Linux 下的DMA浅析

原创 zqixiao_09 于 2016-04-07 20:55:54 发布 29492 收藏 108 版权

分类专栏： Linux 系统 Linux 驱动开发基础 Linux 驱动开发 文章标签： Linux DMA

DMA是一种无需CPU的参与就可以让外设和系统内存之间进行双向数据传输的硬件机制。使用DMA可以使系统CPU从实际的I/O数据传输过程中摆脱出来，从而大大提高系统的吞吐率。DMA经常与硬件 体系结构 特别是外设的总线技术密切相关。

一、DMA控制器硬件结构

DMA允许外围设备和主内存之间直接传输 I/O 数据， DMA 依赖于系统。每一种体系结构DMA传输不同，编程接口也不同。

数据传输可以以两种方式触发：一种**软件请求数据**，另一种**由硬件异步传输。**

a -- 软件请求数据

调用的步骤可以概括如下（以read为例）：

（1）在进程调用 read 时，驱动程序的方法分配一个 DMA 缓冲区，随后指示硬件传送它的数据。进程进入睡眠。
（2）硬件将数据写入 DMA 缓冲区并在完成时产生一个中断。
（3）中断处理程序获得输入数据，应答中断，最后唤醒进程，该进程现在可以读取数据了。

b -- 由硬件异步传输

在 DMA 被异步使用时发生的。以数据采集设备为例：

（1）硬件发出中断来通知新的数据已经到达。
（2）中断处理程序分配一个DMA缓冲区。
（3）外围设备将数据写入缓冲区，然后在完成时发出另一个中断。
（4）处理程序利用DMA分发新的数据，唤醒任何相关进程。

网卡传输也是如此，网卡有一个**循环缓冲区（通常叫做 DMA 环形缓冲区）**建立在与处理器共享的内存中。每一个输入数据包被放置在环形缓冲区中下一个可用缓冲区，并且发出中断。然后驱动程序将网络数据包传给内核的其它部分处理，并在环形缓冲区中放置一个新的 DMA 缓冲区。

驱动程序在初始化时分配DMA缓冲区，并使用它们直到停止运行。

二、DMA通道使用的地址

DMA通道用**dma_chan结构数组**表示，这个结构在kernel/dma.c中，列出如下：

```
struct dma_chan {
    int  lock;
    const char *device_id;
};

static struct dma_chan dma_chan_busy[MAX_DMA_CHANNELS] = {
    [4] = { 1, "cascade" },
};
```

如果dma_chan_busy[n].lock != 0表示忙，DMA0保留为DRAM更新用，DMA4用作级联。DMA 缓冲区的主要问题是，当它大于一页时，它必须占据物理内存中的连续页。
由于DMA需要连续的内存，因而在引导时分配内存或者为缓冲区保留物理 RAM 的顶部。在引导时给内核传递一个"mem="参数可以保留 RAM 的顶部。例如，如果系统有 32MB 内存，参数"mem=31M"阻止内核使用最顶部的一兆字节。稍后，模块可以使用下面的代码来访问这些保留的内存：

dmabuf = ioremap(0x1F00000 /* 31M */, 0x100000 /* 1M */);

分配 DMA 空间的方法，代码调用 **kmalloc（GFP_ATOMIC）** 直到失败为止，然后它等待内核释放若干页面，接下来再一次进行分配。最终会发现由连续页面组成的DMA 缓冲区的出现。

一个使用 DMA 的设备驱动程序通常会与连接到接口总线上的硬件通讯，这些硬件使用物理地址，而程序代码使用虚拟地址。基于 DMA 的硬件使用总线地址而不是物理地址，有时，接口总线是通过将 I/O 地址映射到不同物理地址的桥接电路连接的。甚至某些系统有一个页面映射方案，能够使任意页面在外围总线上表现为连续的。

当驱动程序需要向一个 I/O 设备（例如扩展板或者DMA控制器）发送地址信息时，必须使用 virt_to_bus 转换，在接受到来自连接到总线上硬件的地址信息时，必须使用 bus_to_virt 了。

三、DMA操作函数

写一个DMA驱动的主要工作包括：**DMA通道申请、DMA中断申请、控制寄存器设置、挂入DMA等待队列、清除DMA中断、释放DMA通道**

因为 DMA 控制器是一个系统级的资源，所以内核协助处理这一资源。内核使用 **DMA 注册表**为 DMA 通道提供了请求/释放机制，并且提供了一组函数在 DMA 控制器中配置通道信息。

