Linux 内核文档

GPIO 接口

原文：Documentation/gpio.txt
翻译：tekkamanninja@gmail.com
翻译完成时间：2012 年 2 月 18 日星期六

V1.0

目录

本文档提供了一个在 Linux 下访问 GPIO 的公约概述。
这些函数以 gpio_* 作为前缀。其他的函数不允许使用这样的前缀或相关的 __gpio_* 前缀。

什么是 GPIO?

"通用输入/输出口"(GPIO)是一个灵活的由软件控制的数字信号。他们可由多种芯片提供，且对于从事嵌入式和定制硬件的 Linux 开发者来说是比较熟悉。每个 GPIO 都代表一个连接到特定引脚或球栅阵列(BGA)封装中“球珠”的一个位。电路板原理图显示了 GPIO 与外部硬件的连接关系。驱动可以编写成通用代码，以使板级启动代码可传递引脚配置数据给驱动。

片上系统 (SOC) 处理器对 GPIO 有很大的依赖。在某些情况下，每个非专用引脚都可配置为 GPIO，且大多数芯片都最少有一些 GPIO。可编程逻辑器件(类似 FPGA) 可以方便地提供 GPIO。像电源管理和音频编解码器这样的多功能芯片经常留有一些这样的引脚来帮助那些引脚匮乏的 SOC。同时还有通过 I2C 或 SPI 串行总线连接的"GPIO 扩展器"芯片。大多数 PC 的南桥有一些拥有 GPIO 能力的引脚 (只有 BIOS 固件才知道如何使用他们)。

GPIO 的实际功能因系统而异。通常的用法有：

- 输出值可写 (高电平=1, 低电平=0)。一些芯片也有如何驱动这些值的选项，例如只允许输出一个值、支持“线与”及其他取值类似的模式（值得注意的是“开漏”信号）。

- 输入值可读(1、0)。一些芯片支持引脚在配置为“输出”时回读，这对于类似“线与”的情况（以支持双向信号）是非常有用的。GPIO 控制器可能有输入去毛刺/消抖逻辑，这有时需要软件控制。

- 输入通常可作为 IRQ 信号，一般是沿触发，但有时是电平触发。这样的 IRQ 可能配置为系统唤醒事件，以将系统从低功耗状态下唤醒。

- 通常一个 GPIO 根据不同产品电路板的需求，可以配置为输入或输出，也有仅支持单向的。

- 大部分 GPIO 可以在持有自旋锁时访问，但是通常由串行总线扩展的 GPIO 不允许持有自旋锁。但某些系统也支持这种类型。

对于给定的电路板，每个 GPIO 都用于某个特定的目的，如监控 MMC/SD 卡的插入/移除、检测卡的写保护状态、驱动 LED、配置收发器、模拟串行总线、复位硬件看门狗、感知开关状态等等。

GPIO 公约

注意，这个叫做“公约”，因为这不是强制性的，不遵循这个公约是无伤大雅的，因为此时可移植性并不重要。GPIO 常用于板级特定的电路逻辑，甚至可能随着电路板的版本而改变，且不可能在不同走线的板子上使用。仅有在很少的功能上才具有可移植性，其他功能是平台特定。这也是由于“胶合”的逻辑造成的。

此外，这不需要任何的执行框架，只是一个接口。某个平台可能通过一个简单的访问芯片寄存器的内联函数来实现它，其他平台可能通过委托一系列不同的 GPIO 控制器的抽象函数来实现它。(有一些可选的代码能支持这种策略的实现，本文档后面会介绍，但作为 GPIO 接口的客户端驱动程序必须与它的实现无关。)

也就是说，如果在他们的平台上支持这个公约，驱动应该尽可能的使用它。平台必须在 Kconfig 中声明对 GENERIC_GPIO 的支持 (布尔型 true)，并提供一个 <asm/gpio.h> 文件。那些调用标准 GPIO 函数的驱动应该在 Kconfig 入口中声明依赖 GENERIC_GPIO。当驱动包含文件：

```
#include <linux/gpio.h>
```

