详解ARM的AMBA设备中的DMA设备PL08X的Linux驱动

版本：v1.1

Crifan Li

摘要

本文主要分析了Linux的DMA驱动中ARM的PL08X的实现细节，几乎详细分析了每一个函数的主体代码，同时介绍了DMA的基本概念和工作原理。

本文提供多种格式供：

在线阅读	HTML [1]	HTMLs [2]	PDF [3]	CHM [4]	TXT [5]	RTF [6]	WEBHELP [7]
下载（7zip压缩包）	HTML [8]	HTMLs [9]	PDF [10]	CHM [11]	TXT [12]	RTF [13]	WEBHELP [14]

HTML版本的在线地址为：

http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/html/dma_pl08x_analysis.html

有任何意见，建议，提交bug等，都欢迎去讨论组发帖讨论：

http://www.crifan.com/bbs/categories/dma_pl08x_analysis/

修订历史		
修订 1.0	2010-04-23	crl
1. 详解ARM的AMBA设备中的DMA设备PL08X的Linux驱动		
修订 1.1	2012-09-01	crl
1. 通过Docbook发布		

[1] http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/html/dma_pl08x_analysis.html
[2] http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/htmls/index.html
[3] http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/pdf/dma_pl08x_analysis.pdf
[4] http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/chm/dma_pl08x_analysis.chm
[5] http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/txt/dma_pl08x_analysis.txt
[6] http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/rtf/dma_pl08x_analysis.rtf
[7] http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/webhelp/index.html
[8] http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/html/dma_pl08x_analysis.html.7z
[9] http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/htmls/index.html.7z
[10] http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/pdf/dma_pl08x_analysis.pdf.7z
[11] http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/chm/dma_pl08x_analysis.chm.7z
[12] http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/txt/dma_pl08x_analysis.txt.7z
[13] http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/rtf/dma_pl08x_analysis.rtf.7z
[14] http://www.crifan.com/files/doc/docbook/dma_pl08x_analysis/release/webhelp/dma_pl08x_analysis.webhelp.7z

详解ARM的AMBA设备中的DMA设备PL08X的Linux驱动:

Crifan Li

版本：v1.1

出版日期 2012-09-01

版权 © 2012 Crifan, http://crifan.com

本文章遵从：署名-非商业性使用 2.5 中国大陆(CC BY-NC 2.5)[15]

[15] http://www.crifan.com/files/doc/docbook/soft_dev_basic/release/html/soft_dev_basic.html#cc_by_nc

目录

正文之前 v
1. 此文目的 v
2. 适合读者 v
3. 声明 v
1. ARM的PL08X的代码的详细解析 第一部分 1
1.1. PL08X Documentation 1
1.2. DMA简述 1
1.2.1. 什么是DMA 1
1.2.2. DMA的一些基础概念 2
1.3. PL08X asynchronous transfer 2
1.4. PL08X Memory to peripheral transfer 3
1.5. PL08X dma_device 3
1.6. PL08X _cctl_data _cctl 5
1.7. PL08X _lli _chan_lli pl08x_driver_data 7
1.8. PL08X macros 9
1.9. pl08x_decode_widthbits 11
1.10. pl08x_encode_width 12
1.11. pl08x_choose_master_bus 12
1.12. pl08x_fill_lli_for_desc 13
1.13. pl08x_pre_boundary 14
2. ARM的PL08X的代码的详细解析 第二部分 16
2.1. fill_LLIS_for_desc 16
2.2. pl08x_set_cregs 22
2.3. pl08x_memrelease 22
2.4. pl08x_disable_dmac_chan 23
2.5. pl08x_enable_dmac_chan 24
2.6. pl08x_dma_busy 25
2.7. pl08x_wqfunc 25
2.8. pl08x_amba_driver 26
2.9. pl08x_make_LLIs 26
2.10. pl08x_irq 27
2.11. pl08x_ensure_on 29
2.12. pl08x_dma_enumerate_channels 29
2.13. pl08x_probe 30
2.14. pl08x_init 32
3. ARM的PL08X的代码的详细解析 第三部分 33
3.1. 简述PL08X的DMA驱动的基本流程 33
3.2. pl08x_alloc_chan_resources 33
3.3. pl08x_free_chan_resources 34
3.4. pl08x_tx_submit 35
3.5. pl08x_prep_dma_memcpy 36
3.6. pl08x_prep_dma_interrupt 37
3.7. pl08x_dma_is_complete 37
3.8. pl08x_issue_pending 38
3.9. pl08x_prep_slave_sg 39
3.10. pl08x_terminate_all 40

插图清单

1.1. LLI寄存器的含义 8
1.2. Souce Transfer Width 11
1.3. Souce or Destination Transfer Width 12
1.4. LLI寄存器的含义 14
2.1. 配置寄存器的含义 27

正文之前

1. 此文目的

记录笔者对ARM的PL08x的DMA驱动PL08x.c理解。

给其他不熟悉此DMA驱动的读者一点借鉴和参考。

2. 适合读者

你已经具备一定驱动编程能力，知道一些最基本的概念，比如用于输出输出数据的设备的FIFO等，一般设备所具有的比如DATA等寄存器

希望对ARM的PL080的DMA驱动的工作流程有深入的了解，希望知道如何使用此DMA驱动

3. 声明

如果此文档中任何内容，侵犯了您的版权，涉及任何法律问题，请立即通知笔者，笔者会立即删除

本文所涉及的代码，获取自[RFC PATCH 1/3] ARM: PL08X: Add a driver for the PL08x series of DMAC controllers.[1]

刚刚又发现，此驱动，有了最新的2010的版本，感兴趣的可以在这里找到其源码：Linux/drivers/dma/amba-pl08x.c[2]

相关的数据手册，可以去这里找到：第 1.1 节 “PL08X Documentation”

水平有限，难免理解有误，欢迎指正和交流：admin (at) crifan.com

[1] http://www.spinics.net/linux/lists/arm-kernel/msg60664.html
[2] http://tomoyo.sourceforge.jp/cgi-bin/lxr/source/drivers/dma/amba-pl08x.c

第 1 章 ARM的PL08X的代码的详细解析第一部分

1.1. PL08X Documentation

```
/* Copyright(c) 2006 ARM Ltd.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the Free
 * Software Foundation; either version 2 of the License, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc., 59
 * Temple Place - Suite 330, Boston, MA  02111-1307, USA.
 *
 * The full GNU General Public License is iin this distribution in the
 * file called COPYING.
 *
 * Documentation: ARM DDI 0196G❶ == PL080❷
 * Documentation: ARM DDI 0218E == PL081❸
*
 */
```

❶ 用DDI0196表示此pl08x.c dma驱动对应的datasheet
❷ 对应文档下载地址：PL080[1]
❸ 对应文档下载地址：PL081[2]

Datasheet

又称数据手册，是关于硬件资源，功能等详细描述的手册，是做相关驱动开发必不可少的资料，否则，你都不知道你的硬件能做什么事情，就没法去写对应的驱动了。

1.2. DMA简述

在开始分析代码之前，先简要介绍一下DMA的基础知识。

1.2.1. 什么是DMA

DMA，Direct Memory Access，直接内存访问。

既然叫 直接内存访问，那么相对应地，应该就有 “间接的内存访问”。

间接的内存访问，我的理解是，就是指最常见的，我们利用CPU的指令，去从一个内存地址中读出数据，然后写到另外一个内存地址中，完成对应的赋值操作。

[1] http://infocentre.arm.com/help/topic/com.arm.doc.ddi0196g/DDI0196.pdf
[2] http://infocentre.arm.com/help/topic/com.arm.doc.ddi0218e/DDI0218.pdf

此过程，完全都是CPU去操作的，如果是单个这样的数据读取和写入，还没啥，但是如果数据量很大，比如我们用memcpy(addr1,addr2，1024)去从地址addr1地址开始，拷贝1024个字节到内存addr2处，那么CPU这段时间，就不要干别的事情了，就一直这么的给你读取，写入数据吧，另外的还有，常见于驱动中的，尤其是涉及到和外设打交道，我们让CPU从内存一个地址，读取了一个数据，然后写入到某个设备的FIFO或者DATA寄存器中，咋写入之前，常常会等待FIFO不是满的，然后才能写入数据，要从FIFO中读取数据，要等到FIFO不是空，才能读取，这样来来回回，会比较消耗资源。

鉴于此，才出现了DMA这个硬件设备，专门设计用来处理这些相对用CPU去操作这样的事情，效率很低，换做专门的硬件的DMA来负责数据的读取和写入，释放了CPU这个苦力，可以让，在DMA忙着数据传输的过程中，CPU去忙其他更重要的事情。而专门的DMA硬件负责这样的数据传输，效率也会更高。

之所以这样，才叫做内存直接访问的。

1.2.2. DMA的一些基础概念

DMA传输，总的来说就是：

硬件上，会有对应的控制寄存器ctrl和配置寄存器config，

比如你想要从内存一个地址addr传输，N个字word（32bit）的数据到设备dev上，

那么你就要先去根据你的请求，去配置config寄存器，首先是传输方向，

是DMA_TO_DEVICE,然后是源地址source address是你的内存地址addr，

和目标destination address是你的dev的DATA寄存器地址，

然后要传输额transfer size是N个，每个位宽是32bit，

将源地址，目标地址，位宽，DMA传输方向设置好，

整理成一个结构，专有名称叫做LLI（Link List Item），把这个LLi设置到ctrl里面。

然后去enable DMA，DMA就可以按照你的要求，把数据传输过去了。

这样的DMA叫做single DMA传输，LLI中的next lli的域设置为空，表示就一个LLI要传输。

如果源地址或目标地址是多个分散的地址，叫做scatter/gather DMA，

就要将这些LLI组合一下，即将第一个LLI的next lli那个域，设置成下一个LLI的地址，

这样一个个链接起来，最后一个LLI的nex lli的域为空，这样设置好后，将第一个LLI的值写入到ctrl中，DMA就会自动地去执行第一个LLI的数据传输，传完后，发现next lli不为空，就找到next lli的位置，

找到对应的配置，开始这个lli的数据传送，直至传完所有的数据为止。

说完了DMA的由来和基本概念后，下面来分析一下，具体的ARM的PL08x驱动是如何实现的。

1.3. PL08X asynchronous transfer

```
/*
 * The AMBA DMA API is modelled on the ISA DMA API and performs❶
 * single asynchronous❷ transfers between a device and memory
 * i.e. some platform fixed device address and a driver defined memory address❸
```

