努力成为 linux kernel hacker 的人李万鹏原创作品，为梦而战。转载请标明出处
http://blog.csdn.net/woshixingaaa/archive/2011/06/02/6462151.aspx

```
1. void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

内存映射函数 mmap 负责把文件内容映射到进程的虚拟内存空间，通过对这段内存的读取和修改，来实现对文件的读取和修改，而不需要再调用 read,write 等操作。
addr:指定映射的起始地址，通常设为 NULL，由系统指定。
length:映射到内存的文件长度。
prot:映射的保护方式，可以是：
PROT_EXEC：映射区可被执行
PROT_READ：映射区可被读取
PROT_WRITE：映射区可被写入
PROT_NONE：映射区不能存取
Flags:映射区的特性，可以是：
MAP_SHARED：
写入映射区的数据会复制回文件，且允许其他映射该文件的进程共享。
MAP_PRIVATE：
对映射区的写入操作会产生一个映射区的复制(copy_on_write)，对此区域所做的修改不会写回原文件。
fd:由 open 返回的文件描述符，代表要映射的文件。
offset:以文件开始处的偏移量，必须是分页大小的整数倍，通常为 0，表示从文件头开始映射。
解除映射：

```
1. int munmap(void *start, size_t length);
```

功能：取消参数 start 所指向的映射内存，参数 length 表示欲取消的内存大小。

返回值：解除成功返回 0，否则返回-1 错误原因存在于 errno 中。
虚拟地址区域：vm_area_struct
Linux 内核使用结构 vm_area_struct(<linux/mm_types.h>)描述虚拟内存区域，其中几个主要成员如下：
unsigned long vm_start 虚拟内存区域起始地址
unsigned long vm_end　虚拟内存区域结束地址

unsigned long vm_flags 该区域的标志
如：VM_IO 和 VM_RESERVED。VM_IO 将该 VMA 标记为内存映射的 IO 区域，VM_IO 会阻止系统将该区域包含在进程的存放转存(core dump)中，VM_RESERVED 标志内存区域不能被换出。
mmap 设备操作
映射一个设备是指把用户空间的一段地址关联到设备内存上，当程序读写这段用户空间的地址时，它实际上是在访问设备。这里需要做的两个操作：
1.找到可以用来关联的虚拟地址区间
2.关联
其中找到可以用来关联的虚拟地址区间是由内核完成的，mmap 只要关联这个操作。
mmap 方法是 file_operations 结构的成员，在 mmap 系统调用发出时被调用。在此之前，内核已经完成了很多工作。mmap 设备方法所需要做的就是建立虚拟地址到物理地址的页表。

```
1. void (*mmap)(struct file*, struct vm_area_struct *);
```

其中第二个参数 struct vm_area_struct *相当于内核找到的，可以拿来用的虚拟内存区间。
mmap 完成页表的建立：
方法有二：
1.使用 remap_pfn_range 一次建立所有页表；
2.使用 nopage VMA 方法每次建立一个页表；
构造页表的工作可由 remap_pgn_range 函数完成，原型如下：

```
1. int remap_pfn_range(struct vm_area_struct *vma, unsigned long addr, unsigned long pfn, unsigned long size, pgprot_t prot);
```

vma 是内核为我们找到的虚拟地址空间，addr 要关联的是虚拟地址，pfn 是要关联的物理地址，size 是关联的长度是多少。
mmap 设备操作实例：

```
1. int memdev_map(struct file *filp, struct vm_area_struct *vma){
2.     vma->vm_flags |= VM_IO;
3.     vma->vm_flags |= VM_RESERVED;
4.     if(remap_pgn_range(vma,vma->start, virt_to_phys(dev->data>>PAGE_SHIFT),size,vma->vm_page_prot))
5.         return -EAGAIN;
6.     return 0;
7. }
```

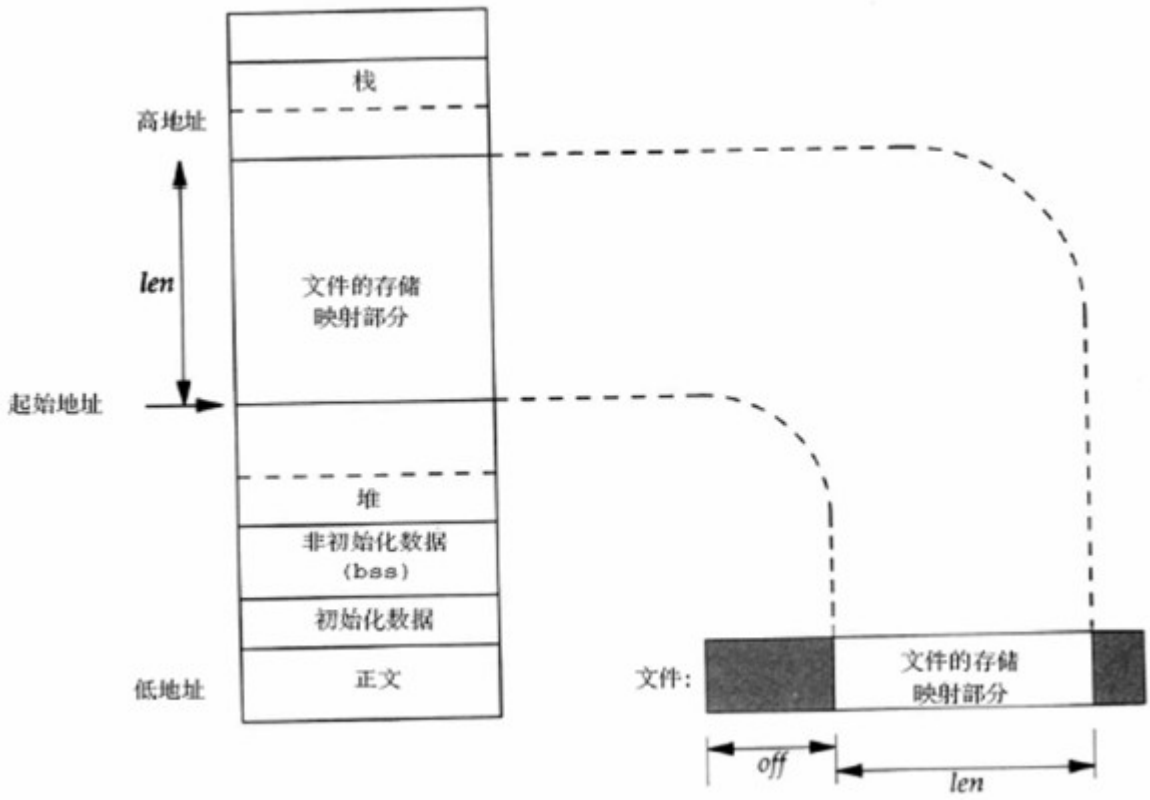
先说一下对于 ARM 而言虚拟地址与物理地址的关系：
在 arch/arm/include/asm/memory.h 中：