GPIO 函数是可用，无论是“真实代码”还是经优化过的语句。

如果你遵守这个公约，当你的代码完成后，对其他的开发者来说会更容易看懂和维护。

注意，这些操作包含所用平台的 I/O 屏障代码，驱动无须显式地调用他们。

标识 GPIO

GPIO 是通过无符号整型来标识的，范围是 0 到 MAX_INT。保留“负”数用于其他目的，例如标识信号“在这个板子上不可用”或指示错误。未接触底层硬件的代码会忽略这些整数。

平台会定义这些整数的用法，且通常使用 #define 来定义 GPIO ，这样板级特定的启动代码可以直接关联相应的原理图。相对来说，驱动应该仅使用启动代码传递过来的 GPIO 编号，使用 platform_data 保存板级特定引脚配置数据 (同时还有其他须要的板级特定数据)，避免可能出现的问题。

例如一个平台使用编号 32-159 来标识 GPIO，而在另一个平台使用编号 0-63 标识一组 GPIO 控制器，64-79 标识另一类 GPIO 控制器，且在一个含有 FPGA 的特定板子上使用 80-95 。编号不一定要连续，那些平台中，也可以使用编号 2000-2063 来标识一个 I2C 接口的 GPIO 扩展器中的 GPIO。

如果你要初始化一个带有无效 GPIO 编号的结构体，可以使用一些负编码（比如"-EINVAL"），那将永远不会是有效。来测试这样一个结构体中的编号是否关联一个 GPIO，你可使用以下断言：

```
int gpio_is_valid(int number);
```

如果编号不存在，则请求和释放 GPIO 的函数将拒绝执行相关操作（见下文）。其他编号也可能被拒绝，比如一个编号可能存在，但暂时在给定的板子上不可用。

一个平台是否支持多个 GPIO 控制器是平台特定的实现问题，就像是否可以在 GPIO 编号空间中有“空洞”和是否可以在运行时添加新的控制器一样。这些问题会影响其他事情，包括相邻的 GPIO 编号是否存在等。

使用 GPIO

对于一个 GPIO，系统应该做的第一件事情就是通过 gpio_request() 函数分配他，见下文。

而接下来要做的是标识它的方向，这通常是在板级启动代码中为 GPIO 设置一个 platform_device 时做的。

```
/* 设置为输入或输出，返回 0 或负的错误代码 */
int gpio_direction_input(unsigned gpio);
int gpio_direction_output(unsigned gpio, int value);
```

返回值为零代表成功，否则返回一个负的错误代码。这个返回值需要检查，因为 get/set（获取/设置）函数调用没法返回错误，且有可能是配置错误。通常，你应该在进程上下文中调用这些函数。然而，对于自旋锁安全的 GPIO，在板子启动的早期、进程启动前使用他们也是可以的。

对于作为输出的 GPIO，为其提供初始输出值，对于避免在系统启动期间出现信号毛刺是很有帮助的。

为了与传统的 GPIO 接口兼容，在设置一个 GPIO 方向时，如果它还未被申请，则隐含了申请那个 GPIO 的操作(见下文)。这种兼容性正在从可选的 gpiolib 框架中移除。

如果这个 GPIO 编码不存在或特定的 GPIO 不能用于那种模式，则方向设置可能失败。依赖启动固件来正确地设置方向通常是一个坏主意，因为它可能除了启动 Linux，并没有做更多的验证工作。(类似地，板子的启动代码可能需要将这个复用的引脚设置为 GPIO，并正确地配置上拉/下拉电阻。)

访问自旋锁安全的 GPIO

大多数 GPIO 控制器可以通过内存读/写指令来访问。这些指令不会休眠，可以安全地在硬（非线程）中断例程和类似的上下文中完成。

对于那些用 gpio_cansleep()测试总是返回失败的 GPIO（见下文），使用以下的函数访问：

```
/* GPIO 输入：返回零或非零 */
int gpio_get_value(unsigned gpio);

/* GPIO 输入 */
void gpio_set_value(unsigned gpio, int value);
```

