc dma buf and all channel buf err\n");
        return HI_ERR_AI_NOMEM;
    }
}
```



```
pAIDMA->astDMABuf[1].u32DMABufAddr =
sizeof(SIO_FIFO_U)*MAX_AUDIO_POINT_NUM
+ pAIDMA->astDMABuf[0].u32DMABufAddr;

/*ioremap*/
pAIDMA->astDMABuf[0].pDMABuf = ioremap_nocache(pAIDMA-
>astDMABuf[0].u32DMABufAddr,u32Maxsize);
pAIDMA->astDMABuf[1].pDMABuf = MAX_AUDIO_POINT_NUM + pAIDMA-
>astDMABuf[0].pDMABuf;
if(NULL == pAIDMA->astDMABuf[0].pDMABuf)
{
    HI_TRACE_SIO(HI_DBG_ERR,"ioremap dma buff failed\n");
    s32Ret = HI_ERR_AI_NOMEM;
    goto freephys;
}
memset(pAIDMA->astDMABuf[0].pDMABuf,0,u32Maxsize);

/*alloc dmac requester*/
switch(AudioDevId)
{
case 0:
    pAIDMA->u32DMARequester = DMAC_SIO0_RX_REQ;
    break;
case 1:
    pAIDMA->u32DMARequester = DMAC_SIO1_RX_REQ;
    break;
default:
    HI_TRACE_SIO(HI_WARN_LEVEL(255),"dev num overflow\n");
    s32Ret = HI_ERR_AI_INVALID_DEVID;
    goto unmap;
}

/*alloc dma channel*/
pAIDMA->s32DMAChannel = dmac_channel_allocate(AIDMAIsr);
if(pAIDMA->s32DMAChannel < 0)
{
    HI_TRACE_SIO(HI_DBG_ERR,"alloc dma channel failed\n");
    s32Ret = HI_ERR_AI_NOMEM;
    goto unmap;
}

/*alloc channel buf*/
pAIDMA->pAllChannelBuf = ioremap_cached(pAIDMA-
>astDMABuf[0].u32DMABufAddr + 2*sizeof(SIO_FIFO_U)*MAX_AUDIO_POINT_NUM,
```




```

        u32Maxsize -
2*sizeof(SIO_FIFO_U)*MAX_AUDIO_POINT_NUM);
    if(NULL == pAIDMA->pAllChannelBuf)
    {
        HI_TRACE_SIO(HI_DBG_ERR, "ioremap all chn buff failed\n");
        s32Ret = HI_ERR_AI_NOMEM;
        goto unmap;
    }

    pAIDMA->bStart = HI_FALSE;
    init_MUTEX(&pAIDMA->stSem);

#ifdef AUDIO_USETASKLET
    tasklet_init(&pAIDMA->stTasklet, AIDMATasklet, AudioDevId);
#endif

    pAIDMA->bInited = HI_TRUE;
    return HI_SUCCESS;

unmap:
    iounmap(pAIDMA->astDMABuf[0].pDMABuf);
freephys:
    g_pAImmb->delete_mmb(pAIDMA->astDMABuf[0].u32DMABufAddr);

    pAIDMA->astDMABuf[0].u32DMABufAddr = 0;
    return s32Ret;

}
/* 2 申请链表空间并建立链表 */
static HI_S32 AIDMAInit(AUDIO_DEV AudioDevId)
{
#ifdef _USE_LLI_/*use lli trans mode*/
    SIO_DMA_S *pAIDMA = &g_stAIDMA[AudioDevId];
    HI_U32 i;
    for(i=0; i<2; i++)
    {
        if(HI_SUCCESS != allocate_dmalli_space(pAIDMA->astDMABuf[i].ppheadlli,2) )
        {
            HI_TRACE_SIO(HI_INFO_LEVEL(255), "alloc dmallli_space failed\n");
            goto FREELLI;
        }
        if(HI_SUCCESS != dmac_buildllim2p(pAIDMA->astDMABuf[i].ppheadlli, &pAIDMA->astDMABuf[i].u32DMABufAddr,

```



