简介Linux DMA 功能及接口分析，你了解吗?

时间：2020-09-22 22:36:53 关键字： Linux dma 手机看文章

[导读] 从我们的直观感受来说，DMA并不是一个复杂的东西，要做的事情也很单纯直白。因此Linux kernel对它的抽象和实现，也应该简洁、易懂才是。不过现实却不甚乐观(个人感觉)，Linux kernel dmaengine framework的实现，真有点晦涩的感觉。为什么会这样呢?

从我们的直观感受来说，DMA并不是一个复杂的东西，要做的事情也很单纯直白。因此Linux kernel对它的抽象和实现，也应该简洁、易懂才是。不过现实却不甚乐观(个人感觉)，Linux kernel dmaengine framework的实现，真有点晦涩的感觉。为什么会这样呢?

1.前言

如果一个软件模块比较复杂、晦涩，要么是设计者的功力不够，要么是需求使然。当然，我们不敢对Linux kernel的那些大神们有丝毫怀疑和不敬，只能从需求上下功夫了：难道Linux kernel中的driver对DMA的使用上，有一些超出了我们日常的认知范围?

要回答这些问题并不难，将dmaengine framework为consumers提供的功能和API梳理一遍就可以了，这就是本文的目的。当然，也可以借助这个过程，加深对DMA的理解，以便在编写那些需要DMA传输的driver的时候，可以更游刃有余。

2. Slave-DMA API和Async TX API

从方向上来说，DMA传输可以分为4类：memory到memory、memory到device、device到memory以及device到device。Linux kernel作为CPU的代理人，从它的视角看，外设都是slave，因此称这些有device参与的传输(MEM2DEV、DEV2MEM、DEV2DEV)为Slave-DMA传输。而另一种memory到memory的传输，被称为Async TX。

为什么强调这种差别呢?因为Linux为了方便基于DMA的memcpy、memset等操作，在dma engine之上，封装了一层更为简洁的API(如下面图片1所示)，这种API就是Async TX API(以async_开头，例如async_memcpy、async_memset、async_xor等)。

图片1 DMA Engine API示意图

最后，因为memory到memory的DMA传输有了比较简洁的API，没必要直接使用dma engine提供的API，最后就导致dma engine所提供的API就特指为Slave-DMA API(把mem2mem剔除了)。

本文主要介绍dma engine为consumers提供的功能和API，因此就不再涉及Async TX API了(具体可参考本站后续的文章。

注1：Slave-DMA中的“slave”，指的是参与DMA传输的设备。而对应的，“master”就是指DMA controller自身。一定要明白“slave”的概念，才能更好的理解kernel dma engine中有关的术语和逻辑。

3. dma engine的使用步骤

注2：本文大部分内容翻译自kernel document[1]，喜欢读英语的读者可以自行参考。

对设备驱动的编写者来说，要基于dma engine提供的Slave-DMA API进行DMA传输的话，需要如下的操作步骤：

1)申请一个DMA channel。

2)根据设备(slave)的特性，配置DMA channel的参数。

3)要进行DMA传输的时候，获取一个用于识别本次传输(transaction)的描述符(descriptor)。

4)将本次传输(transacTIon)提交给dma engine并启动传输。

5)等待传输(transacTIon)结束。

然后，重复3~5即可。

上面5个步骤，除了3有点不好理解外，其它的都比较直观易懂，具体可参考后面的介绍。

3.1 申请DMA channel

任何consumer(文档[1]中称作client，也可称作slave driver，意思都差不多，不再特意区分)在开始DMA传输之前，都要申请一个DMA channel(有关DMA channel的概念，请参考[2]中的介绍)。

DMA channel(在kernel中由“struct dma_chan”数据结构表示)由provider(或者是DMA controller)提供，被consumer(或者client)使用。对consumer来说，不需要关心该数据结构的具体内容(我们会在dmaengine provider的介绍中在详细介绍)。

consumer可以通过如下的API申请DMA channel：

struct dma_chan *dma_request_chan(struct device *dev， const char *name);

该接口会返回绑定在指定设备(dev)上名称为name的dma channel。dma engine的provider和consumer可以使用device tree、ACPI或者struct dma_slave_map类型的match table提供这种绑定关系，具体可参考XXXX章节的介绍。

最后，申请得到的dma channel可以在不需要使用的时候通过下面的API释放掉：

void dma_release_channel(struct dma_chan *chan);

3.2 配置DMA channel的参数

driver申请到一个为自己使用的DMA channel之后，需要根据自身的实际情况，以及DMA controller的能力，对该channel进行一些配置。可配置的内容由struct dma_slave_config数据结构表示(具体可参考4.1小节的介绍)。driver将它们填充到一个struct dma_slave_config变量中后，可以调用如下API将这些信息告诉给DMA controller：

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

3.3 获取传输描述(tx descriptor)