以下具体分析关键函数（linux/arch/arm/mach-s3c2410/dma.c）

```
int s3c2410_request_dma(const char *device_id, dmach_t channel,
    dma_callback_t write_cb, dma_callback_t read_cb) (s3c2410_dma_queue_buffer);
/*
函数描述：申请某通道的DMA资源，填充s3c2410_dma_t 数据结构的内容，申请DMA中断。
输入参数：device_id DMA 设备名；channel 通道号；
write_cb DMA写操作完成的回调函数；read_cb DMA读操作完成的回调函数
输出参数：若channel通道已使用，出错返回；否则，返回0
*/

int s3c2410_dma_queue_buffer(dmach_t channel, void *buf_id,
dma_addr_t data, int size, int write) (s3c2410_dma_stop);
/*
函数描述：这是DMA操作最关键的函数，它完成了一系列动作：分配并初始化一个DMA内核缓冲区控制结构，并将它插入DMA等待队列，设置DMA控制寄存器内容，等
输入参数： channel 通道号；buf_id,缓冲区标识
dma_addr_t data DMA数据缓冲区起始物理地址；size DMA数据缓冲区大小；write 是写还是读操作
输出参数：操作成功，返回0；否则，返回错误号
*/

int s3c2410_dma_stop(dmach_t channel)
//函数描述：停止DMA操作。

int s3c2410_dma_flush_all(dmach_t channel)
//函数描述：释放DMA通道所申请的所有内存资源

void s3c2410_free_dma(dmach_t channel)
//函数描述：释放DMA通道
```

四、DMA映射

一个DMA映射就是分配一个 DMA 缓冲区并为该缓冲区生成一个能够被设备访问的地址的组合操作。一般情况下，简单地调用函数virt_to_bus 就设备总线上的地址，但有些硬件映射寄存器也被设置在总线硬件中。映射寄存器（mapping register）是一个类似于外围设备的虚拟内存等价物。在使用这些寄存器的系统上，外围设备有一个相对较小的、专用的地址区段，可以在此区段执行 DMA。通过映射寄存器，这些地址被重映射到系统 RAM。映射寄存器具有一些好的特性，包括使分散的页面在设备地址空间看起来是连续的。但不是所有的体系结构都有映射寄存器，特别地，PC 平台没有映射寄存器。

在某些情况下，为设备设置有用的地址也意味着需要构造一个反弹（bounce）缓冲区。例如，当驱动程序试图在一个不能被外围设备访问的地址（一个高端内存地址）上执行 DMA 时，反弹缓冲区被创建。然后，按照需要，数据被复制到反弹缓冲区，或者从反弹缓冲区复制。

根据 DMA 缓冲区期望保留的时间长短，PCI 代码区分两种类型的 DMA 映射：

a -- 一致 DMA 映射

它们存在于驱动程序的生命周期内。一个被一致映射的缓冲区必须同时可被 CPU 和外围设备访问，这个缓冲区被处理器写时，可立即被设备读取而没有cache效应，反之亦然，使用函数pci_alloc_consistent建立一致映射。

b -- 流式 DMA映射

流式DMA映射是为单个操作进行的设置。它映射处理器虚拟空间的一块地址，以致它能被设备访问。应尽可能使用流式映射，而不是一致映射。这是因为在支持一致映射的系统上，每个 DMA 映射会使用总线上一个或多个映射寄存器。具有较长生命周期的一致映射，会独占这些寄存器很长时间——即使它们没有被使用。使用函数dma_map_single建立流式映射。

1、建立一致 DMA 映射

函数**pci_alloc_consistent**处理缓冲区的分配和映射，函数分析如下（在include/asm-generic/pci-dma-compat.h中）：

```
static inline void *pci_alloc_consistent(struct pci_dev *hwdev,
                                size_t size, dma_addr_t *dma_handle)
{
    return dma_alloc_coherent(hwdev == NULL ? NULL : &hwdev->dev,
                        size, dma_handle, GFP_ATOMIC);
}
```

结构**dma_coherent_mem**定义了DMA一致性映射的内存的地址、大小和标识等。结构dma_coherent_mem列出如下（在arch/i386/kernel/pci-dma.c中）：

```
struct dma_coherent_mem {
    void        *virt_base;
    u32     device_base;
    int     size;
    int     flags;
    unsigned long   *bitmap;
};
```