❶ 此AMBA DMA驱动，基于ISA DMA的API，好像应该就是那个DMA engine的架构吧，对应的，是这两个相关文件：

\include\linux\dmaengine.h

\drivers\dma\dmaengine.c

❷ 主要实现了异步传输，细看内部实现，就是，你设置好所有的参数之后，就提交你的请求后，然后此dma驱动会去在合适的时候帮你实现你的dma请求。因此，不保证是立刻就去执行你的请求的，此之所以称作异步。

❸ 正如上面的解释，常见的应用就是，

对应某个外设有某个固定的设备地址，一般都是某个FIFO的地址，或者DATA之类的寄存器，然后你的DMA请求是，从内存某个地址传输一定数据到你这个设备的FIFO或者data寄存器，即往你设备里面写数据，或者相反，从你的设备的FIFO地址中，读取一定量数据到内存某个位置，即从你设备里面读取数据。

1.4. PL08X Memory to peripheral transfer

```
*
* Memory to peripheral transfer may be visualized as
*  Get data from memory to DMAC
* Until no data left
*  On burst request from peripheral
*   Destination burst from DMAC to peripheral
*   Clear burst request
* Raise terminal count interrupt
*
* For peripherals with a FIFO:❶
* Source      burst size == half the depth of the peripheral FIFO
* Destination burst size == width of the peripheral FIFO
```

❶ 关于提交DMA传输请求的时候，对于突发传输大小（burst size）的设置，虽然此处建议对于source burst size，设置成你的FIFO大小的一半，而对于destination burst size，设置为你的FIFO大小等同，但是，实际一般是根据你的外设控制器的设置而去具体设置的

比如你的nand flash控制器有个fifo是36个word，但是，其nand flash controller中关于burst size的说明是，，DMA模式时候，当fifo中小于4个word并且将要写入数据大于32个word的时候，才会发送write burst信号给dma，要求burst传输，期望一下子传输32个word，

这样，一下子传输32个word，写入到nand flash的fifo里面，这样就比dma传输一次一个word，即single word transfer的效率高多了。

此时，你的destination burst size，就应该设置成32个word，之前，我以为也可以设置成16，8之类比32小的值，但是除了理论上理解的，没有充分利用硬件的能力、效率低之外，，实际上，驱动并不能正常工作，数据总是少不部分，就是说，硬件上人家有burst的dma请求了，就是已经准备了32个数据让你传，结果你只传输了部分，所以数据就少了一些，就乱了。

一般来说，source burst size，多数和destination burst size相等。具体，还要去看你的设备的datasheet。

1.5. PL08X dma_device

```
*
* (Bursts are irrelevant for mem to mem transfers - there are no burst signals)
*
* Details of each tranfer on a particular channel
* are held in the DMA channel array
* A workqueue uses those details to initiate the actual transfer
* The DMAC interrupt clears the relevant transfer in the channel array
```

```
 *
 * ASSUMES only one DMAC device exists in the system
 * ASSUMES default (little) endianness for DMA transfers
 *
 * Only DMAC flow control is implemented
 *
 */
#include <linux/device.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/interrupt.h>

#include <linux/workqueue.h>

#include <linux/dmapool.h>
#include <asm/dma.h>
#include <asm/mach/dma.h>
#include <asm/atomic.h>
#include <asm/processor.h>
#include <linux/amba/bus.h>

#include <linux/dmaengine.h>
#include <asm/cacheflush.h>

#include <linux/amba/pl08x.h>
int ctr_chan[2];

/*
 * Predeclare the DMAENGINE API functions
 */
static int pl08x_alloc_chan_resources(struct dma_chan *chan,
      struct dma_client *client);
static void pl08x_free_chan_resources(struct dma_chan *chan);
static struct dma_async_tx_descriptor *pl08x_prep_dma_memcpy(
   struct dma_chan *chan, dma_addr_t dest, dma_addr_t src,
    size_t len, unsigned long flags);
static struct dma_async_tx_descriptor *pl08x_prep_dma_interrupt(
   struct dma_chan *chan, unsigned long flags);
static enum dma_status pl08x_dma_is_complete(struct dma_chan *chan,
     dma_cookie_t cookie, dma_cookie_t *last,
      dma_cookie_t *used);
static void pl08x_issue_pending(struct dma_chan *chan);
static struct dma_async_tx_descriptor *pl08x_prep_slave_sg(
  struct dma_chan *chan, struct scatterlist *sgl,
  unsigned int sg_len, enum dma_data_direction direction,
  unsigned long flags);
static void pl08x_terminate_all(struct dma_chan *chan);

struct dma_device❶ dmac = {
 .device_alloc_chan_resources = pl08x_alloc_chan_resources,
 .device_free_chan_resources = pl08x_free_chan_resources,
 .device_prep_dma_memcpy  = pl08x_prep_dma_memcpy,
 .device_prep_dma_xor  = NULL,
 .device_prep_dma_zero_sum = NULL,
 .device_prep_dma_memset  = NULL,
 .device_prep_dma_interrupt = pl08x_prep_dma_interrupt,
 .device_is_tx_complete  = pl08x_dma_is_complete,
```

```
.device_issue_pending  = pl08x_issue_pending,
.device_prep_slave_sg  = pl08x_prep_slave_sg,
.device_terminate_all  = pl08x_terminate_all,
};
```

❶ 这个dma_device是在dmaengine.h中定义的

其就是定义了一个架构，实现了DMA驱动的与具体设备无关的部分，然后提供具体接口，然后你具体的DMA驱动，去实现对应的接口

这样的好处是，省却了你的具体驱动，去关心太多那些通用dma驱动都应该要实现的一些功能等，省却了具体驱动编程者的精力

1.6. PL08X _cctl_data _cctl

```
/*
 * PL08X private data structures  ==== START
 */
struct _cctl_data{
 unsigned int tsize:12;❶
 unsigned int sbsize:3;
 unsigned int dbsize:3;❷
 unsigned int swidth:3;
 unsigned int dwidth:3;❸
 unsigned int smaster:1;
 unsigned int dmaster:1;❹
 unsigned int si:1;
 unsigned int di:1;❺
 unsigned int prot:3;❻
 unsigned int intr:1;❼
};
union❽ _cctl{
 struct _cctl_data bits;
 unsigned int val;
};
```

❶ 即TransferSize，传输大小，bit0-bit11，共12位。

表示，如果当前传输是DMA控制的情况下，当前此次DMA传输要传输的个数，其中每个具体大小是几个字节，由下面的位宽决定。

在配置完dma，开始传输后，此传输大小从你设置的值，即要传输的数，一点点减少直至0.如果读取此域，得到的值，表示当前还剩下多少个要传输。

不过特殊一点是，如果当前正在进行DMA传输，由于正在传输，所以你读取到的值，并不一定是真正的还剩多少个要传输的。一般用法是，在启用DMA传输，后来又由于传输完成后或者出错等特殊情况又禁用了DMA，此时再去读此域的值，就能真正有效地表示还多少没有传输的。

如果不是DMA控制数据流的传输，（之后会提到，关于DMA数据传输方向，除了DMA_TO_DEVICE，DMA_FROM_DEVICE之外，还有MEM_TO_MEM之类的，非DMA控制数据传输的情况）那么就应该在开始配置的时候，将此域值设置成0.

❷ 即source burst size和destination burst size，即前面提到的，当设备发送给DMA控制器一个突发传输信号之后，DMA要传输多少个数据，就由此处设置的值决定。

支持的burst size有1，4，8，。。。，128，256。

前面已经解释过了，具体此处需要设置成多少，要根据你的硬件的datasheet中描述的你的硬件的能力去决定。

Datasheet中写的DMACxBREQ信号线，就是设备的控制器，如果支持DMA，都会有此信号线接出来的，这样，接到DMA的对应的DMACxBREQ引脚上，这样，如果设备控制器，发现当前的条件，满足DMA burst传输，比如FIFO一共36个word，发现FIFO中数据&=32个word了，快满了，就会向DMA控制器PL080，通过这个信号线发送burst read请求，然后PL080就会根据你DMA传输前此处设置的值，比如是32，去你的设备的FIFO中，一次性地读取你之前设置的32个word。

其中具体满足什么条件，才会触发DMA burst读取或写入的请求信号，都是对应你的设备的控制器的datasheet中描述的，也就是你的硬件本身的能力决定的。.

❸ 即source width和destination width。

具体支持的位宽类型有，byte（8bit），half word（16bit），word（32bit）等。

关于source和destination，解释一下，

比如，你要从你的设备读取数据，即从你设备的FIFO中读取数据到内存某个位置，那么你的设备就是source，此时你的设备的FIFO的位宽，就是source width，比如你FIFO位宽是32bit的，那么此处，根据datasheet，就应该设置为word（32bit）。

❹ 即source master和destination master，当你的传输由DMA控制时，

比如上面说的DMA_FROM_DEVICE，DMA从设备的FIFO中读取数据到内存中，此时，你要设置一下，你的DMA是用哪一个，即source master是谁。因为，此处的DMA的控制者有2个，可以简单理解为，有两个dma控制器，master1和master2，你具体选哪个master来控制你的DMA传输。关于master的选择，后面的代码分析中，会再次提及。

❺ 即source increment和destination increment，此处可翻译为，源地址递增和目标地址递增，

关于什么叫递增，为何要递增，可以从最开始提到的，DMA的常见应用情况中来解释，

因为常见的DMA传输，是从某个设备读取数据到某块内存区域，即DMA_FROM_DEVICE，以DMA模式，将数据“From从”你的设备中，读取到某块内存里面；或者将某块内存区域内数据，写入到设备里面，即DMA_TO_DEVICE。

而设备往往都是只有一个DATA寄存器（或是FIFO），此地址是固定不变的，

因此：

对于DMA_FROM_DEVICE，你传输完一个数据了，再传下一个数据的时候，此时你的Source源，还是你的设备的那个寄存器的地址，没有变化，而目标地址，往往要增加一个你的destination width，比如是一个word，32bit，4个字节，即地址要加4了，对应的就要将destination increment设置成1，表示，你的dma每次传输完，目标地址要增加的。

当然具体增加的大小，由你的设置的destination width的值决定。

相应地，如果是DMA_TO_DEVICE，那么就是DMA将数据从内存中写入到你的设备里面，每次传输完后，就是源source地址要增加，source increment要设置为1.

