第12章 APP 怎么读取按键值

在做单片机开发时，要读取 GPIO 按键，我们通常是执行一个循环，不断地检测 GPIO 引脚电平有没有发生变化。但是在 Linux 系统中，读取 GPIO 按键要考虑到效率，引入了很多种方法：查询方式(非阻塞)、休眠-唤醒(阻塞方式)、poll 方式、异步通知方式。这 4 种方法并不仅仅用于 GPIO 按键，在所有的 APP 调用驱动程序过程中，都是使用这些方法。通过这 4 种方式的学习，我们可以掌握如下知识：

① 驱动的基本技能：中断、休眠、唤醒、poll 等机制。

这些基本技能是驱动开发的基础，其他大型驱动复杂的地方是它的框架及设计思想，但是基本技术就这些。

② APP 开发的基本技能：阻塞 、非阻塞、休眠、poll、异步通知。

12.1 妈妈怎么知道孩子醒了

图 12.1 示例

妈妈怎么知道卧室里小孩醒了？

① 时不时进房间看一下：查询方式
- 简单，但是累

② 进去房间陪小孩一起睡觉，小孩醒了会吵醒她：休眠-唤醒
- 不累，但是妈妈干不了活了

③ 妈妈要干很多活，但是可以陪小孩睡一会，定个闹钟：poll 方式
- 要浪费点时间，但是可以继续干活。
- 妈妈要么是被小孩吵醒，要么是被闹钟吵醒。

④ 妈妈在客厅干活，小孩醒了他会自己走出房门告诉妈妈：异步通知

■ 妈妈、小孩互不耽误。

这 4 种方法没有优劣之分，在不同的场合使用不同的方法。

12.2 APP 读取按键的 4 种方法

APP 去读取按键和举例的场景很相似，也有 4 种方法：

① 查询方式

② 休眠-唤醒方式

③ poll 方式

④ 异步通知方式

第 2、3、4 种方法，都涉及中断服务程序。中断，就像小孩醒了会哭闹一样，中断不经意间到来，它会做某些事情：唤醒 APP、向 APP 发信号。

所以，在按键驱动程序中，中断是核心。

实际上，中断无论是在单片机还是在 Linux 中都很重要。在 Linux 中，中断的知识还涉及进程、线程等。

12.2.1 查询方式

这种方法最简单：

图 12.2 查询方式

驱动程序中构造、注册一个 file_operations 结构体，里面提供有对应的 open,read 函数。APP 调用 open 时，导致驱动中对应的 open 函数被调用，在里面配置 GPIO 为输入引脚。APP 调用 read 时，导致驱动中对应的 read 函数被调用，它读取寄存器，把引脚状态直接返回给 APP。

12.2.2 休眠-唤醒方式

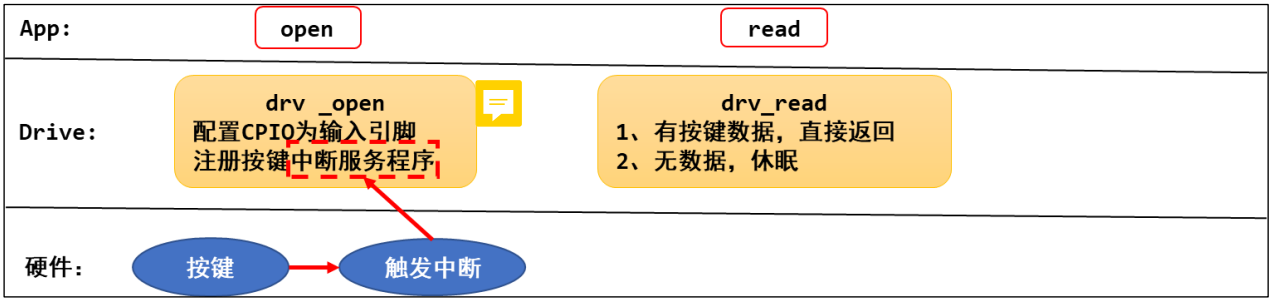

图 12.3 休眠唤醒方式

驱动程序中构造、注册一个 file_operations 结构体，里面提供有对应的 open,read 函数。

- APP 调用 open 时，导致驱动中对应的 open 函数被调用，在里面配置 GPIO 为输入引脚；并且注册 GPIO 的中断处理函数。
- APP 调用 read 时，导致驱动中对应的 read 函数被调用，如果有按键数据则直接返回给 APP；否则 APP 在内核态休眠。