函数**dma_alloc_coherent**分配size字节的区域的一致内存，得到的dma_handle是指向分配的区域的地址指针，这个地址作为区域的物理基地址。dma_handle是与总线一样的位宽的无符号整数。 函数dma_alloc_coherent分析如下（在arch/i386/kernel/pci-dma.c中）：

```
void *dma_alloc_coherent(struct device *dev, size_t size,
               dma_addr_t *dma_handle, int gfp)
{
    void *ret;
    //若是设备，得到设备的dma内存区域，即mem= dev->dma_mem
    struct dma_coherent_mem *mem = dev ? dev->dma_mem : NULL;
    int order = get_order(size);//将size转换成order，即
    //忽略特定的区域，因而忽略这两个标识
    gfp &= ~(__GFP_DMA | __GFP_HIGHMEM);

    if (mem) {//设备的DMA映射，mem= dev->dma_mem
       //找到mem对应的页
        int page = bitmap_find_free_region(mem->bitmap, mem->size,
                            order);
        if (page >= 0) {
            *dma_handle = mem->device_base + (page << PAGE_SHIFT);
            ret = mem->virt_base + (page << PAGE_SHIFT);
            memset(ret, 0, size);
            return ret;
        }
        if (mem->flags & DMA_MEMORY_EXCLUSIVE)
            return NULL;
    }

    //不是设备的DMA映射
    if (dev == NULL || (dev->coherent_dma_mask < 0xffffffff))
        gfp |= GFP_DMA;
    //分配空闲页
```

```
    ret = (void *)__get_free_pages(gfp, order);

    if (ret != NULL) {
        memset(ret, 0, size);//清0
        *dma_handle = virt_to_phys(ret);//得到物理地址
    }
    return ret;
}
```

当不再需要缓冲区时（通常在模块卸载时），应该调用函数 pci_free_consitent 将它返还给系统。

2、建立流式 DMA 映射

在流式 DMA 映射的操作中，缓冲区传送方向应匹配于映射时给定的方向值。缓冲区被映射后，它就属于设备而不再属于处理器了。在缓冲区调用函数pci_unmap_single撤销映射之前，驱动程序不应该触及其内容。

在缓冲区为 DMA 映射时，内核必须确保缓冲区中所有的数据已经被实际写到内存。可能有些数据还会保留在处理器的高速缓冲存储器中，因此必须显式刷新。在刷新之后，由处理器写入缓冲区的数据对设备来说也许是不可见的。

如果欲映射的缓冲区位于设备不能访问的内存区段时，某些体系结构仅仅会操作失败，而其它的体系结构会创建一个反弹缓冲区。反弹缓冲区是被设备访问的独立内存区域，反弹缓冲区复制原始缓冲区的内容。

函数**pci_map_single**映射单个用于传送的缓冲区，返回值是可以传递给设备的总线地址，如果出错的话就为 NULL。一旦传送完成，应该使用函数pci_unmap_single 删除映射。其中，参数direction为传输的方向，取值如下：

PCI_DMA_TODEVICE 数据被发送到设备。
PCI_DMA_FROMDEVICE如果数据将发送到 CPU。
PCI_DMA_BIDIRECTIONAL数据进行两个方向的移动。
PCI_DMA_NONE 这个符号只是为帮助调试而提供。

函数**pci_map_single**分析如下（在arch/i386/kernel/pci-dma.c中）

```
static inline dma_addr_t  pci_map_single(struct pci_dev *hwdev,
                            void *ptr, size_t size, int direction)
{
    return dma_map_single(hwdev == NULL ? NULL : &hwdev->dev, ptr, size,
                                    (enum ma_data_direction)direction);
}
```

函数**dma_map_single**映射一块处理器虚拟内存，这块虚拟内存能被设备访问，返回内存的物理地址，函数dma_map_single分析如下（在include/asm-i386/dma-mapping.h中）：

```
static inline dma_addr_t dma_map_single(struct device *dev, void *ptr,
                        size_t size, enum dma_data_direction direction)

{
    BUG_ON(direction == DMA_NONE);
    //可能有些数据还会保留在处理器的高速缓冲存储器中，因此必须显式刷新
    flush_write_buffers();
    return virt_to_phys(ptr);  //虚拟地址转化为物理地址

}
```