同理，如果是MEM_TO_MEM类型的，内存到内存，就是两者都要设置成1了，因为DMA传输完单个数据后，源地址和目标地址都要改变，要增加相应位宽的大小的。

❻ 即Protection，保护位，共3bit。

一般很少用到此域，我也没有完全理解，故不多解释，仅简述其义：

1. Bit0
   0是普通用户模式,1是特权模式

2. Bit1
   0是non-bufferable，1是bufferable

3. Bit2
   0是non-cacheable，1是cacheable

❼ 是否启用计数终止中断（Terminal count interrupt）。

关于此位，之前一直很迷惑，后面终于看懂了。

就是说，对于DMA的传输中的单个LLI来说，当前传输完成了，对应的寄存器中的transfer size域的值，也就是从设置的值，递减到0，也就是此计数递减到0，即结束了，即terminal count，而Terminal Count Interrupt，即传输完了，达到了terminal count时候，发送一个中断，告诉设备此次传输完成了。

如果此位被设置为0，那么传输完当前的LLI，就不会发送这个中断了，设置为1，就发送此中断。

❽ 此处之所以定义成union类型，就是方便，在设置好了之后，将此32位的值，直接写入对应的32位的寄存器中。

1.7. PL08X _lli _chan_lli pl08x_driver_data

```
/*
 * An LLI struct - see pl08x TRM
 * Note that next uses bit[0] as a bus bit,
 * start & end do not - their bus bit info
 * is in cctl
 */
struct _lli❶{
 dma_addr_t src;
 dma_addr_t dst;
 dma_addr_t next;❷
 union _cctl cctl;
};
struct _chan_lli❸ {
 dma_addr_t bus_list;
 void *va_list;
};
struct pl08x_driver_data {
 void __iomem *base;❹
 struct amba_device❺ *dmac;
 struct pl08x_platform_data *pd;
 /*
  * Single dma_pool for the LLIs
  */
 struct dma_pool *pool;❻
 int pool_ctr;❼
 struct work_struct dwork;❽
 wait_queue_head_t *waitq;
 int max_num_llis;
 /*
  * LLI details for each DMA channel
  */
 struct _chan_lli *chanllis;
 /* Wrappers for the struct dma_chan */
 struct pl08x_dma_chan❾ *chanwrap[MAX_DMA_CHANNELS];
 /* List of descriptors which can be freed */
 spinlock_t   lock;
};
/*
 * PL08X private data structures  ==== END
```

```
*/
```

❶ 这个就是DMA里面最核心的概念，LLI，Link List Item，包括了

1. 源地址
2. 目标地址
3. 下一个LLI的地址
   如果其为0/NULL/空，说明当前只需要传输一个信息，如果非空，传完当前的LLI，就会跳转到对应的地址，执行下一个LLI的传输
4. 对应的控制信息cctrl - channel control

这四个值，会分别写入到对应的四个寄存器。

这样配置好了之后，再去启用DMA，DMA就会按照你的要求，把数据从源地址传送到目的地址。

❷ next域的值，即下一个LLI的地址，其中的bit0，是bus bit，即指示当前用哪个bus。

现将datasheet中相关解释截图如下：

图 1.1. LLI寄存器的含义

Figure 3-15 shows the register bit assignments.

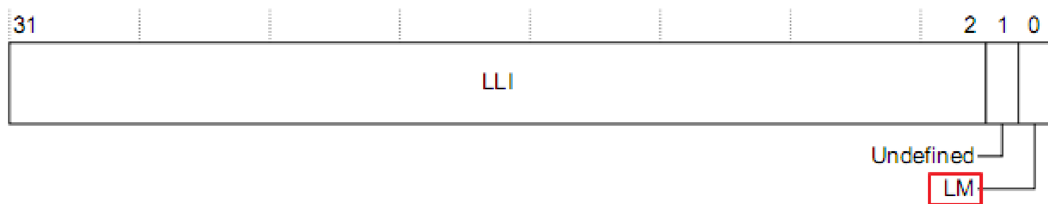

Figure 3-15 DMACCxLLI Register bit assignments

Table 3-18 lists the register bit assignments.

Table 3-18 DMACCxLLI Register bit assignments

Bits	Name	Function
[31:2]	LLI	Linked list item. Bits [31:2] of the address for the next LLI. Address bits [1:0] are 0.
[1]	-	Read undefined. Write as zero.
[0]	LM	AHB master select for loading the next LLI LM = 0 = AHB Master 1 LM = 1 = AHB Master 2.

对应的bit[0],LM位，就表示了，当前使用哪个AHB master，而真正的LLI的地址，是存放在bit[2-31]

❸ LLI结构体，存放LLI的dma地址（供DMA控制器访问的） bus_list，和虚拟地址（普通的地址，CPU可以访问的地址），va_list

❹ 记录对dma控制器硬件基地址ioremap之后的，驱动中可以直接访问的那个基地址

加上对应寄存器偏移量，就可以访问寄存器了

❺ 此pl08x的DMA控制也是一个amba设备

❻ dma内存池，用于存放之后，每次驱动的LLI

❼ 记录已使用内存池的数量，当达到足够大一个值的时候，做一次清理动作

❽ 一个工作队列，后面可以看到，挂了一个函数，作用是清理释放dma pool
❾ 将DMA 通道相关的信息，打包在这个变量里

1.8. PL08X macros

```
/*
 * Power Management.
 * PM support is not complete. Turn it off.❶
 */
#undef CONFIG_PM

#ifdef CONFIG_PM
#else
# define pl08x_do_suspend NULL
# define pl08x_do_resume  NULL
# define pl08x_suspend  NULL
# define pl08x_resume  NULL
#endif

#ifdef MODULE

# error "AMBA PL08X DMA CANNOT BE COMPILED AS A LOADABLE MODULE AT PRESENT"

/*
 a) Some devices might make use of DMA during boot
    (esp true for DMAENGINE implementation)
 b) Memory allocation will need much more attention
    before load/unload can be supported
 */
#endif

struct pl08x_driver_data pd;

/*
 * PL08X specific defines
 */
/* Minimum period between work queue runs */
#define PL08X_WQ_PERIODMIN 20
/* Size (bytes) of each buffer allocated for one transfer */
# define PL08X_LLI_TSFR_SIZE 0x2000❷
/* Maximimum times we call dma_pool_alloc on this pool without freeing */
# define PL08X_MAX_ALLOCS 0x40
#define MAX_NUM_TSFR_LLIS  (PL08X_LLI_TSFR_SIZE/sizeof(struct _lli))❸
#define PL08X_ALIGN  8
#define PL08X_ALLOC  0

/* Register offsets */❹
#define PL08X_OS_ISR  0x00
#define PL08X_OS_ISR_TC  0x04
#define PL08X_OS_ICLR_TC 0x08
#define PL08X_OS_ISR_ERR 0x0C
#define PL08X_OS_ICLR_ERR 0x10
#define PL08X_OS_CFG  0x30
#define PL08X_OS_CCFG  0x10
#define PL08X_OS_ENCHNS  0x1C
#define PL08X_OS_CHAN  0x20
```

```
#define PL08X_OS_CHAN_BASE 0x100

/* Channel registers */❺
#define PL08X_OS_CSRC  0x00
#define PL08X_OS_CDST  0x04
#define PL08X_OS_CLLI  0x08
#define PL08X_OS_CCTL  0x0C
/* register masks */
#define PL08X_MASK_CFG  0xFFFFFFF1
#define PL08X_MASK_EN  0x00000001
#define PL08X_MASK_CLLI  0x00000002
#define PL08X_MASK_TSFR_SIZE 0x00000FFF
#define PL08X_MASK_INTTC 0x00008000
#define PL08X_MASK_INTERR 0x00004000
#define PL08X_MASK_CCFG  0x00000000
#define PL08X_MASK_HALT  0x00040000
#define PL08X_MASK_ACTIVE 0x00020000
#define PL08X_MASK_CEN  0x00000001
#define PL08X_MASK_ENCHNS 0x000000FF
#define PL08X_WIDTH_8BIT 0x00
#define PL08X_WIDTH_16BIT 0x01
#define PL08X_WIDTH_32BIT 0x02

/*❻
 * Transferring less than this number of bytes as bytes
 * is faster than calculating the required LLIs....
 * (8 is the real minimum
 * >7 bytes must have a word alignable transfer somewhere)
 */
#define PL08X_BITESIZE  0x10
/*
 * Flow control bit masks❼
 */
#define PL08X_FCMASK_M2M_DMA 0x00000000
#define PL08X_FCMASK_M2P_DMA 0x00000800
#define PL08X_FCMASK_P2M_DMA 0x00001000
#define PL08X_FCMASK_P2P_DMA 0x00001800
#define PL08X_FCMASK_P2P_DST 0x00002000
#define PL08X_FCMASK_M2P_PER 0x00002800
#define PL08X_FCMASK_P2P_PER 0x00003000
#define PL08X_FCMASK_P2P_SRC 0x00003800
/* Max number of transfers which can be coded in the control register */
#define PL08X_MAX_TSFRS  0xFFF

#define PL08X_CODING_ERR 0xFFFFFFFF
```

❶ 暂时没有实现Linux中PM 的支持

❷ 对于其他使用此DMA 的驱动中，会去提交DMA请求，此处就是指一次DMA请求中，最大所允许的传输大小，此处是0x2000= 8192

❸ 后面代码中会看到，对于你的驱动提交的DMA请求，此DMA驱动内部会自动帮你转换成对应的一个个LLI，此数值就是限制一次DMA请求中，最大支持多少个LLI

❹ 一些全局的寄存器的偏移地址，这些值都是根据datasheet中定义出来的

❺ DMA控制器，有很多个通道channel，每个channel都对应有自己的一些寄存器，下面就是这些寄存器的地址偏移和位域值的含义

❻ 下面这几个宏定义，好像没用到

❼ 下面这个宏定义，好像也没用到，具体的流控制flow control的设置以及宏定义，在对应头文件中：

```
#define PL08X_CCFG_FCTL_MEM_TO_MEM  (0)
#define PL08X_CCFG_FCTL_MEM_TO_PERI  (1)
```

1.9. pl08x_decode_widthbits

```
static unsigned int pl08x_decode_widthbits❶(unsigned int coded)
{
 if (coded < 3)❷
  return 1 << coded;

 dev_err(&pd.dmac->dev, "%s - illegal width bits 0x%08x\n", __func__,
      coded);
 return PL08X_CODING_ERR;
}
```

❶ 根据设置的值，解码出实际位的宽度
❷ 根据datasheet中的解释：

图 1.2. Souce Transfer Width

[20:18]	SWidth	Source transfer width. Transfers wider than the AHB master bus width are illegal. The source and destination widths can be different from each other. The hardware automatically packs and unpacks the data when required.
[17:15]	DBSize	Destination burst size. Indicates the number of transfers that make up a destination burst transfer request. You must set this value to the burst size of the destination peripheral, or if the destination is memory, to the memory boundary size. The burst size is the amount of data that is transferred when the **DMACxBREQ** signal goes active in the destination peripheral. The burst size is not related to the AHB **HBURST** signal.

