**一 概述**

  Linux内核中gpio是最简单，最常用的资源(和 interrupt ,dma,timer一样)驱动程序，应用程序都能够通过相应的接口使用gpio，gpio使用0～MAX\_INT之间的整数标识，不能使用负数,gpio与硬件体系密切相关的,不过linux有一个框架处理gpio，能够使用统一的接口来操作gpio.在讲gpio核心(gpiolib.c)之前先来看看gpio是怎么使用的

**二 内核中gpio的使用**

     1 测试gpio端口是否合法 int gpio\_is\_valid(int number);

     2 申请某个gpio端口当然在申请之前需要显示的配置该gpio端口的pinmux

        int gpio\_request(unsigned gpio, const char \*label)

     3 标记gpio的使用方向包括输入还是输出

       /\*成功返回零失败返回负的错误值\*/

       int gpio\_direction\_input(unsigned gpio);

       int gpio\_direction\_output(unsigned gpio, int value);

     4 获得gpio引脚的值和设置gpio引脚的值(对于输出)

        int gpio\_get\_value(unsigned gpio);

        void gpio\_set\_value(unsigned gpio, int value);

     5 gpio当作中断口使用

        int gpio\_to\_irq(unsigned gpio);

        返回的值即中断编号可以传给request\_irq()和free\_irq()

        内核通过调用该函数将gpio端口转换为中断，在用户空间也有类似方法

     6 导出gpio端口到用户空间

        int gpio\_export(unsigned gpio, bool direction\_may\_change);

        内核可以对已经被gpio\_request()申请的gpio端口的导出进行明确的管理，

        参数direction\_may\_change表示用户程序是否允许修改gpio的方向，假如可以

        则参数direction\_may\_change为真

        /\* 撤销GPIO的导出 \*/

        void gpio\_unexport();

## 三 用户空间gpio的调用

          用户空间访问gpio，即通过sysfs接口访问gpio，下面是/sys/class/gpio目录下的三种文件：

            --export/unexport文件

            --gpioN指代具体的gpio引脚

            --gpio\_chipN指代gpio控制器

            必须知道以上接口没有标准device文件和它们的链接。

### (1) export/unexport文件接口：

               /sys/class/gpio/export，该接口只能写不能读

               用户程序通过写入gpio的编号来向内核申请将某个gpio的控制权导出到用户空间当然前提是没有内核代码申请这个gpio端口

               比如  echo 19 > export

               上述操作会为19号gpio创建一个节点gpio19，此时/sys/class/gpio目录下边生成一个gpio19的目录

               /sys/class/gpio/unexport和导出的效果相反。

               比如 echo 19 > unexport

               上述操作将会移除gpio19这个节点。

### (2) /sys/class/gpio/gpioN

       指代某个具体的gpio端口,里边有如下属性文件

      direction 表示gpio端口的方向，读取结果是in或out。该文件也可以写，写入out 时该gpio设为输出同时电平默认为低。写入low或high则不仅可以

                      设置为输出 还可以设置输出的电平。 当然如果内核不支持或者内核代码不愿意，将不会存在这个属性,比如内核调用了gpio\_export(N,0)就

                       表示内核不愿意修改gpio端口方向属性

      value      表示gpio引脚的电平,0(低电平)1（高电平）,如果gpio被配置为输出，这个值是可写的，记住任何非零的值都将输出高电平, 如果某个引脚

                      能并且已经被配置为中断，则可以调用poll(2)函数监听该中断，中断触发后poll(2)函数就会返回。

      edge      表示中断的触发方式，edge文件有如下四个值："none", "rising", "falling"，"both"。

           none表示引脚为输入，不是中断引脚

           rising表示引脚为中断输入，上升沿触发

           falling表示引脚为中断输入，下降沿触发

           both表示引脚为中断输入，边沿触发

                      这个文件节点只有在引脚被配置为输入引脚的时候才存在。 当值是none时可以通过如下方法将变为中断引脚

                      echo "both" > edge;对于是both,falling还是rising依赖具体硬件的中断的触发方式。此方法即用户态gpio转换为中断引脚的方式

      active\_low 不怎么明白，也木有用过

### (3)/sys/class/gpio/gpiochipN

      gpiochipN表示的就是一个gpio\_chip,用来管理和控制一组gpio端口的控制器，该目录下存在一下属性文件：

      base   和N相同，表示控制器管理的最小的端口编号。

      lable   诊断使用的标志（并不总是唯一的）

      ngpio  表示控制器管理的gpio端口数量（端口范围是：N ~ N+ngpio-1）

## 四 用户态使用gpio监听中断

首先需要将该gpio配置为中断

echo  "rising" > /sys/class/gpio/gpio12/edge

以下是伪代码

int gpio\_id;

struct pollfd fds[1];

gpio\_fd = open("/sys/class/gpio/gpio12/value",O\_RDONLY);

if( gpio\_fd == -1 )

   err\_print("gpio open");

fds[0].fd = gpio\_fd;

fds[0].events  = POLLPRI;

ret = read(gpio\_fd,buff,10);

if( ret == -1 )

    err\_print("read");

while(1){

     ret = poll(fds,1,-1);

     if( ret == -1 )

         err\_print("poll");

       if( fds[0].revents & POLLPRI){

           ret = lseek(gpio\_fd,0,SEEK\_SET);

           if( ret == -1 )

               err\_print("lseek");

           ret = read(gpio\_fd,buff,10);

           if( ret == -1 )

               err\_print("read");

            /\*此时表示已经监听到中断触发了，该干事了\*/

            ...............

    }

}

记住使用poll()函数，设置事件监听类型为POLLPRI和POLLERR在poll()返回后，使用lseek()移动到文件开头读取新的值或者关闭它再重新打开读取新值。必须这样做否则poll函数会总是返回。