DMA传输属于异步传输，在启动传输之前，slave driver需要将此次传输的一些信息(例如src/dst的buffer、传输的方向等)提交给dma engine(本质上是dma controller driver)，dma engine确认okay后，返回一个描述符(由struct dma_async_tx_descriptor抽象)。此后，slave driver就可以以该描述符为单位，控制并跟踪此次传输。

struct dma_async_tx_descriptor数据结构可参考4.2小节的介绍。根据传输模式的不同，slave driver可以使用下面三个API获取传输描述符(具体可参考DocumentaTIon/dmaengine/client.txt[1]中的说明)：

struct dma_async_tx_descriptor *dmaengine_prep_slave_sg(

struct dma_chan *chan， struct scatterlist *sgl，

unsigned int sg_len， enum dma_data_direcTIon direction，

unsigned long flags);

struct dma_async_tx_descriptor *dmaengine_prep_dma_cyclic(

struct dma_chan *chan， dma_addr_t buf_addr， size_t buf_len，

size_t period_len， enum dma_data_direction direction);

struct dma_async_tx_descriptor *dmaengine_prep_interleaved_dma(

struct dma_chan *chan， struct dma_interleaved_template *xt，

unsigned long flags);

dmaengine_prep_slave_sg用于在“scatter gather buffers”列表和总线设备之间进行DMA传输，参数如下：

注3：有关scatterlist 我们在[3][2]中有提及，后续会有专门的文章介绍它，这里暂且按下不表。

chan，本次传输所使用的dma channel。

sgl，要传输的“scatter gather buffers”数组的地址;

sg_len， “scatter gather buffers”数组的长度。

direction，数据传输的方向，具体可参考enum dma_data_direction (include/linux/dma-direction.h)的定义。

flags，可用于向dma controller driver传递一些额外的信息，包括(具体可参考enum dma_ctrl_flags中以DMA_PREP_开头的定义)：

DMA_PREP_INTERRUPT，告诉DMA controller driver，本次传输完成后，产生一个中断，并调用client提供的回调函数(可在该函数返回后，通过设置struct dma_async_tx_descriptor指针中的相关字段，提供回调函数，具体可参考4.2小节的介绍);

DMA_PREP_FENCE，告诉DMA controller driver，后续的传输，依赖本次传输的结果(这样controller driver就会小心的组织多个dma传输之间的顺序);

DMA_PREP_PQ_DISABLE_P、DMA_PREP_PQ_DISABLE_Q、DMA_PREP_CONTINUE，PQ有关的操作，TODO。

dmaengine_prep_dma_cyclic常用于音频等场景中，在进行一定长度的dma传输(buf_addr&buf_len)的过程中，每传输一定的byte(period_len)，就会调用一次传输完成的回调函数，参数包括：

chan，本次传输所使用的dma channel。

buf_addr、buf_len，传输的buffer地址和长度。

period_len，每隔多久(单位为byte)调用一次回调函数。需要注意的是，buf_len应该是period_len的整数倍。

direction，数据传输的方向。

dmaengine_prep_interleaved_dma可进行不连续的、交叉的DMA传输，通常用在图像处理、显示等场景中，具体可参考struct dma_interleaved_template结构的定义和解释(这里不再详细介绍，需要用到的时候，再去学习也okay)。

3.4 启动传输

通过3.3章节介绍的API获取传输描述符之后，client driver可以通过dmaengine_submit接口将该描述符放到传输队列上，然后调用dma_async_issue_pending接口，启动传输。

dmaengine_submit的原型如下：

dma_cookie_t dmaengine_submit(struct dma_async_tx_descriptor *desc)

参数为传输描述符指针，返回一个唯一识别该描述符的cookie，用于后续的跟踪、监控。

dma_async_issue_pending的原型如下：

void dma_async_issue_pending(struct dma_chan *chan);

参数为dma channel，无返回值。

注4：由上面两个API的特征可知，kernel dma engine鼓励client driver一次提交多个传输，然后由kernel(或者dma controller driver)统一完成这些传输。

3.5 等待传输结束

传输请求被提交之后，client driver可以通过回调函数获取传输完成的消息，当然，也可以通过dma_async_is_tx_complete等API，测试传输是否完成。不再详细说明了。

最后，如果等不及了，也可以使用dmaengine_pause、dmaengine_resume、dmaengine_terminate_xxx等API，暂停、终止传输，具体请参考kernel document[1]以及source code。

4. 重要数据结构说明

4.1 struct dma_slave_config

中包含了完成一次DMA传输所需要的所有可能的参数，其定义如下：

```
/* include/linux/dmaengine.h */
struct dma_slave_config {
enum dma_transfer_direction direction;
phys_addr_t src_addr;
phys_addr_t dst_addr;
enum dma_slave_buswidth src_addr_width;
enum dma_slave_buswidth dst_addr_width;
u32 src_maxburst;
```

u32 dst_maxburst;

bool device_fc;

unsigned int slave_id;