图 1.3. Souce or Destination Transfer Width

Table 3-21 lists the value of the SWidth or DWidth bits and their corresponding widths.

Table 3-21 Source or destination transfer width

Bit value of SWidth or DWidth	Source or destination width
0b000	Byte, 8-bit
0b001	Halfword, 16-bit
0b010	Word, 32-bit
0b011	Reserved
0b100	Reserved
0b101	Reserved
0b110	Reserved
0b111	Reserved

所以，目前只支持8.，16,32字节，位域的值分别是0，1, 2

所以此处判断小于3，才是有效的

然后1<<coded，得到的结果分别是1,2,4个字节。

1.10. pl08x_encode_width

```
static unsigned int pl08x_encode_width❶(unsigned int unencoded)
{
 unsigned int retval = unencoded >> 1;

 if (unencoded == (1 << retval))
  return retval;

 dev_err(&pd.dmac->dev, "%s - illegal width 0x%08x\n", __func__,
      unencoded);
 return PL08X_CODING_ERR;
}
```

❶ 上述函数第 1.9 节 “pl08x_decode_widthbits” 的逆过程

1.11. pl08x_choose_master_bus

```
/*
 * - prefers destination bus if both available
 * - if fixed address on one bus chooses other
```

```
*/
void pl08x_choose_master_bus❶(struct pl08x_bus_data *src_bus,
 struct pl08x_bus_data *dst_bus, struct pl08x_bus_data **mbus,
 struct pl08x_bus_data **sbus, union _cctl *cctl_parm)
{
 if (!cctl_parm->bits.di) {
  *mbus = src_bus;❷
  *sbus = dst_bus;
 } else if (!cctl_parm->bits.si) {
  *mbus = dst_bus;
  *sbus = src_bus;
 } else {
/* TODO: ??? */
  if (dst_bus->buswidth == 4) {
   *mbus = dst_bus;
   *sbus = src_bus;
  } else if (src_bus->buswidth == 4) {
   *mbus = src_bus;
   *sbus = dst_bus;
  } else if (dst_bus->buswidth == 2) {
   *mbus = dst_bus;
   *sbus = src_bus;
  } else if (src_bus->buswidth == 2) {
   *mbus = src_bus;
   *sbus = dst_bus;
  } else {
   /* src_bus->buswidth == 1 */
   *mbus = dst_bus;
   *sbus = src_bus;
  }
 }
}
```

❶ 根据传入的是源自增SI还是DI目的地址自增，决定谁是master bus

❷ 如果是 非DI，不是目标地址自增，那么就用src作为master bus 以实际常用的情况举例，比如要从某个buffer传输数据到nand flash的DATA寄存器中，那么就是DI为0，而作为主方向的buffer，就是master bus，从master的buffer传输到slave 的nand flash的DATA寄存器

1.12. pl08x_fill_lli_for_desc

```
int pl08x_fill_lli_for_desc(struct pl08x_txd *local_txd, int num_llis, int len,
    union _cctl *cctl, int *remainder)
{
 struct _lli *llis_va = (struct _lli *)(local_txd->llis_va);
 struct _lli *llis_bus = (struct _lli *)(local_txd->llis_bus);
 llis_va[num_llis].cctl.val = cctl->val;❶
 llis_va[num_llis].src  = local_txd->srcbus.addr;
 llis_va[num_llis].dst  = local_txd->dstbus.addr;
 /*
  * The bus bit is added to the next lli's address
  */
 llis_va[num_llis].next =
  (dma_addr_t)((unsigned int)&(llis_bus[num_llis + 1]❷)
    + pd.pd->bus_bit_lli❸);
 if (cctl->bits.si)
```

```
 local_txd->srcbus.addr += len;❹
if (cctl->bits.di)
 local_txd->dstbus.addr += len;

*remainder -= len;

return num_llis + 1;
}
```

❶ 填充对应的contrl，source, destination 三个寄存器对应的值
❷ llis_bus[num_llis + 1]比较好理解，就是当前的LLI的next的值，应该赋值为下一个LLI 的地址
❸ 需要特殊说明的是bus_bit_lli，此变量，意思为你当前使用DMA的哪个Master。

因为pl080内部有两个Master主控器，你在使用DMA的时候，要制定你当前是使用哪一个DMA，详情参考datasheet：

图 1.4. LLI寄存器的含义

Table 3-18 DMACCxLLI Register bit assignments

Bits	Name	Function
[31:2]	LLI	Linked list item. Bits [31:2] of the address for the next LLI. Address bits [1:0] are 0.
[1]	-	Read undefined. Write as zero.
[0]	LM	AHB master select for loading the next LLI LM = 0 = AHB Master 1 LM = 1 = AHB Master 2.

其中32位的LLI的地址，由于是4对齐的，所以bit0和bit1肯定是0，正好利用第0位指示是master1还是master2。

此变量不是此处pl08x DMA驱动赋值的，而是在你最开始去注册amba设备的时候赋值的。

❹ 如果是SI，那么源地址在一次LLI的DMA传输后，源地址就应该更新，为再加上对应的每次传输的长度

1.13. pl08x_pre_boundary

```
/*
 * Return number of bytes to fill to boundary, or len
 */
static int pl08x_pre_boundary(int addr, int len)
{
 int boundary;

 if (len <= 0)
  dev_err(&pd.dmac->dev, "%s - zero length\n", __func__);
 boundary = ((addr >> PL08X_BOUNDARY_SHIFT❶) + 1) << PL08X_BOUNDARY_SHIFT;

 if (boundary < addr + len)
  return boundary - addr;
 else
  return len;
}
```

❶ Pl08x.h中定义的：

```
#define PL08X_BOUNDARY_SHIFT (10) /* 1KB 0x400 */
#define PL08X_BOUNDARY_SIZE  (1 << PL08X_BOUNDARY_SHIFT)
```

此函数，目的是为了限制每次DMA传输的字节数，要保证在PL08X_BOUNDARY_SIZE即1KB范围内。

之所以做此限制，是因为datasheet中写了："Bursts do not cross the 1KB address boundary"

