一、 ioremap() 函数基础概念

几乎每一种外设都是通过读写设备上的相关寄存器来进行的，通常包括控制寄存器、状态寄存器和数据寄存器三大类，外设的寄存器通常被连续地编址。根据 CPU 体系结构的不同，CPU 对 IO 端口的编址方式有两种：

a -- I/O 映射方式（I/O-mapped）

典型地，如 X86 处理器为外设专门实现了一个单独的地址空间，称为"I/O 地址空间"或者"I/O 端口空间"，CPU 通过专门的 I/O 指令（如 X86 的 IN 和 OUT 指令）来访问这一空间中的地址单元。

b -- 内存映射方式（Memory-mapped）

RISC 指令系统的 CPU（如 ARM、PowerPC 等）通常只实现一个物理地址空间，外设 I/O 端口成为内存的一部分。此时，CPU 可以象访问一个内存单元那样访问外设 I/O 端口，而不需要设立专门的外设 I/O 指令。

但是，这两者在硬件实现上的差异对于软件来说是完全透明的，驱动程序开发人员可以将内存映射方式的 I/O 端口和外设内存统一看作是"I/O 内存"资源。

一般来说，在系统运行时，外设的 I/O 内存资源的物理地址是已知的，由硬件的设计决定。但是 CPU 通常并没有为这些已知的外设 I/O 内存资源的物理地址预定义虚拟地址范围，驱动程序并不能直接通过物理地址访问 I/O 内存资源，

而必须将它们映射到核心虚地址空间内（通过页表），然后才能根据映射所得到的核心虚地址范围，通过访内指令访问这些 I/O 内存资源。

Linux 在 io.h 头文件中声明了函数 ioremap（），用来将 I/O 内存资源的物理地址映射到核心虚地址空间（3GB－4GB）中（这里是内核空间），原型如下：

1、ioremap 函数

ioremap 宏定义在 asm/io.h 内：

#define ioremap(cookie,size)　　__ioremap(cookie,size,0)

__ioremap 函数原型为(arm/mm/ioremap.c)：

void __iomem * __ioremap(unsigned long phys_addr, size_t size, unsigned long flags);

参数：

phys_addr：要映射的起始的 IO 地址

size：要映射的空间的大小

flags：要映射的 IO 空间和权限有关的标志

该函数返回映射后的内核虚拟地址(3G-4G). 接着便可以通过读写该返回的内核虚拟地址去访问之这段 I/O 内存资源。

2、iounmap 函数

iounmap 函数**用于取消 ioremap（）所做的映射**，原型如下：

void iounmap(void * addr);

二、 ioremap() 相关函数解析

在将 I/O 内存资源的物理地址映射成核心虚地址后，理论上讲我们就可以象读写 RAM 那样直接读写 I/O 内存资源了。为了保证驱动程序的跨平台的可移植性，我们**应该使用 Linux 中特定的函数来访问 I/O 内存资源，而不应该通过指向核心虚地址的指针来访问**。

读写 I/O 的函数如下所示：

a -- writel()

writel()往内存映射的 I/O 空间上**写数据**，wirtel() I/O 上写入 32 位数据 (4 字节)。

原型：**void writel (unsigned char data , unsigned int addr)**

b -- readl()

readl() 从内存映射的 I/O 空间上**读数据**,readl 从 I/O 读取 32 位数据（4 字节）。

原型：**unsigned char readl (unsigned int addr)**

补充说明

在 proc 目录下有 iomem 和 ioports 文件，其主要描述了系统的 io 内存和 io 端口资源分布。

对于外设的访问，最终都是通过读写设备上的寄存器实现的，寄存器不外乎：控制寄存器、状态寄存器和数据寄存器，这些外设寄存器也称为"IO 端口"，并且一个外设的寄存器通常是连续编址的。

不同的 CPU 体系对外设 IO 端口物理地址的编址方式也不同，分为 I/O 映射方式（I/O－mapped）和内存映射方式（Memory－mapped）。

对 X86 熟悉点，以它为例：X86 为外设专门实现有单独的地址空间，可以称为"I/O 地址空间"或"I/O 端口空间"，这个是独立与 CPU 和 RAM 物理地址空间，它将所有外设的 IO 端口均在这一空间进行编址。CPU 通过设立专门的 IN 和 OUT 指令来访问这一空间中的地址单元（即 I/O 端口），这就是所谓的"I/O 映射方式"（I/O－mapped）。和 RAM 物理地址空间相比，I/O 地址空间通常都比较小，如 x86 CPU 的 I/O 空间就只有 64KB（0－0xffff）。这是"I/O 映射方式"的一个主要缺点，你可以通过 **cat /proc/ioports** 去查看，IO port 空间的地址资源分配情况是以树状结构显示。这个源于 x86 平台的设计思想，**目前基本不用了**，获取这些资源的函数接口如 request_region 和 ioremap。

Linux 设计了一个通用的数据结构 resource 来描述各种 I/O 资源（如：I/O 端口、外设内存、DMA 和 IRQ 等）。该结构定义在 include/linux/ioport.h 头文件中。Linux 是以一种倒置的树形结构来管理每一类 I/O 资源。每一类 I/O 资源都对应有一颗倒置的资源树，树中的每一个节点都是个 resource 结构。基于上述这个思想，Linux 将基于 I/O 映射方式的 I/O 端口和基于内存映射方式的 I/O 端口资源统称为"I/O 区域"（I/O Region）。

/proc/iomem 这个文件记录的是物理地址的分配情况，也是以树状结构显示，**对其使用也是 request_mem_region 和 ioremap**，空间大小为 16EB，远大于 io port 的 64K。

ioport 和 iomem 地址空间分别编制，均是从地址 0 开始，如果硬件支持 MMIO，port 地址也可以映射到 memory 空间去。

这里以 pci 设备为例，硬件的拓扑结构就决定了硬件在内存映射到 CPU 的物理地址，由于内存

访问都是虚拟地址，所有就需要 ioremap，此时物理内存是存在的，所以不用再分配内存，只需要做映射即可

应用总结：**使用 I/O 内存首先要申请,然后才能映射**,使用 I/O 端口首先要申请,对 I/O 端口的请求是让内核知道你要访问该端口,内核并让你独占该端口.

申请 I/O 端口的函数是 request_region，申请 I/O 内存的函数是 request_mem_region。request_mem_region 函数并没有做实际性的映射工作，只是告诉内核要使用一块内存地址，声明占有，也方便内核管理这些资源。重要的还是 ioremap 函数，ioremap 主要是检查传入地址的合法性，建立页表（包括访问权限），完成物理地址到虚拟地址的转换。

在 intel 的 X86 平台，GPIO 资源也是类似应用，如果 IO 配置为 SCI 或者 SMI 中断，SCI 可以产生 GPE，然后经历 acpi 子系统，不过 GPE 中断号默认是 0x10+GPIO 端口号。

如果 request_mem_region 和 ioremap 返回失败，那很可能是地址已经被占用，使用 cat /proc/iomem 查看物理地址的占用情况。

树莓派手册中 GPIO 地址为总线地址，物理地址可以通过源码 mach/platform.h 中的 GPIO_BASE 变量得知