};

direction，指明传输的方向，包括(具体可参考enum dma_transfer_direction的定义和注释)：

DMA_MEM_TO_MEM，memory到memory的传输;

DMA_MEM_TO_DEV，memory到设备的传输;

DMA_DEV_TO_MEM，设备到memory的传输;

DMA_DEV_TO_DEV，设备到设备的传输。

注5：controller不一定支持所有的DMA传输方向，具体要看provider的实现。

注6：参考第2章的介绍，MEM to MEM的传输，一般不会直接使用dma engine提供的API。

src_addr，传输方向是dev2mem或者dev2dev时，读取数据的位置(通常是固定的FIFO地址)。对mem2dev类型的channel，不需配置该参数(每次传输的时候会指定);

dst_addr，传输方向是mem2dev或者dev2dev时，写入数据的位置(通常是固定的FIFO地址)。对dev2mem类型的channel，不需配置该参数(每次传输的时候会指定);

src_addr_width、dst_addr_width，src/dst地址的宽度，包括1、2、3、4、8、16、32、64(bytes)等(具体可参考enum dma_slave_buswidth 的定义)。

src_maxburst、dst_maxburst，src/dst最大可传输的burst size(可参考[2]中有关burst size的介绍)，单位是src_addr_width/dst_addr_width(注意，不是byte)。

device_fc，当外设是Flow Controller(流控制器)的时候，需要将该字段设置为true。CPU中有关DMA和外部设备之间连接方式的设计中，决定DMA传输是否结束的模块，称作flow controller，DMA controller或者外部设备，都可以作为flow controller，具体要看外设和DMA controller的设计原理、信号连接方式等，不在详细说明(感兴趣的同学可参考[4]中的介绍)。

slave_id，外部设备通过slave_id告诉dma controller自己是谁(一般和某个request line对应)。很多dma controller并不区分slave，只要给它src、dst、len等信息，它就可以进行传输，因此slave_id可以忽略。而有些controller，必须清晰地知道此次传输的对象是哪个外设，就必须要提供slave_id了(至于怎么提供，可dma controller的硬件以及驱动有关，要具体场景具体对待)。

4.2 struct dma_async_tx_descriptor

传输描述符用于描述一次DMA传输(类似于一个文件句柄)。client driver将自己的传输请求通过3.3中介绍的API提交给dma controller driver后，controller driver会返回给client driver一个描述符。

client driver获取描述符后，可以以它为单位，进行后续的操作(启动传输、等待传输完成、等等)。也可以将自己的回调函数通过描述符提供给controller driver。

传输描述符的定义如下：

struct dma_async_tx_descriptor {

```
dma_cookie_t cookie;
enum dma_ctrl_flags flags; /* not a 'long' to pack with cookie */
dma_addr_t phys;
struct dma_chan *chan;
dma_cookie_t (*tx_submit)(struct dma_async_tx_descriptor *tx);
int (*desc_free)(struct dma_async_tx_descriptor *tx);
dma_async_tx_callback callback;
void *callback_param;
struct dmaengine_unmap_data *unmap;
#ifdef CONFIG_ASYNC_TX_ENABLE_CHANNEL_SWITCH
struct dma_async_tx_descriptor *next;
struct dma_async_tx_descriptor *parent;
spinlock_t lock;
#endif
};
```

cookie，一个整型数，用于追踪本次传输。一般情况下，dma controller driver会在内部维护一个递增的number，每当client获取传输描述的时候(参考3.3中的介绍)，都会将该number赋予cookie，然后加一。

注7：有关cookie的使用场景，我们会在后续的文章中再详细介绍。

flags， DMA_CTRL_开头的标记，包括：

DMA_CTRL_REUSE，表明这个描述符可以被重复使用，直到它被清除或者释放;

DMA_CTRL_ACK，如果该flag为0，表明暂时不能被重复使用。

phys，该描述符的物理地址??不太懂!

chan，对应的dma channel。

tx_submit，controller driver提供的回调函数，用于把改描述符提交到待传输列表。通常由dma engine调用，client driver不会直接和该接口打交道。

desc_free，用于释放该描述符的回调函数，由controller driver提供，dma engine调用，client driver不会直接和该接口打交道。

callback、callback_param，传输完成的回调函数(及其参数)，由client driver提供。

后面其它参数，client driver不需要关心，暂不描述了。

5. 参考文档

[1] Documentation/dmaengine/client.txt

[2] Linux DMA Engine framework(1)_概述

[3] Linux MMC framework(2)_host controller driver

[4] https://forums.xilinx.com/xlnx/attachments/xlnx/ELINUX/10658/1/drivers-session4-dma-4public.pdf

以上就是今天的介绍啦，你了解了吗?