突发传输，不能跨界超过1KB 的范围。

所以，如果你上层程序调用此pl08x驱动，希望每次传输2KB，那么此驱动会自动帮你限制为每次最多1KB，然后自动帮你拆分出对应的多个LLI。

第 2 章 ARM的PL08X的代码的详细解析第二部分

2.1. fill_LLIS_for_desc

```
/*
 * Note that we assume we never have to change the burst sizes
 * Return 0 for error
 */
int fill_LLIS_for_desc(struct pl08x_txd *local_txd, int pl08x_chan_num)
{
 struct pl08x_clientdev_data *client = local_txd->pcd;
 struct pl08x_bus_data *mbus, *sbus;
 int remainder;
 int num_llis = 0;
 union _cctl cctl_parm;
 int max_bytes_per_lli;
 int total_bytes = 0;
 struct _lli *llis_va;
 struct _lli *llis_bus;

 if (!local_txd) {
  dev_err(&pd.dmac->dev, "%s - no descriptor\n", __func__);
  return 0;
 }

 /*
  * Get some LLIs
  *  This alloc can wait if the pool is used up so we need to cleanup
  */
 local_txd->llis_va = dma_pool_alloc(pd.pool, GFP_KERNEL,
     &local_txd->llis_bus);
 if (!local_txd->llis_va) {
  dev_err(&pd.dmac->dev, "%s - no llis\n", __func__);
  return 0;
 }

 pd.pool_ctr++;

 /*
  * Initialize bus values for this transfer
  * from the passed optimal values
  */
 if (!client) {
  dev_err(&pd.dmac->dev, "%s - no client\n", __func__);
  return 0;
 }

 cctl_parm.val = client->cctl_opt;
 local_txd->srcbus.maxwidth =❶
  pl08x_decode_widthbits(cctl_parm.bits.swidth);

 if (local_txd->srcbus.maxwidth == PL08X_CODING_ERR) {
```

```
 dev_err(&pd.dmac->dev,
  "%s - local_txd->srcbus.maxwidth codeing error cctl_parm.bits.swidth %d\n",
   __func__, cctl_parm.bits.swidth);
 return 0;
}

local_txd->srcbus.buswidth = local_txd->srcbus.maxwidth;
local_txd->dstbus.maxwidth =
 pl08x_decode_widthbits(cctl_parm.bits.dwidth);

if (local_txd->dstbus.maxwidth == PL08X_CODING_ERR) {
 dev_err(&pd.dmac->dev,
  "%s - local_txd->dstbus.maxwidth coding error - cctl_parm.bits.dwidth %d\n",
   __func__, cctl_parm.bits.dwidth);
 return 0;
}

local_txd->dstbus.buswidth = local_txd->dstbus.maxwidth;

/*
 *  Note bytes transferred == tsize * MIN(buswidths), not max(buswidths)
 */
max_bytes_per_lli = ❷min(local_txd->srcbus.maxwidth,
    local_txd->dstbus.maxwidth) *
     cctl_parm.bits.tsize;

remainder = local_txd->len;
/*
 * Choose bus to align to
 * - prefers destination bus if both available
 * - if fixed address on one bus chooses other
 */
pl08x_choose_master_bus(&local_txd->srcbus,
 &local_txd->dstbus, &mbus, &sbus, &cctl_parm);

if (local_txd->len < mbus->buswidth)❸ {
 /*
  * Less than a bus width available
  * - send as single bytes
  */
 while (remainder) {
  cctl_parm.bits.swidth = pl08x_encode_width(1);
  cctl_parm.bits.dwidth = pl08x_encode_width(1);
  cctl_parm.bits.tsize = 1;
  num_llis =
   pl08x_fill_lli_for_desc(local_txd, num_llis, 1,
    &cctl_parm, &remainder);
  total_bytes++;
 }
} else {
 /*
  *  Make one byte LLIs until master bus is aligned
  *  - slave will then be aligned also
  */
 while ((mbus->addr) % (mbus->buswidth)) {
  cctl_parm.bits.swidth = pl08x_encode_width(1);
  cctl_parm.bits.dwidth = pl08x_encode_width(1);
  cctl_parm.bits.tsize = 1;
```

```
 num_llis = pl08x_fill_lli_for_desc
  (local_txd, num_llis, 1, &cctl_parm,
   &remainder);
 total_bytes++;
}
/*
 *  Master now aligned
 * - if slave is not then we must set its width down
 */
if (sbus->addr % sbus->buswidth)
 sbus->buswidth = 1;

/*
 * Make largest possible LLIs until less than one bus width left
 */
while (remainder > (mbus->buswidth - 1)) {
 int lli_len, target_len;
 int tsize;
 int odd_bytes;
 /*
  * If enough left try to send max possible,
  * otherwise try to send the remainder
  */
 target_len = remainder;
 if (remainder > max_bytes_per_lli)
  target_len = max_bytes_per_lli;
 /*
  *  Set bus lengths for incrementing busses
  *  to number of bytes which fill
  *  to next memory boundary
  */
 if (cctl_parm.bits.si)
  local_txd->srcbus.fill_bytes =
   pl08x_pre_boundary(
    local_txd->srcbus.addr,
    remainder);❹
 else
  local_txd->srcbus.fill_bytes =
   max_bytes_per_lli;
 if (cctl_parm.bits.di)
  local_txd->dstbus.fill_bytes =
   pl08x_pre_boundary(
    local_txd->dstbus.addr,
    remainder);
 else
  local_txd->dstbus.fill_bytes =
    max_bytes_per_lli;
 /*
  *  Find the nearest
  */
 lli_len = min(local_txd->srcbus.fill_bytes,
    local_txd->dstbus.fill_bytes);

 if (lli_len <= 0) {
  dev_err(&pd.dmac->dev,
   "%s - lli_len is %d, <= 0\n",
    __func__, lli_len);
  return 0;
```

```
 }

 if (lli_len == target_len) {
  /*
   * Can send what we wanted
   */
  /*
   *  Maintain alignment
   */
  lli_len = (lli_len/mbus->buswidth) *
     mbus->buswidth;
  odd_bytes = 0;
 } else {
  /*
   * So now we know how many bytes to transfer
   * to get to the nearest boundary
   * The next lli will past the boundary
   * - however we may be working to a boundary
   *   on the slave bus
   *   We need to ensure the master stays aligned
   */
  odd_bytes = lli_len % mbus->buswidth;
  /*
   * - and that we are working in multiples
   *   of the bus widths
   */
  lli_len -= odd_bytes;

 }
 if (lli_len) {
  /*
   * Check against minimum bus alignment
   */
  target_len = lli_len;
  tsize = lli_len/min(mbus->buswidth,
     sbus->buswidth);
  lli_len = tsize * min(mbus->buswidth,
     sbus->buswidth);

  if (target_len != lli_len) {
   dev_err(&pd.dmac->dev,
   "%s - can't send what we want. Desired %d, sent %d in transfer of %d\n",
   __func__, target_len, lli_len, local_txd->len);
   return 0;
  }

  cctl_parm.bits.swidth = pl08x_encode_width
   (local_txd->srcbus.buswidth);
  cctl_parm.bits.dwidth = pl08x_encode_width
   (local_txd->dstbus.buswidth);
  if ((cctl_parm.bits.swidth == PL08X_CODING_ERR) ||
   (cctl_parm.bits.dwidth == PL08X_CODING_ERR)) {
   dev_err(&pd.dmac->dev,
   "%s - cctl_parm.bits.swidth or dwidth coding error - local_txd->dstbus.buswidth %d,
local_txd->srcbus.buswidth %d\n",
   __func__,
   local_txd->dstbus.buswidth,
   local_txd->srcbus.buswidth
```

```
    );
    return 0;
   }
   cctl_parm.bits.tsize = tsize;
   num_llis = pl08x_fill_lli_for_desc(local_txd,
     num_llis, lli_len, &cctl_parm,
     &remainder);
   total_bytes += lli_len;
  }
  if (odd_bytes) {
   /*
    * Creep past the boundary,
    * maintaining master alignment
    */
   int j;
   for (j = 0; (j < mbus->buswidth)
     && (remainder); j++) {
    cctl_parm.bits.swidth =
     pl08x_encode_width(1);
    cctl_parm.bits.dwidth =
     pl08x_encode_width(1);

    cctl_parm.bits.tsize = 1;
    num_llis =
     pl08x_fill_lli_for_desc(
      local_txd, num_llis, 1,
      &cctl_parm, &remainder);
    total_bytes++;
   }
  }
 }

 /*
  * Send any odd bytes
  */
 if (remainder < 0) {
  dev_err(&pd.dmac->dev, "%s - -ve remainder 0x%08x\n",
    __func__, remainder);
  return 0;
 }

 while (remainder) {
  cctl_parm.bits.swidth = pl08x_encode_width(1);
  cctl_parm.bits.dwidth = pl08x_encode_width(1);
  cctl_parm.bits.tsize = 1;
  num_llis = pl08x_fill_lli_for_desc(local_txd, num_llis,
    1, &cctl_parm, &remainder);
  total_bytes++;
 }
}
if (total_bytes != local_txd->len) {
 dev_err(&pd.dmac->dev,
  "%s - only transferred 0x%08x from size 0x%08x\n",
   __func__, total_bytes, local_txd->len);
 return 0;
}

if (num_llis >= MAX_NUM_TSFR_LLIS) {❺
```

```
 dev_err(&pd.dmac->dev,
  "%s - need to increase MAX_NUM_TSFR_LLIS from 0x%08x\n",
   __func__, MAX_NUM_TSFR_LLIS);
 return 0;
}
/*
 * Decide whether this is a loop or a terminated transfer
 */
llis_va = ((struct _lli *)local_txd->llis_va);
llis_bus = ((struct _lli *)local_txd->llis_bus);

if (client->circular_buffer) {❻
 llis_va[num_llis - 1].next =
  (dma_addr_t)((unsigned int)&(llis_bus[0]) +
     pd.pd->bus_bit_lli);
} else {
 /*
  * Final LLI terminates
  */

 llis_va[num_llis - 1].next = 0;❼
 /*
  * Final LLI interrupts
  */
 llis_va[num_llis - 1].cctl.bits.intr = PL08X_CCTL_INTR_YES;❽
}

/* Now store the channel register values */
local_txd->csrc = llis_va[0].src;
local_txd->cdst = llis_va[0].dst;
if (num_llis > 1)
 local_txd->clli = llis_va[0].next;
else
 local_txd->clli = 0;

local_txd->cctl = llis_va[0].cctl.val;
local_txd->ccfg = client->config_base;

/*
 * TODO: Change to use /proc data
 */
if (pd.max_num_llis < num_llis)
 pd.max_num_llis = num_llis;

return num_llis;
}
```

❶ 从传入的配置中，解码出bus宽度，单位字节

❷ 从源总线和目的总线选出一个最小带宽，然后乘与一个传输的个数，得到单个LLI的最大允许的字节数

❸ 要传输的数据，比带宽还小，那简单地分成几个LLI就搞定了

❹ 检查数据有没有超过允许的范围，如果超过了，就用PL08X_BOUNDARY_SIZE=0x400=1KB

❺ 如果你一次要求传输数据太多，然后拆分成了太多个LLI，那么这里会告诉你超过限制了

❻ 如果是循环缓存circular buffer，那么就告诉DMA传完最后一个LLI的时候，继续从最这个LLI的链表的最开始一个传，这样就周而复始地传输了，一般适用于音频流数据

❼ 最后一个LLI的next LLI指针的值，一定要设置为NULL，表示DMA传输完这个LLI之后，就结束了

❸ 最后DMA传完所有的数据了，肯定要发生中断，然后此出pl08x的irq函数会被调用，然后会再接着调用你的驱动做DMA请求时候挂载的callback函数

2.2. pl08x_set_cregs

```
/*
 * Set the initial DMA register values i.e. those for the first LLI
 * The next lli pointer and the configuration interrupt bit have
 * been set when the LLIs were constructed
 */
void pl08x_set_cregs(struct pl08x_txd *entry, int pl08x_chan_num)
{
 unsigned int reg;
 unsigned int chan_base❶ = (unsigned int)pd.base
     + PL08X_OS_CHAN_BASE;
 chan_base += pl08x_chan_num * PL08X_OS_CHAN;

 /* Wait for channel inactive */❷
 reg = readl(chan_base + PL08X_OS_CCFG);
 while (reg & PL08X_MASK_ACTIVE)
  reg = readl(chan_base + PL08X_OS_CCFG);

 writel(entry->csrc, chan_base + PL08X_OS_CSRC);
 writel(entry->cdst, chan_base + PL08X_OS_CDST);
 writel(entry->clli, chan_base + PL08X_OS_CLLI);
 writel(entry->cctl, chan_base + PL08X_OS_CCTL);
 writel(entry->ccfg, chan_base + PL08X_OS_CCFG);
 mb();
}
```

❶ 找到当前使用的channel的基地址

❷ 如果之前正有人用此channel的DMA，就等待直到不在用，即inactive为止

注意，此处没有直接去Halt或者disable对应的channel，因为有可能之前的某个DMA正在传输过程中，所以应该等待其完成，当然，大多数情况都是已经是inactive了