3、分散/集中映射

分散/集中映射是流式 DMA 映射的一个特例。它将几个缓冲区集中到一起进行一次映射，并在一个 DMA 操作中传送所有数据。这些分散的缓冲区由分散表结构scatterlist来描述，多个分散的缓冲区的分散表结构组成缓冲区的**struct scatterlist数组**。

分散表结构列出如下（在include/asm-i386/scatterlist.h）：

```
struct scatterlist {
    struct page     *page;
```

```
    unsigned int    offset;         dma_addr_t      dma_address;  //用在分散/集中操作中的缓冲区地址
    unsigned int    length;//该缓冲区的长度
};
```

每一个缓冲区的地址和长度会被存储在 struct scatterlist 项中，但在不同的体系结构中它们在结构中的位置是不同的。下面的两个宏定义来解决平台移植性问题，这些宏定义应该在一个**pci_map_sg** 被调用后使用：

```
//从该分散表项中返回总线地址
#define sg_dma_address(sg)  (sg)->dma_address)
//返回该缓冲区的长度
#define sg_dma_len(sg)      (sg)->length)
```

函数 **pci_map_sg**完成分散/集中映射，其返回值是要传送的 DMA 缓冲区数；它可能会小于 nents（也就是传入的分散表项的数量），因为可能有的缓冲区地址上是相邻的。一旦传输完成，分散/集中映射通过调用函数pci_unmap_sg 来撤销映射。 函数pci_map_sg分析如下（在include/asm-generic/pci-dma-compat.h中）：

```
static inline int pci_map_sg(struct pci_dev *hwdev, struct scatterlist *sg,
                             int nents, int direction)
{
    return dma_map_sg(hwdev == NULL ? NULL : &hwdev->dev, sg, nents,
(enum dma_data_direction)direction);
}
include/asm-i386/dma-mapping.h
static inline int dma_map_sg(struct device *dev, struct scatterlist *sg,
int nents, enum dma_data_direction direction)
{
    int i;

    BUG_ON(direction == DMA_NONE);

    for (i = 0; i < nents; i++ ) {
        BUG_ON(!sg[i].page);
       //将页及页偏移地址转化为物理地址
        sg[i].dma_address = page_to_phys(sg[i].page) + sg[i].offset;
    }
    //可能有些数据还会保留在处理器的高速缓冲存储器中，因此必须显式刷新
    flush_write_buffers();
    return nents;
}
```

五、DMA池

许多驱动程序需要又多又小的一致映射内存区域给DMA描述子或I/O缓存buffer，这使用DMA池比用dma_alloc_coherent分配的一页或多页内存区域好，DMA池用函数dma_pool_create创建，用函数dma_pool_alloc从DMA池中分配一块一致内存，用函数dmp_pool_free放内存回到DMA池中，使用函数dma_pool_destory释放DMA池的资源。

结构dma_pool是DMA池描述结构，列出如下：

```
struct dma_pool {   /* the pool */
    struct list_head    page_list;//页链表
    spinlock_t      lock;
    size_t          blocks_per_page;  //每页的块数
    size_t          size;     //DMA池里的一致内存块的大小
    struct device       *dev; //将做DMA的设备
    size_t          allocation; //分配的没有跨越边界的块数，是size的整数倍
    char            name [32];  //池的名字
    wait_queue_head_t   waitq;  //等待队列
    struct list_head    pools;
};
```

函数**dma_pool_create**给DMA创建一个一致内存块池，其参数name是DMA池的名字，用于诊断用，参数dev是将做DMA的设备，参数size是DMA池里的块的大小，参数align是块的对齐要求，是2的幂，参数allocation返回没有跨越边界的块数（或0）。

函数dma_pool_create返回创建的带有要求字符串的DMA池，若创建失败返回null。对被给的DMA池，函数dma_pool_alloc被用来分配内存，这些内存都是一致DMA映射，可被设备访问，且没有使用缓存刷新机制，因为对齐原因，分配的块的实际尺寸比请求的大。如果分配非0的内存，从函数dma_pool_alloc返回的对象将不跨越size边界（如不跨越4K字节边界）。这对在个体的DMA传输上有地址限制的设备来说是有利的。

