**内核中的readx和writex系列宏的解释**

1.  实际上涉及到的是内核的内存IO区域的读写以及X86平台上IO端口空间的读写，这里只考虑arm平台。内核版本是linux-3.14.1

2.  readx和writex系列宏的定义在不同层次的io.h中。

linux/io.h是最顶层的系统级别的通用级别的io.h，这个io.h又根据当前编译的平台来选择特定的平台目录下的asm/io.h。asm/io.h还会根据具体的板级信息来决定是否包含板级的mach/io.h。所以层次关系很明确：linux/io.h包含arch/arm/include/asm/io.h, arch/arm/include/asm/io.h根据需要包含arch/arm/mach-xxx/include/mach/io.h……所以在编写驱动中，如果需要这3层io.h中的函数，只要引入头文件linux/io.h就OK。

我之前发现GPIO的头文件也是这样的层次化结构，开始不是很习惯，因为linux2.6中并不是这样的层次化头文件。这样的层次化头文件确实很好。

3.  先看最顶层的linux/io.h

实际上，几乎什么也没做，就是引入了arch/arm/include/asm/io.h，声明了devm\_ioremap以及devm\_ioremap\_nocache。

4.  再看arch/arm/include/asm/io.h，这才是核心

A.声明了

\_\_raw\_writesb  \_\_raw\_writesw \_\_raw\_writesl

\_\_raw\_readsb \_\_raw\_readsw \_\_raw\_readsl

这几个函数都是在arch/arm/lib中的汇编文件中定义的，都加了s后缀，就是这些函数会根据参数的from和to进行大小端的强制转换，也就是每次调用这几个函数都会进行大小端转换。也是和\_\_raw\_writex \_\_raw\_readx系列函数的区别。

B.定义了

\_\_raw\_writew \_\_raw\_writeb \_\_raw\_writel

\_\_raw\_readw \_\_raw\_readb \_\_raw\_readl

可以看到，这几个函数都是用arm指令str ldr 系列来实现的。没有考虑大小端问题。

C.关于ioremap方面的定义和声明

D.为了在arm中支持PCI ISA设备的访问，模拟了x86下的IO端口访问，因为，使用的应该是非常少，所以不解释这个。

E．最后，定义了很重要的一些宏

readb\_relaxed readw\_relaxed readl\_relaxed

writeb\_relaxed writew\_relaxed writel\_relaxed

这几个函数使用了raw系列函数，仅仅加入了条件大小端转换，实际上几乎不会进行大小端转换。relaxed系列会把读入前的数据认为是小端，把数据也输出为小端，是不是就是arm系列的任何SOC中外设控制器的寄存器都是采用小端模式，所以才这样处理？如果是这样，那么确实在大小端的arm中都可以正常工作。在arm这样的几乎都使用小端的机器上，转换几乎不会进行，虽然arm支持大端，但很少使用。

F.然后就是经常使用的

#define readb(c)      ({ u8  \_\_v = readb\_relaxed(c); \_\_iormb(); \_\_v; })

#define readw(c)      ({ u16 \_\_v = readw\_relaxed(c); \_\_iormb(); \_\_v; })

#define readl(c)      ({ u32 \_\_v = readl\_relaxed(c); \_\_iormb(); \_\_v; })

#define writeb(v,c)      ({ \_\_iowmb(); writeb\_relaxed(v,c); })

#define writew(v,c)      ({ \_\_iowmb(); writew\_relaxed(v,c); })

#define writel(v,c)      ({ \_\_iowmb(); writel\_relaxed(v,c); })

很显然了，这几个驱动中常用的宏就是relaxed系列添加了内存障碍。内存障碍就是用于确保读入的寄存器数据是实时的或者说是寄存器中的真实数据，而不是缓存的数据。在写入寄存器之前，之前对这个寄存器的写入操作已经完成。