2.3. pl08x_memrelease

```
/*
 *  Might take too long to do all at once
 *  if so - free some then hand it off
 */
void pl08x_memrelease❶(void)
{
 struct pl08x_txd *local_txd, *next;
 unsigned long flags;
 int chan;

 for (chan = 0; chan < MAX_DMA_CHANNELS; chan++) {

  list_for_each_entry_safe(local_txd, next, &pd.chanwrap[chan]->release_list, txd_list)
  {
   spin_lock_irqsave(&pd.lock, flags);
   if (!local_txd)
```

```
   dev_err(&pd.dmac->dev,
    "%s - no descriptor\n", __func__);

  if (local_txd->tx.callback) {
   ((struct pl08x_callback_param *)
    local_txd->tx.callback_param)->act =
     PL08X_RELEASE;
   local_txd->tx.callback(
    local_txd->tx.callback_param);
  }

  list_del(&local_txd->txd_list);
  dma_pool_free(pd.pool, local_txd->llis_va,
   local_txd->llis_bus);

  /*
   * p/m data not mapped - uses coherent or io mem
   */
  if (!local_txd->slave) {
   dma_unmap_single(dmac.dev,
    local_txd->srcbus.addr, local_txd->len,
     local_txd->srcdir);

   dma_unmap_single(dmac.dev,
    local_txd->dstbus.addr, local_txd->len,
     local_txd->dstdir);
  }
  kfree(local_txd);
  spin_unlock_irqrestore(&pd.lock, flags);
 }
}
pd.pool_ctr = 0;
}
```

❶ 释放DMA内存

2.4. pl08x_disable_dmac_chan

```
/*
 * Overall DMAC remains enabled always.
 *
 * Disabling individual channels could lose data.
 *
 * Disable the peripheral DMA after disabling the DMAC
 * in order to allow the DMAC FIFO to drain, and
 * hence allow the channel to show inactive
 *
 */
void pl08x_disable_dmac_chan(unsigned int ci)
{
 unsigned int reg;
 unsigned int chan_base = (unsigned int)pd.base
     + PL08X_OS_CHAN_BASE;

 chan_base += ci * PL08X_OS_CHAN;
```

```
 /*
  * Ignore subsequent requests
  */
 reg = readl(chan_base + PL08X_OS_CCFG);
 reg |= PL08X_MASK_HALT;
 writel(reg, chan_base + PL08X_OS_CCFG);
   ❶
 /* Wait for channel inactive */
 reg = readl(chan_base + PL08X_OS_CCFG);
 while (reg & PL08X_MASK_ACTIVE)
  reg = readl(chan_base + PL08X_OS_CCFG);

 reg = readl(chan_base + PL08X_OS_CCFG);
 reg &= ~PL08X_MASK_CEN;

 writel(reg, chan_base + PL08X_OS_CCFG);
 mb();

 return;
}
```

❶ 此处对于disable一个DMA的channel的做法，是根据datasheet中的描述：

"

Disabling a DMA channel without losing data in the FIFO

To disable a DMA channel without losing data in the FIFO:

1. Set the Halt bit in the relevant channel Configuration Register. See Channel Configuration Registers on page 3-27. This causes any subsequent DMA requests to be ignored

2. Poll the Active bit in the relevant channel Configuration Register until it reaches 0. This bit indicates whether there is any data in the channel that has to be transferred

3. Clear the Channel Enable bit in the relevant channel Configuration Register

"

即先设置Halt位，然后一直轮询检测对应channel，直到inactive，然后再清空对应的Channel Enable位，如果直接不做任何检测判断，上来就去清空对应的Enable位，那么就会“losing data in the FIFO”，丢失了FIFO中的数据，是不太安全的

2.5. pl08x_enable_dmac_chan

```
/*
 * Enable the DMA channel
 * ASSUMES All other configuration bits have been set
 *         as desired before this code is called
 */
void pl08x_enable_dmac_chan(unsigned int cnum)
{
 void __iomem *cbase = pd.base + PL08X_OS_CHAN_BASE +
    (cnum * PL08X_OS_CHAN);

 unsigned int r = 0;
```

```
/*
 * Do not access config register until channel shows as disabled
 */
while ((readl(pd.base + PL08X_OS_ENCHNS)❶ & (1 << cnum))
  & PL08X_MASK_ENCHNS);

/*
 * Do not access config register until channel shows as inactive
 */
r = readl(cbase + PL08X_OS_CCFG);
while ((r & PL08X_MASK_ACTIVE)❷ || (r & PL08X_MASK_CEN))
 r = readl(cbase + PL08X_OS_CCFG);

writel(r | PL08X_MASK_CEN, cbase + PL08X_OS_CCFG);
mb();
}
```

❶ 等到对应channel是disable状态了，再接下来去enable它，会更安全
❷ 等到对应channel是inactive状态了，再接下来去enable它，会更安全

2.6. pl08x_dma_busy

```
static int pl08x_dma_busy❶(dmach_t ci)
{
 unsigned int reg;
 unsigned int chan_base = (unsigned int)pd.base
     + PL08X_OS_CHAN_BASE;

 chan_base += ci * PL08X_OS_CHAN;

 /*
  * Check channel is inactive
  */
 reg = readl(chan_base + PL08X_OS_CCFG);

 return reg & PL08X_MASK_ACTIVE;
}
```

❶ 检测对应的位，得到是否是active的状态，对应着是否是busy

2.7. pl08x_wqfunc

```
/*
 * Function for the shared work queue
 * - scheduled by the interrupt handler when sufficient
 *   list entries need releasing
 */
void pl08x_wqfunc(struct work_struct *work)
{
 if (pd.pd)
  pl08x_memrelease();
}
```

2.8. pl08x_amba_driver

```
static struct amba_id pl08x_ids[] = {
 /* PL080 */
 {
  .id = 0x00041080❶,
  .mask = 0x000fffff,
 },
 /* PL081 */
 {
  .id = 0x00041081❷,
  .mask = 0x000fffff,
 },
 { 0, 0 },
};

#define DRIVER_NAME "pl08xdmac"

static int pl08x_probe(struct amba_device *amba_dev, void *id);

static struct amba_driver pl08x_amba_driver = {
 .drv.name = "pl08xdmaapi",
 .id_table = pl08x_ids,
 .probe  = pl08x_probe,
};
```

❶ PL080对应的设备ID是0x00041080详情参看Datasheet的"3.4.16 Peripheral Identification Registers 0-3"和"3.4.17 PrimeCell Identification Registers 0-3"

❷ PL081对应的设备ID是0x00041081详情参看Datasheet的"3.4.16 Peripheral Identification Registers 0-3"和"3.4.17 PrimeCell Identification Registers 0-3"

2.9. pl08x_make_LLIs

```
/*
 * This single pool easier to manage than one pool per channel
 */
int pl08x_make_LLIs(void)
{
 int ret = 0;

 /*
  * Make a pool of LLI buffers
  */
 pd.pool = dma_pool_create(pl08x_amba_driver.drv.name, &pd.dmac->dev,
   PL08X_LLI_TSFR_SIZE, PL08X_ALIGN, PL08X_ALLOC);
 if (!pd.pool) {
  ret = -ENOMEM;
  kfree(pd.chanllis);
 }
 pd.pool_ctr = 0;
 return ret;
}
```

注意

需要提到一点的是，如果你的驱动在初始化的时候，设置了下图：

图 2.1. 配置寄存器的含义

Table 3-23 DMACCxConfiguration Register bit assignments

Bits	Name	Type	Function
[31:19]	-	-	Read undefined. Write as zero.
[18]	H	R/W	Halt: 0 = enable DMA requests 1 = ignore extra source DMA requests. The contents of the channels FIFO are drained. You can use this value with the Active and Channel Enable bits to cleanly disable a DMA channel.
[17]	A	RO	Active: 0 = there is no data in the FIFO of the channel 1 = the FIFO of the channel has data. You can use this value with the Halt and Channel Enable bits to cleanly disable a DMA channel.
[16]	L	R/W	Lock. When set, this bit enables locked transfers. For details of how lock control works, see *Lock control* on page 2-15.
[15]	ITC	R/W	Terminal count interrupt mask. When cleared, this bit masks out the terminal count interrupt of the relevant channel.

中的config寄存器中的ITC位，即终止计数中断，那么就会在每一个LLI传完之后就调用下面的这个pl08x_irq中断函数，如果没有设置ITC，而只是对最有一个LLI的ITC 设置了，那么就是所有的传输完成了，才会调用下面的pl08x_irq中断函数了

2.10. pl08x_irq

```
/*
 * issue pending() will start the next transfer
 * - it only need be loaded here
 * CAUTION the work queue function may run during the handler
 * CAUTION function callbacks may have interesting side effects
 */
static irqreturn_t pl08x_irq(int irq, void *dev)
{
 u32 mask = 0;
 u32 reg;
 unsigned long flags;
 int c;
 reg = readl(pd.base + PL08X_OS_ISR_ERR);
 mb();
 if (reg) {
  /*
   * An error interrupt (on one or more channels)
   */
  dev_err(&pd.dmac->dev,
   "%s - Error interrupt, register value 0x%08x\n",
    __func__, reg);
  /*
   * Simply clear ALL PL08X error interrupts,❶
```

```
  * regardless of channel and cause
  */
 writel(0x000000FF, pd.base + PL08X_OS_ICLR_ERR);
}
reg = readl(pd.base + PL08X_OS_ISR);
mb();
for (c = 0; c < MAX_DMA_CHANNELS; c++) {
 if ((1 << c) & reg) {
  struct pl08x_txd *entry = NULL;
  struct pl08x_txd *next;

  spin_lock_irqsave(&pd.chanwrap[c]->lock, flags);

  if (pd.chanwrap[c]->at) {
   /*
    * Update last completed
    */
   pd.chanwrap[c]->lc =
    (pd.chanwrap[c]->at->tx.cookie);
   /*
    * Callback peripheral driver for p/m
    * to signal completion
    */
   if (pd.chanwrap[c]->at->tx.callback) {
    /*
     * Pass channel number
     */
    ((struct pl08x_callback_param *)
     pd.chanwrap[c]->at->tx.callback_param)->act = c;
    pd.chanwrap[c]->at->tx.callback❷(
     pd.chanwrap[c]->at->tx.callback_param);
   }
   /*
    * Device callbacks should NOT clear
    * the current transaction on the channel
    */
   if (!pd.chanwrap[c]->at)
    BUG();

   /*
    * Free the descriptor if it's not for a device
    * using a circular buffer
    */
   if (!(pd.chanwrap[c]->at->pcd->circular_buffer)) {
    list_add_tail(&pd.chanwrap[c]->at->txd_list,
     &pd.chanwrap[c]->release_list);
    pd.chanwrap[c]->at = NULL;
    if (pd.pool_ctr > PL08X_MAX_ALLOCS) {
     schedule_work(&pd.dwork);
    }
   }
   /*
    * else descriptor for circular
    * buffers only freed when
    * client has disabled dma
    */
  }
  /*
```

```
  * If descriptor queued, set it up
  */
  if (!list_empty(&pd.chanwrap[c]->desc_list)) {
   list_for_each_entry_safe(entry,
   next, &pd.chanwrap[c]->desc_list, txd_list) {
    list_del_init(&entry->txd_list);
    pd.chanwrap[c]->at = entry;
    break;
   }
  }

  spin_unlock_irqrestore(&pd.chanwrap[c]->lock, flags);

  mask |= (1 << c);
 }
}
/*
 * Clear only the terminal interrupts on channels we processed
 */
writel(mask, pd.base + PL08X_OS_ICLR_TC);
mb();

return mask ? IRQ_HANDLED : IRQ_NONE;
}
```