函数dma_pool_create分析如下（在drivers/base/dmapool.c中）：

```
struct dma_pool *dma_pool_create (const char *name, struct device *dev,
                        size_t size, size_t align, size_t allocation)
{
    struct dma_pool     *retval;

    if (align == 0)
        align = 1;
    if (size == 0)
        return NULL;
    else if (size < align)
        size = align;
    else if ((size % align) != 0) {//对齐处理
        size += align + 1;
        size &= ~(align - 1);
    }
    //如果一致内存块比页大，是分配为一致内存块大小，否则，分配为页大小
    if (allocation == 0) {
        if (PAGE_SIZE < size)//页比一致内存块小
            allocation = size;
        else
            allocation = PAGE_SIZE;//页大小
        // FIXME: round up for less fragmentation
    } else if (allocation < size)
        return NULL;
    //分配dma_pool结构对象空间
    if (!(retval = kmalloc (sizeof *retval, SLAB_KERNEL)))
        return retval;

    strlcpy (retval->name, name, sizeof retval->name);

    retval->dev = dev;
    //初始化dma_pool结构对象retval
    INIT_LIST_HEAD (&retval->page_list);//初始化页链表
    spin_lock_init (&retval->lock);
    retval->size = size;
    retval->allocation = allocation;
    retval->blocks_per_page = allocation / size;
    init_waitqueue_head (&retval->waitq);//初始化等待队列

    if (dev) {//设备存在时
        down (&pools_lock);
        if (list_empty (&dev->dma_pools))
           //给设备创建sysfs文件系统属性文件
            device_create_file (dev, &dev_attr_pools);
        /* note:  not currently insisting "name" be unique */
        list_add (&retval->pools, &dev->dma_pools);  //将DMA池加到dev中
        up (&pools_lock);
    } else
        INIT_LIST_HEAD (&retval->pools);

    return retval;
}
```

函数dma_pool_alloc从DMA池中分配一块一致内存，其参数pool是将产生块的DMA池，参数mem_flags是GFP_*位掩码，参数handle是指向块的DMA地址，函数dma_pool_alloc返回当前没用的块的内核虚拟地址，并通过handle给出它的DMA地址，如果内存块不能被分配，返回null。

函数dma_pool_alloc包裹了dma_alloc_coherent页分配器，这样小块更容易被总线的主控制器使用。这可能共享slab分配器的内容。

函数dma_pool_alloc分析如下（在drivers/base/dmapool.c中）：

```
void *dma_pool_alloc (struct dma_pool *pool, int mem_flags, dma_addr_t *handle)
{
    unsigned long       flags;
    struct dma_page     *page;
    int         map, block;
    size_t          offset;
    void            *retval;

restart:
    spin_lock_irqsave (&pool->lock, flags);
    list_for_each_entry(page, &pool->page_list, page_list) {
        int     i;
        /* only cachable accesses here ... */
        //遍历一页的每块，而每块又以32字节递增
        for (map = 0, i = 0;
                i < pool->blocks_per_page;  //每页的块数
                i += BITS_PER_LONG, map++) {  // BITS_PER_LONG定义为32
            if (page->bitmap [map] == 0)
                continue;
            block = ffz (~ page->bitmap [map]);//找出第一个0
            if ((i + block) < pool->blocks_per_page) {
                clear_bit (block, &page->bitmap [map]);
             //得到相对于页边界的偏移
                offset = (BITS_PER_LONG * map) + block;
                offset *= pool->size;
                goto ready;
            }
        }
    }
//给DMA池分配dma_page结构空间，加入到pool->page_list链表，
//并作DMA一致映射，它包括分配给DMA池一页。
// SLAB_ATOMIC表示调用 kmalloc（GFP_ATOMIC） 直到失败为止，
//然后它等待内核释放若干页面，接下来再一次进行分配。
    if (!(page = pool_alloc_page (pool, SLAB_ATOMIC))) {
        if (mem_flags & __GFP_WAIT) {
            DECLARE_WAITQUEUE (wait, current);

            current->state = TASK_INTERRUPTIBLE;
            add_wait_queue (&pool->waitq, &wait);
            spin_unlock_irqrestore (&pool->lock, flags);

            schedule_timeout (POOL_TIMEOUT_JIFFIES);

            remove_wait_queue (&pool->waitq, &wait);
            goto restart;
        }
        retval = NULL;
        goto done;
    }

    clear_bit (0, &page->bitmap [0]);
    offset = 0;
ready:
    page->in_use++;
    retval = offset + page->vaddr;  //返回虚拟地址
    *handle = offset + page->dma;  //相对DMA地址
#ifdef  CONFIG_DEBUG_SLAB
    memset (retval, POOL_POISON_ALLOCATED, pool->size);
#endif
done:
    spin_unlock_irqrestore (&pool->lock, flags);
    return retval;
}
```