```
        pAIDMA->u32DMARequester, pAIDMA->u32TransLen, 64, 0))
    {
        HI_TRACE_SIO(HI_INFO_LEVEL(255), "build dmaller failed\n");
        goto FREELLI;
    }
}
return HI_SUCCESS;
FREELLI:
for(i=0; i<2; i++)
{
    if(NULL != pAIDMA->astDMABuf[i].ppheadlli)
    {
        (void) free_dmaller_space(pAIDMA->astDMABuf[i].ppheadlli, 2);
    }
}
return HI_FAILURE;
#endif
}
/* 3 执行一次DMA传输 */
static HI_S32 AIDMAStart(AUDIO_DEV AudioDevId)
{
    SIO_DMA_S *pAIDMA = &g_stAIDMA[AudioDevId];

    /*if pinpon buf overflow, return*/
    if(pAIDMA->u32PinPon > 1)
    {
        return HI_FAILURE;
    }

#ifdef _USE_LLI_ /*use lli trans mode*/
    /*configure next dma trans*/
    if(HI_SUCCESS != dmacer_start_llim2p(pAIDMA->s32DMACHannel, pAIDMA-
>astDMABuf[pAIDMA->u32PinPon].ppheadlli,
        pAIDMA->u32DMARequester))
    {
        HI_TRACE_SIO(HI_WARN_LEVEL(255), "start dmaller failed\n");
        return HI_FAILURE;
    }
#else
    if(HI_SUCCESS != dmacer_start_m2p(pAIDMA->s32DMACHannel, pAIDMA-
>astDMABuf[pAIDMA->u32PinPon].u32DMABufAddr,
        pAIDMA->u32DMARequester, pAIDMA->u32TransLen))
    {
        HI_TRACE_SIO(HI_WARN_LEVEL(255), "dmacer_start_m2p error\n");
    }
}
```



```

        return HI_FAILURE;
    }
#endif

    pAIDMA->u32PinPon = (pAIDMA->u32PinPon == 0);
    (void)dmac_channelstart(pAIDMA->s32DMACchannel);
    return HI_SUCCESS;
}

/* 4 释放链表空间 */
static void AIDMARelease(AUDIO_DEV AudioDevId)
{
#ifdef _USE_LLI_ /* use lli trans mode */
    SIO_DMA_S *pAIDMA = &g_stAIDMA[AudioDevId];
    HI_U32 i;
    for(i=0; i<2; i++)
    {
        if(NULL != pAIDMA->astDMABuf[i].ppheadlli)
        {
            (void)free_dmali_space(pAIDMA->astDMABuf[i].ppheadlli, 2);
        }
    }
    /* clean up dma buf */
    if(pAIDMA->astDMABuf[0].pDMABuf != NULL)
    {
        memset(pAIDMA->astDMABuf[0].pDMABuf, 0, pAIDMA->u32TransLen * 2);
    }
#endif
}

/* 5 关闭并释放DMA通道 */
static void AIDMAStop(AUDIO_DEV AudioDevId)
{
    SIO_DMA_S *pAIDMA = &g_stAIDMA[AudioDevId];
    /* close dma channel */
    (void)dmac_channelclose(pAIDMA->s32DMACchannel);
}

```

3.2 无链表方式的 DMA 传输

3.2.1 内存到内存

此例通过调用 DMAC 接口函数演示内存到内存无链表方式的 DMA 传输。



```
int dmam_m2m_transfer(unsigned int *psource,
                      unsigned int *pdest,
                      unsigned int uwtransfersize)
{
    unsigned int ulChnn, dmaSize = 0;
    unsigned int dmaCount, leftSize;
    leftSize = uwtransfersize;
    dmaCount = 0;
    ulChnn = dmam_channel_allocate(NULL);
    if(DMAC_CHANNEL_INVALID == ulChnn)
        return -1;