```
#define __virt_to_phys(x)   ((x) - PAGE_OFFSET + PHYS_OFFSET)
#define __phys_to_virt(x)   ((x) - PHYS_OFFSET + PAGE_OFFSET)

static inline unsigned long virt_to_phys(void *x)
{
    return __virt_to_phys((unsigned long)(x));
}

static inline void *phys_to_virt(unsigned long x)
{
    return (void *)(__phys_to_virt((unsigned long)(x)));
}
```

上面转换过程的 PAGE_OFFSET 通常为 3G，而 PHYS_OFFSET 则定于为系统 DRAM 内存的基地址。因此，对于我们的开发板，并不是将 0 地址映射到 3G，而是将外接的 SDRAM 的首地址映射到 3G。注意：这里的 virt_to_phys 和 phys_to_virt 方法仅适用于 896MB 以下的低端内存，高端内存的虚拟地址与物理地址之间不存在如此简单的换算关系。下边是 fbmem.c 中的 mmap 操作,示意图如下：




```
static int
fb_mmap(struct file *file, struct vm_area_struct * vma)
__acquires(&info->lock)
```

```
4.  __releases(&info->lock)
5.  {
6.      int fbidx = iminor(file->f_path.dentry->d_inode);
7.      struct fb_info *info = registered_fb[fbidx];
8.      struct fb_ops *fb = info->fbops;
9.      unsigned long off;
10.     unsigned long start;
11.     u32 len;
12.
13.     if (vma->vm_pgoff > (~0UL >> PAGE_SHIFT))
14.         return -EINVAL;
15.     /*
16.      *vma->vm_pgoff 是 vma 区域在文件中的偏移量，即图中的 off,左移 PAGE_SHIFT 是把以页为单位转为以
    字节为单位
17.      *若 PAGE_SIZE 为 4KB，则 PAGE_SHIFT 为 12，因为 PAGE_SIZE 等于 1<<PAGE_SHIFT。
18.      */
19.     off = vma->vm_pgoff << PAGE_SHIFT;
20.     if (!fb)
21.         return -ENODEV;
22.     /*如果具体设备驱动中实现了 mmap，则调用具体设备驱动中的 mmap，否则使用 fbmem.c 中的*/
23.     if (fb->fb_mmap) {
24.         int res;
25.         mutex_lock(&info->lock);
26.         res = fb->fb_mmap(info, vma);
27.         mutex_unlock(&info->lock);
28.         return res;
29.     }
30.
31.     mutex_lock(&info->lock);
32.
33.     /*显存的物理地址*/
34.     start = info->fix.smem_start;
35.     /*求出帧缓冲的长度*/
36.     len = PAGE_ALIGN((start & ~PAGE_MASK) + info->fix.smem_len);
37.     if (off >= len) {
38.         /* memory mapped io */
39.         off -= len;
40.         if (info->var.accel_flags) {
41.             mutex_unlock(&info->lock);
42.             return -EINVAL;
43.         }
44.         /*内存映射 I/O 的开始位置*/
45.         start = info->fix.mmio_start;
46.         /*内存映射 I/O 的长度*/
47.         len = PAGE_ALIGN((start & ~PAGE_MASK) + info->fix.mmio_len);
```

```
48.     }
49.     mutex_unlock(&info->lock);
50.     /*把起始地址页对齐*/
51.     start &= PAGE_MASK;
52.     /*如果区域大于总长度，报错*/
53.     if ((vma->vm_end - vma->vm_start + off) > len)
54.         return -EINVAL;
55.     /*得到在文件中的偏移*/
56.     off += start;
57.     /*把以页为单位转为以字节为单位*/
58.     vma->vm_pgoff = off >> PAGE_SHIFT;
59.     /* This is an IO map - tell maydump to skip this VMA */
60.     /*设置 VMA 标志，将 VMA 设置成一个内存映射的 IO 区域，VM_RESERVED 告诉内存管理系统不要将 VMA
    交换出去*/
61.     vma->vm_flags |= VM_IO | VM_RESERVED;
62.     fb_pgprotect(file, vma, off);
63.     /*真正建立映射的部分，为物理地址和虚拟地址建立页表*/
64.     if (io_remap_pfn_range(vma, vma->vm_start, off >> PAGE_SHIFT,
65.                 vma->vm_end - vma->vm_start, vma->vm_page_prot))
66.         return -EAGAIN;
67.     return 0;
68. }
```