六、一个简单的使用DMA 例子

示例：下面是一个简单的使用DMA进行传输的驱动程序，它是一个假想的设备，只列出DMA相关的部分来说明驱动程序中如何使用DMA的。

函数dad_transfer是设置DMA对内存buffer的传输操作函数，它使用流式映射将buffer的虚拟地址转换到物理地址，设置好DMA控制器，然后开始传输数据。

```
int dad_transfer(struct dad_dev *dev, int write, void *buffer,
                 size_t count)
{
   dma_addr_t bus_addr;
   unsigned long flags;

   /* Map the buffer for DMA */
   dev->dma_dir = (write ? PCI_DMA_TODEVICE : PCI_DMA_FROMDEVICE);
   dev->dma_size = count;
   //流式映射，将buffer的虚拟地址转化成物理地址
   bus_addr = pci_map_single(dev->pci_dev, buffer, count,
                             dev->dma_dir);
   dev->dma_addr = bus_addr; //DMA传送的buffer物理地址

   //将操作控制写入到DMA控制器寄存器，从而建立起设备
   writeb(dev->registers.command, DAD_CMD_DISABLEDMA);
  //设置传输方向--读还是写
   writeb(dev->registers.command, write ? DAD_CMD_WR : DAD_CMD_RD);
   writel(dev->registers.addr, cpu_to_le32(bus_addr));//buffer物理地址
   writel(dev->registers.len, cpu_to_le32(count)); //传输的字节数

   //开始激活DMA进行数据传输操作
   writeb(dev->registers.command, DAD_CMD_ENABLEDMA);
   return 0;
}
```

函数dad_interrupt是中断处理函数，当DMA传输完时，调用这个中断函数来取消buffer上的DMA映射，从而让内核程序可以访问这个buffer。

```
void dad_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
  struct dad_dev *dev = (struct dad_dev *) dev_id;

  /* Make sure it's really our device interrupting */

  /* Unmap the DMA buffer */
  pci_unmap_single(dev->pci_dev, dev->dma_addr, dev->dma_size,
       dev->dma_dir);

  /* Only now is it safe to access the buffer, copy to user, etc. */
  ...
}
```

函数dad_open打开设备，此时应申请中断号及DMA通道

```
int dad_open (struct inode *inode, struct file *filp)
{
  struct dad_device *my_device;

  // SA_INTERRUPT表示快速中断处理且不支持共享 IRQ 信号线
  if ( (error = request_irq(my_device.irq, dad_interrupt,
                            SA_INTERRUPT, "dad", NULL)) )
      return error; /* or implement blocking open */

  if ( (error = request_dma(my_device.dma, "dad")) ) {
      free_irq(my_device.irq, NULL);
      return error; /* or implement blocking open */
  }

  return 0;
```

```
}
```

在与open 相对应的 close 函数中应该释放DMA及中断号。

```
void dad_close (struct inode *inode, struct file *filp)
{
  struct dad_device *my_device;
  free_dma(my_device.dma);
  free_irq(my_device.irq, NULL);
  ……
}
```

函数dad_dma_prepare初始化DMA控制器，设置DMA控制器的寄存器的值，为 DMA 传输作准备。

```
int dad_dma_prepare(int channel, int mode, unsigned int buf,
                    unsigned int count)
{
  unsigned long flags;

  flags = claim_dma_lock();
  disable_dma(channel);
  clear_dma_ff(channel);
  set_dma_mode(channel, mode);
  set_dma_addr(channel, virt_to_bus(buf));
  set_dma_count(channel, count);
  enable_dma(channel);
  release_dma_lock(flags);

  return 0;
}
```

函数dad_dma_isdone用来检查 DMA 传输是否成功结束。

```
int dad_dma_isdone(int channel)
{
   int residue;
   unsigned long flags = claim_dma_lock ();
   residue = get_dma_residue(channel);
   release_dma_lock(flags);
   return (residue == 0);
}
```