    while((leftSize >> 2) >= 0xffc)
    {
        dmaSize = 0xffc;
        leftSize -= dmaSize*4;
        dmam_start_m2m(ulChnn,
                       (unsigned int)(psource + dmaCount * dmaSize),
                       (unsigned int)(pdest + dmaCount * dmaSize),
                       (dmaSize << 2));
        if(dmam_channelstart(ulChnn)!=0)
            return -1;
        if(dmam_wait(ulChnn) != DMAC_CHN_SUCCESS)
            return -1;
        dmaCount ++;
    }
    dmam_start_m2m(ulChnn,
                   (unsigned int)(psource + dmaCount * dmaSize),
                   (unsigned int)(pdest + dmaCount * dmaSize),
                   leftSize);
    if(dmam_channelstart(ulChnn)!=0)
        return -1;
    if(dmam_wait(ulChnn) != DMAC_CHN_SUCCESS)
        return -1;
    return 0;
}
```

3.2.2 内存到外设

此例通过调用 DMAC 接口函数演示内存到外设无链表方式的 DMA 传输。

```
int dmam_m2p_transfer(unsigned int *pmemaddr,
                      unsigned int uwperipheralid,
                      unsigned int uwtransfersize)
{
    }
```



```
unsigned int ulChnn, dmaSize = 0;
unsigned int dmaCount, leftSize ,uwidth;
leftSize = uwtransfersize;
dmaCount = 0;
ulChnn = dmac_channel_allocate(NULL);
if(DMAC_CHANNEL_INVALID == ulChnn)
    return -1;

if((uwtransfersize > (MAXTRANSFERSIZE<<2)) || (uwtransfersize&0x3))
{
    printk("Invalidate transfer size,size=%x \n",uwtransfersize);
    return -EINVAL;
}

if((DMAC_UART0_TX_REQ == uwperipheralid)
    || (DMAC_UART0_RX_REQ == uwperipheralid)
    || (DMAC_UART1_TX_REQ == uwperipheralid)
    || (DMAC_UART1_RX_REQ == uwperipheralid)
    || (DMAC_SSP_TX_REQ==uwperipheralid)
    || (DMAC_SSP_RX_REQ==uwperipheralid))
{
    uwidth=0;
}

else if((DMAC_SIO0_TX_REQ == uwperipheralid)
        || (DMAC_SIO0_RX_REQ == uwperipheralid))
{
    uwidth=1;
}

else
{
    uwidth=2;
}

while((leftSize >> uwidth) >= 0xffc)
{
    dmaSize = 0xffc;
    leftSize -= dmaSize*2*uwidth;
    dmac_start_m2p(ulChnn,
                   (unsigned int)(pmemaddr + dmaCount *dmaSize),
                   uwperipheralid, (dmaSize << 2), 0);
    if(dmac_channelstart(ulChnn) != 0)
        return -1;
}
```



```
        if(dmac_wait(ulChnn) != DMAC_CHN_SUCCESS)
            return -1;
        dmaCount ++;
    }

    dmac_start_m2p(ulChnn, (unsigned int)(pmemaddr + dmaCount *dmaSize),
                   uwperipheralid, leftSize, 0);
    if(dmac_channelstart(ulChnn) != 0)
        return -1;
    if(dmac_wait(ulChnn) != DMAC_CHN_SUCCESS)
        return -1;
    return 0;
}
```



A 缩略语

C

CPU	Central Processing Unit	中央处理器
------------	-------------------------	-------

D

DMA	Direct Memory Access	直接存储器存取
------------	----------------------	---------

DMAC	Direct Memory Access Controller	直接存储器存取控制器
-------------	---------------------------------	------------

F

FIFO	First In First Out	先入先出
-------------	--------------------	------

G

GPIO	General Purpose Input/Output	通用输入输出接口
-------------	------------------------------	----------

I

I2C	Inter-Integrated Circuit	一种串行总线协议标准
------------	--------------------------	------------

L

LED	Light Emitting Diode	发光二极管
------------	----------------------	-------

S

SD	Security Digital	加密数据卡
-----------	------------------	-------

SPI	Synchronous Physical Interface	同步物理接口
------------	--------------------------------	--------

SSP	Synchronous Serial Protocol	同步串口
------------	-----------------------------	------



T

TI

Texas Instrument

德州仪器