❶ 如果发现有任何错误，就退出

❷ 实际上此处很多种错误，而目前的此版本的驱动，暂时没加入更多的判断，仅仅是报错而已此处，真正去调用你之前挂载的callback函数，在callback函数里面，常见要做的事情就是清除你设备的DMA enable位，然后调用complete去使得completion变量完成，使得你驱动得以继续执行

2.11. pl08x_ensure_on

```
static void pl08x_ensure_on❶(void){
 unsigned int reg;

 reg = readl(pd.base + PL08X_OS_CFG);
 reg &= PL08X_MASK_CFG;
 reg |= PL08X_MASK_EN;
 mb();
 writel(reg, pd.base + PL08X_OS_CFG);
 mb();
}
```

❶ 全局性地Enable PL08x

2.12. pl08x_dma_enumerate_channels

```
/*
 * Initialise the DMAC channels.
 * Make a local wrapper to hold required data
 */
```

```
static int pl08x_dma_enumerate_channels❶(void)
{
 int dma_chan;
 struct pl08x_dma_chan *local_chan;

 dma_cap_set(DMA_MEMCPY, dmac.cap_mask);
 dma_cap_set(DMA_SLAVE, dmac.cap_mask);
 if (dmac.chancnt)
  dev_err(&pd.dmac->dev, "%s - chancnt already set\n", __func__);
 INIT_LIST_HEAD(&dmac.channels);

 for (dma_chan = 0; dma_chan < MAX_DMA_CHANNELS; dma_chan++) {

  local_chan = kzalloc(sizeof(struct pl08x_dma_chan),
     GFP_KERNEL);
  if (!local_chan) {
   dev_err(&pd.dmac->dev,
    "%s - no memory for channel\n", __func__);
   return dmac.chancnt;
  }

  /*
   * Save the DMAC channel number
   * to indicate which registers to access
   */
  local_chan->chan_id = dma_chan;
  local_chan->chan.device = &dmac;
  atomic_set(&local_chan->last_issued, 0);
  local_chan->lc = atomic_read(&local_chan->last_issued);
  if (pd.pd->reserved_for_slave)
   local_chan->slave_only = pd.pd->reserved_for_slave(dma_chan);
  else
   dev_err(&pd.dmac->dev,
    "%s - can't establish channels reserved for slaves\n",
     __func__);

  spin_lock_init(&local_chan->lock);
  INIT_LIST_HEAD(&local_chan->desc_list);
  INIT_LIST_HEAD(&local_chan->release_list);

  list_add_tail(&local_chan->chan.device_node, &dmac.channels);

  pd.chanwrap[dmac.chancnt++] = local_chan;
 }

 return dmac.chancnt;
}
```

❶ 为目前支持的多个channel进行必要的初始化，包括申请空间，设置初始值

2.13. pl08x_probe

```
static int pl08x_probe❶(struct amba_device *amba_dev, void *id)
{
 int ret = 0;
```

```
ret = pl08x_make_LLIs();❷
if (ret)
 return -ENOMEM;

ret = amba_request_regions❸(amba_dev, NULL);
if (ret)
 return ret;

pd.max_num_llis = 0;

/*
 * We have our own work queue for cleaning memory
 */
INIT_WORK(&pd.dwork, pl08x_wqfunc);
pd.waitq = kmalloc(sizeof(wait_queue_head_t), GFP_KERNEL);
init_waitqueue_head(pd.waitq);

spin_lock_init(&pd.lock);

pd.base = ioremap(amba_dev->res.start, SZ_4K);
if (!pd.base) {
 ret = -ENOMEM;
 goto out1;
}
/*
 * Attach the interrupt handler
 */
writel(0x000000FF, pd.base + PL08X_OS_ICLR_ERR);
writel(0x000000FF, pd.base + PL08X_OS_ICLR_TC);
mb();

ret = request_irq(amba_dev->irq[0], pl08x_irq, IRQF_DISABLED,
  DRIVER_NAME, &amba_dev->dev);
if (ret) {
 dev_err(&pd.dmac->dev, "%s - Failed to attach interrupt %d\n",
   __func__, amba_dev->irq[0]);
 goto out2;
}
/*
 * Data exchange
 */
pd.dmac = amba_dev;
amba_set_drvdata(amba_dev, &pd);
pd.pd = (struct pl08x_platform_data *)(amba_dev->dev.platform_data);
/*
 * Negotiate for channels with the platform
 * and its DMA requirements
 */
dmac.dev = &amba_dev->dev;

ret = pl08x_dma_enumerate_channels();

if (!ret) {
 dev_warn(&pd.dmac->dev,
  "%s - failed to enumerate channels - %d\n",
   __func__, ret);
 goto out2;
```

```
} else {
 ret = dma_async_device_register❹(&dmac);
 if (ret) {
  dev_warn(&pd.dmac->dev,
   "%s - failed to register as an async device - %d\n",
    __func__, ret);
  goto out2;
 }
}
pl08x_ensure_on();❺
dev_info(&pd.dmac->dev,
 "%s - ARM(R) PL08X DMA driver found\n",
  __func__);

goto out;
out2:
 iounmap(pd.base);
out1:
 amba_release_regions(amba_dev);
out:
 return ret;
}
```

❶ 经典的probe函数，对驱动进行必要的软件和硬件的初始化

❷ 为后面要用到的LLI申请空间

❸ 获得自己资源，其是你开始进行amba设备注册时候的，已经初始化好的对应的资源，包括基地址，中断号等

❹ 将自己这个pl08x的DMA驱动，注册到DMA Engine 框架中

别处看到某人评论说是这个DMA Engine驱动框架属于老的了，有更好的新的，不了解，有空去看看

❺ 硬件上确保启动DMA了

2.14. pl08x_init

```
static int __init pl08x_init(void)
{
 int retval;
 retval = amba_driver_register(&pl08x_amba_driver);
 if (retval)
  printk(KERN_WARNING
   "PL08X::pl08x_init() - failed to register as an amba device - %d\n",
    retval);
 return retval;
}
device_initcall(pl08x_init);
```

第 3 章 ARM的PL08X的代码的详细解析第三部分

3.1. 简述PL08X的DMA驱动的基本流程

接下来将要解释的这些函数，都是挂靠在DMA Engine架构下面的函数。

此 pl08x的DMA驱动也要是实现对应的函数，以便其他程序调用这些接口使用pl08x。

下面简单介绍一下通用的调用此pl08x的DMA驱动的大概流程：

1. 得到当前可以用于DMA传输的数据buffer
   - 如果已经有了普的数据buffer，那么一般用dma_map_single去将普通的CPU访问的buffer映射成DMA可以访问的buffer
   - 如果没有现存buffer，那么一般用dma_alloc_writecombine，自己申请一个，申请的此buffer，是combine，绑定一起的，即不论是DMA还是CPU，都可以访问的
2. 根据自己的需求，设置好传输的client的一堆参数后，然后调用
   dma_async_client_register(txclient);

   dma_async_client_chan_request(txclient);

   去注册申请对应的client。
3. 设置好scatter/gather list的信息后，调用
   device_prep_slave_sg
   去寻找并获得和自己匹配的那个描述符desc
4. 然后设置对应的callback函数和对应参数
   desc->callback = as353x_nand_dma_complete_callback;

   desc->callback_param = &info->callback_param;

   注意

   callback函数里面，会调用complete(&info->done);去完成对应的变量

   而你的callback函数，会在DMA完全传输完毕后，DMA驱动中的irq中会被调用。
5. 都准备好了后，再调用
   desc->tx_submit(desc);
   去提交你的DMA请求
6. 一切完毕后，调用
   info->txchan->device->device_issue_pending(info->txchan);
   真正的开始DMA的数据传输。
7. 之后，你就可以调用
   wait_for_completion_timeout
   去等待你的传输完毕了。

3.2. pl08x_alloc_chan_resources

```
/*
 ===============================================================
```

```
 * The DMA ENGINE API
 *
 ===================================================================
 */
static int pl08x_alloc_chan_resources(struct dma_chan *chan,
     struct dma_client *client)
{
 struct pl08x_dma_chan *local_chan = container_of
    (chan, struct pl08x_dma_chan, chan);

 int assigned = 0;
 /*
  * Channels may be reserved for slaves
  * Channels doing slave DMA can only handle one client
  */
 if (local_chan-<slave_only) {
  if (!client-<slave)
   return -EBUSY;
 } else if (client-<slave)
  return -EBUSY;


 /* Slave DMA clients can only use one channel each */
 if ((client-<slave) && (chan-<client_count))
  return -EBUSY;
 else {
  ❶/* Channels doing slave DMA can only handle one client. */
  if ((local_chan-<slave) && (chan-<client_count)) {
   dev_err(&pd.dmac-<dev,
    "%s - channel %d already has a slave - local_chan-<slave %p\n",
     __func__,
     local_chan-<chan_id,
     local_chan-<slave);
   return -EBUSY;
  }

  /* Check channel is idle */
  if (pl08x_dma_busy(local_chan-<chan_id)) {
   dev_err(&pd.dmac-<dev,
    "%s - channel %d not idle\n",
     __func__, local_chan-<chan_id);
   return -EIO;
  }
  local_chan-<slave = client-<slave;
 }

 return assigned;
}
```