当用户按下按键时，GPIO 中断被触发，导致驱动程序之前注册的中断服务程序被执行。它会记录按键数据，并唤醒休眠中的 APP。

APP 被唤醒后继续在内核态运行，即继续执行驱动代码，把按键数据返回给 APP(的用户空间)。

12.2.3 poll 方式

上面的休眠-唤醒方式有个缺点：如果用户一直没操作按键，那么 APP 就会永远休眠。

我们可以给 APP 定个闹钟，这就是 poll 方式。

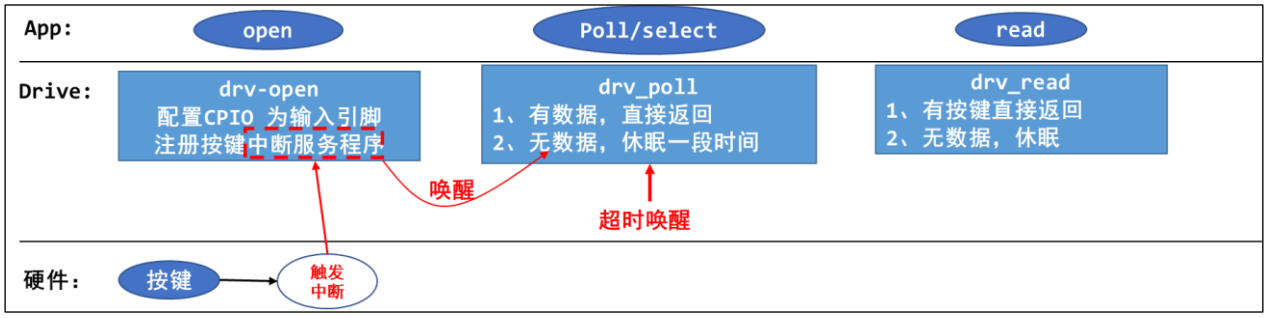

图 12.4 poll 方式

驱动程序中构造、注册一个 file_operations 结构体，里面提供有对应的 open,read,poll 函数。

- APP 调用 open 时，导致驱动中对应的 open 函数被调用，在里面配置 GPIO 为输入引脚；并且注册 GPIO 的中断处理函数。
- APP 调用 poll 或 select 函数，意图是“查询”是否有数据，这 2 个函数都可以指定一个超时时间，即在这段时间内没有数据的话就返回错误。这会导致驱动中对应的 poll 函数被调用，如果有按键数据则直接返回给 APP；否则 APP 在内核态休眠一段时间。

当用户按下按键时，GPIO 中断被触发，导致驱动程序之前注册的中断服务程序被执行。它会记录按键数据，并唤醒休眠中的 APP。

如果用户没按下按键，但是超时时间到了，内核也会唤醒 APP。

所以 APP 被唤醒有 2 种原因：用户操作了按键，超时。被唤醒的 APP 在内核态继续运行，即继续执行驱动代码，把“状态”返回给 APP(的用户空间)。

APP 得到 poll/select 函数的返回结果后，如果确认是有数据的，则再调用 read 函数，这会导致驱动中的 read 函数被调用，这时驱动程序中含有数据，

会直接返回数据。

12.2.4 异步通知方式

1 异步通知的原理：发信号

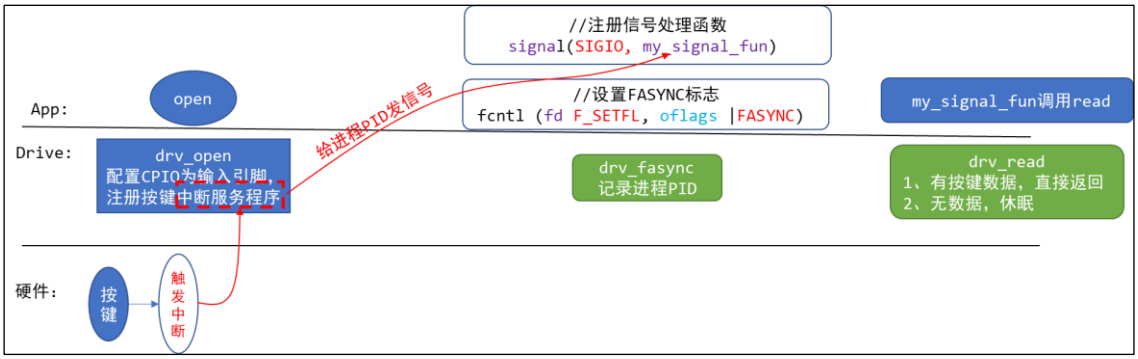

图 12.5 异步通知的原理

异步通知的实现原理是：内核给 APP 发信号。信号有很多种，这里发的是 SIGIO。

驱动程序中构造、注册一个 file_operations 结构体，里面提供有对应的 open,read,fasync 函数。

- APP 调用 open 时，导致驱动中对应的 open 函数被调用，在里面配置 GPIO 为输入引脚；并且注册 GPIO 的中断处理函数。
- APP 给信号 SIGIO 注册自己的处理函数：my_signal_fun。
- APP 调用 fcntl 函数，把驱动程序的 flag 改为 FASYNC，这会导致驱动程序的 fasync 函数被调用，它只是简单记录进程 PID。
- 当用户按下按键时，GPIO 中断被触发，导致驱动程序之前注册的中断服务程序被执行。它会记录按键数据，然后给进程 PID 发送 SIGIO 信号。
- APP 收到信号后会被打断，先执行信号处理函数：在信号处理函数中可以去调用 read 函数读取按键值。
- 信号处理函数返回后，APP 会继续执行原先被打断的代码。

2 应用程序之间发信号示例代码

使用 GIT 下载所有源码后，本节源码位于如下目录：

```
01_all_series_quickstart\
05_嵌入式 Linux 驱动开发基础知识\source\03_signal_example
```

代码并不复杂，如下。

第 13 行注册信号处理函数，第 15 行就是一个无限循环。在它运行期间，你可以用另一个 APP 发信号给它。

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <signal.h>
04 void my_sig_func(int signo)
```

```
05 {
06     printf("get a signal : %d\n", signo);
07 }
08
09 int main(int argc, char **argv)
10 {
11     int i = 0;
12
13     signal(SIGIO, my_sig_func);
14
15     while (1)
16     {
17         printf("Hello, world %d!\n", i++);
18         sleep(2);
19     }
20
21     return 0;
22 }
```

在 Ubuntu 上的测试方法：

```
$ gcc   -o   signal  signal.c   // 编译程序
$ ./signal &                   // 后台运行
$ ps  -A | grep signal          // 查看进程 ID，假设是 9527
$ kill  -SIGIO  9527           // 给这个进程发信号
```

12.2.5 驱动程序提供能力，不提供策略

我们的驱动程序可以实现上述 4 种提供按键的方法，但是驱动程序不应该限制 APP 使用哪种方法。

这就是驱动设计的一个原理：提供能力，不提供策略。就是说，你想用哪种方法都行，驱动程序都可以提供；但是驱动程序不能限制你使用哪种方法。