❶ 这部分没完全看懂，直译是用Slave DMA的channel，只能有一个client，但是具体为什么，不懂。。。

3.3. pl08x_free_chan_resources

```
static void pl08x_free_chan_resources(struct dma_chan *chan)
```

```
{
 dev_warn(&pd.dmac->dev, "%s - UNIMPLEMENTED\n", __func__);
}
```

3.4. pl08x_tx_submit

```
/*
 * First make the LLIs (could/should we do this earlier??)
 * slave (m/p) - no queued transactions allowed at present
 * TODO allow queued transactions for non circular buffers
 * Set up the channel active txd as inactive
 * m2m - transactions may be queued
 * If no active txd on channel
 * set it up as inactive
 * - issue_pending() will set active & start
 * else
 * queue it
 * Lock channel since there may be (at least for m2m) multiple calls
 *
 * Return < 0 for error
 */
static dma_cookie_t pl08x_tx_submit❶(struct dma_async_tx_descriptor *tx)
{
 int num_llis;
 unsigned long flags;
 struct pl08x_txd *local_txd = container_of(tx, struct pl08x_txd, tx);
 struct pl08x_dma_chan *local_chan =
   container_of(tx->chan, struct pl08x_dma_chan, chan);
 int pl08x_chan_num = local_chan->chan_id;
 num_llis = ❷fill_LLIS_for_desc(local_txd, pl08x_chan_num);

 if (num_llis) {
  spin_lock_irqsave(&local_chan->lock, flags);
  atomic_inc(&local_chan->last_issued);
  tx->cookie = atomic_read(&local_chan->last_issued);

  if (local_chan->at) {

   /*
    * If this device not using a circular buffer then
    * queue this new descriptor.
    * The descriptor for a circular buffer continues
    * to be used until the channel is freed
    */
   if (local_txd->pcd->circular_buffer)
    dev_err(&pd.dmac->dev,
     "%s - attempting to queue a circular buffer\n",
      __func__);
   else
    list_add_tail(&local_txd->txd_list,
     &local_chan->desc_list);

  } else {
   local_txd->slave = local_chan->slave;
   local_chan->at = local_txd;
```

```
  local_txd->active = 0;
 }

 spin_unlock_irqrestore(&local_chan->lock, flags);

 return tx->cookie;
} else
 return -EINVAL;
}
```

❶ 此submit函数是你之前准备好了一切后，最后调用此函数提交你的请求，但是实际并没开始真正的DMA传输，要等到最后的issue pending才真正开始

❷ 为当前的desc描述符，填充好对应的LLI

3.5. pl08x_prep_dma_memcpy

```
/*
 * Initialize a descriptor to be used by submit
 */
static struct dma_async_tx_descriptor *pl08x_prep_dma_memcpy❶(
  struct dma_chan *chan, dma_addr_t dest, dma_addr_t src,
  size_t len, unsigned long flags)
{
 struct pl08x_txd *local_txd;
 local_txd = kzalloc(sizeof(struct pl08x_txd), GFP_KERNEL);
 if (!local_txd) {
  dev_err(&pd.dmac->dev,
   "%s - no memory for descriptor\n", __func__);
  return NULL;
 } else {
  dma_async_tx_descriptor_init(&local_txd->tx, chan);
  local_txd->srcbus.addr = src;
  local_txd->dstbus.addr = dest;
  local_txd->tx.tx_submit = pl08x_tx_submit;
  local_txd->len = len;
  /*
   *  dmaengine.c has these directions hard coded,
   *  but not acessible
   */
  local_txd->dstdir = DMA_FROM_DEVICE;
  local_txd->srcdir = DMA_TO_DEVICE;
  INIT_LIST_HEAD(&local_txd->txd_list);
  /*
   * Ensure the platform data for m2m is set on the channel
   */
  local_txd->pcd = &pd.pd->sd[PL08X_DMA_SIGNALS];
 }
 return &local_txd->tx;
}
```

❶ 为DMA的memcpy做一些准备工作，主要就是初始化一些参数，尤其是传输方向

3.6. pl08x_prep_dma_interrupt

```
static struct dma_async_tx_descriptor *pl08x_prep_dma_interrupt(
  struct dma_chan *chan, unsigned long flags)
{
 struct dma_async_tx_descriptor *retval = NULL;
 return retval;
}
```

3.7. pl08x_dma_is_complete

```
/*
 * Code accessing dma_async_is_complete() in a tight loop
 * may give problems - could schedule where indicated.
 * If slaves are relying on interrupts to signal completion this
 * function must not be called with interrupts disabled
 */
static enum dma_status pl08x_dma_is_complete❶(struct dma_chan *chan,
      dma_cookie_t cookie,
      dma_cookie_t *done,
      dma_cookie_t *used)
{
 struct pl08x_dma_chan *local_chan = container_of(chan,
   struct pl08x_dma_chan, chan);
 dma_cookie_t last_used;
 dma_cookie_t last_complete;
 enum dma_status ret;

 last_used = atomic_read(&local_chan->last_issued);
 last_complete = local_chan->lc;

 if (done)
  *done = last_complete;
 if (used)
  *used = last_used;

 ret = dma_async_is_complete(cookie, last_complete, last_used);
 if (ret == DMA_SUCCESS)
  return ret;

 /*
  * schedule(); could be inserted here
  */

 /*
  * This cookie not complete yet
  */
 last_used = atomic_read(&local_chan->last_issued);
 last_complete = local_chan->lc;

 if (done)
  *done = last_complete;
 if (used)
```

```
 *used = last_used;

 return dma_async_is_complete(cookie, last_complete, last_used);
}
```

❶ 提供了一个函数，用于检测当前DMA是否已经完成了，实际好像很少用到，多数是用complete机制

3.8. pl08x_issue_pending

```
/*
 * Slave transactions callback to the slave device to allow
 * synchronization of slave DMA signals with the DMAC enable
 */
static void pl08x_issue_pending❶(struct dma_chan *chan)
{
 struct pl08x_dma_chan *local_chan
  = container_of(chan, struct pl08x_dma_chan, chan);
 int pl08x_chan_num = local_chan->chan_id;

 if (local_chan->at) {❷
  if (!local_chan->at->active) {
   pl08x_set_cregs(local_chan->at, local_chan->chan_id);
   if (local_chan->slave) {
    /*
     * Allow slaves to activate signals
     * concurrent to the DMAC enable
     */
    if (local_chan->at->tx.callback) {
     ((struct pl08x_callback_param *)
      local_chan->at->tx.callback_param)->act =
        PL08X_SIGNAL_START;
     local_chan->at->tx.callback(
      local_chan->at->tx.callback_param);
    }
   }
   pl08x_enable_dmac_chan(local_chan->chan_id);
   local_chan->at->active = 1;
  }
  /*
   * else skip active transfer
   * Calls with active txd occur for NET_DMA
   * - there can be queued descriptors
   */
 }
 /*
  * else - calls with no active descriptor occur for NET_DMA
  */
}
```

❶ 真正的提交DMA请求，进行DMA传输的函数
❷ 如果当前没人在用此channel，那么就可以真正去使用了，先去设置寄存器的值，然后通知client DMA开始了，最后真正的设置物理上的寄存器，开始DMA传输

3.9. pl08x_prep_slave_sg

```
struct dma_async_tx_descriptor *pl08x_prep_slave_sg❶(
  struct dma_chan *chan, struct scatterlist *sgl,
  unsigned int sg_len, enum dma_data_direction direction,
  unsigned long flags)
{
 struct pl08x_txd *local_txd;
 unsigned int reg = 0;
 int i;

 /*
  * Current implementation ASSUMES only one sg
  */
 if (sg_len != 1)
  BUG();

 local_txd = kmalloc(sizeof(struct pl08x_txd), GFP_KERNEL);
 if (!local_txd) {
  dev_err(&pd.dmac->dev, "%s - no local_txd\n", __func__);
  return NULL;
 } else {
  struct pl08x_dma_chan *local_chan =
   container_of(chan, struct pl08x_dma_chan, chan);

  dma_async_tx_descriptor_init(&local_txd->tx, chan);

  if (direction == DMA_TO_DEVICE) {

   local_txd->srcbus.addr = sgl->dma_address;
   local_txd->dstbus.addr =
    reg = local_chan->slave->tx_reg;

  } else if (direction == DMA_FROM_DEVICE) {
   local_txd->srcbus.addr =
    reg = local_chan->slave->rx_reg;
   local_txd->dstbus.addr = sgl->dma_address;
  } else {
   dev_err(&pd.dmac->dev,
    "%s - direction unsupported\n", __func__);
   return NULL;
  }
  /*
   * Find the device array entry for this txd
   * so that the txd has access to the peripheral data
   */
  for (i = 0; i < PL08X_DMA_SIGNALS; i++) {
   if (reg == (((unsigned int)(pd.pd->sd[i].io_addr))))❷
    break;
  }
  local_txd->pcd = &pd.pd->sd[i];
  local_txd->tx.tx_submit = pl08x_tx_submit;
  local_txd->len = sgl->length;
  INIT_LIST_HEAD(&local_txd->txd_list);
 }
```

```
 return &local_txd->tx;
}
```

❶ 上层程序调用此函数，对提出的DMA请求和scatter/gather list 信息进行预处理，其实主要就是根据你当前的一些参数，包括设备的DMA的地址进行匹配，找到合适的配置参数，用于以后的DMA各个参数的设置

❷ 找到和自己的匹配的那个参数，这些参数是之前在amba设备注册时候，已经设置和初始化好的一堆设置参数

3.10. pl08x_terminate_all

```
/*
 * CAUTION: Called by ALSA interrupt handler
 */
void pl08x_terminate_all❶(struct dma_chan *chan)
{
 struct pl08x_dma_chan *local_chan =
  container_of(chan, struct pl08x_dma_chan, chan);
 int pl08x_chan_num = local_chan->chan_id;

 if (local_chan->slave->dev) {
  pl08x_disable_dmac_chan(pl08x_chan_num);
  /*
   * Allow slaves to activate signals
   * concurrent to the DMAC enable
   */
  if (local_chan->at) {
   if (local_chan->at->tx.callback) {
    ((struct pl08x_callback_param *)
     local_chan->at->tx.callback_param)
      ->act = PL08X_SIGNAL_STOP;

    local_chan->at->tx.callback(
     local_chan->at->tx.callback_param);
   }
  }
 }
}
```

❶ 结束所有的通道