返回值是布尔值，零表示低电平，非零表示高电平。当读取一个输出引脚的值时，返回值应该是引脚上的值。这个值不总是和输出值相符，因为存在开漏输出信号和输出潜伏期的问题。

以上的 get/set 函数不会对早期已经通过 gpio_direction_*()报告“无效的 GPIO”返回错误。此外，还需要注意的是并不是所有平台都可以从输出引脚中读取数据的，那些引脚也不总是返回零。且对那些无法安全访问（可能会休眠）的 GPIO（见下文）使用这些函数是错误的。

在 GPIO 编号（还有输出、值）为常数的情况下，鼓励通过平台特定的实现来优化这两个函数来访问 GPIO 值。这种情况（读写一个硬件寄存器）下只需要几条指令是很正常的，且无须自旋锁。这种优化函数比起那些在子程序上花费许多指令的函数可以使得模拟接口（译者注：例如 GPIO 模拟 I2C、1-wire 或 SPI）的应用（在空间和时间上都）更具效率。

访问可能休眠的 GPIO

某些 GPIO 控制器必须通过基于总线（如 I2C 或 SPI）的消息访问。读或写这些 GPIO 值的命令需要等待其信息排到队首才发送命令，再获得其反馈。期间需要休眠，这不能在 IRQ 例程（中断上下文）中执行。

支持此类 GPIO 的平台通过以下函数返回非零值来区分出这种 GPIO。(此函数需要一个之前通过 gpio_request 分配到的有效的 GPIO 编号)：

```
int gpio_cansleep(unsigned gpio);
```

为了访问这种 GPIO，内核定义了一套不同的函数：

```
/* GPIO 输入：返回零或非零 ，可能会休眠 */
int gpio_get_value_cansleep(unsigned gpio);

/* GPIO 输入，可能会休眠 */
void gpio_set_value_cansleep(unsigned gpio, int value);
```

访问这样的 GPIO 需要一个允许休眠的上下文，例如线程 IRQ 处理例程，并用以上的访问函数替换那些没有 cansleep()后缀的自旋锁安全访问函数。

除了这些访问函数可能休眠，且它们操作的 GPIO 不能在硬件 IRQ 处理例程中访问的事实，这些处理例程实际上和自旋锁安全的函数是一样的。

** 除此之外 ** 调用设置和配置此类 GPIO 的函数也必须在允许休眠的上下文中，因为它们可能也需要访问 GPIO 控制器芯片： (这些设置函数通常在板级启动代码或者驱动探测/断开代码中，所以这是一个容易满足的约束条件。)

```
gpio_direction_input()
gpio_direction_output()
gpio_request()

## gpio_request_one()
## gpio_request_array()
## gpio_free_array()

gpio_free()
gpio_set_debounce()
```

声明和释放 GPIO

为了有助于捕获系统配置错误，定义了两个函数。

```
/* 申请 GPIO, 返回 0 或负的错误代码.
* 非空标签可能有助于诊断.
*/
int gpio_request(unsigned gpio, const char *label);

/* 释放之前声明的 GPIO */
void gpio_free(unsigned gpio);
```

将无效的 GPIO 编码传递给 gpio_request()会导致失败，申请一个已使用这个函数声明过的 GPIO 也会失败。gpio_request()的返回值必须检查。你应该在进程上下文中调用这些函数。然而，对于自旋锁安全的 GPIO，在板子启动的早期、进入进程之前是可以申请的。

这个函数完成两个基本的目标。一是标识那些实际上已作为 GPIO 使用的信号线，这样便于更好地诊断；系统可能需要服务几百个潜在的 GPIO，但是对于任何一个给定的电路板通常只有一些被使用。另一个目的是捕获冲突，查明错误：如两个或更多驱动错误地认为他们已经独占了某个信号线，或是错误地认为移除一个管理着某个已激活信号的驱动是安全的。也就是说，申请 GPIO 的作用类似一种锁机制。

某些平台可能也使用 GPIO 作为电源管理（例如通过关闭未使用芯片区和简单地关闭未使用时钟）激活信号。

注意：申请一个 GPIO 并没有以任何方式配置它，只不过标识那个 GPIO 处于使用状态。必须有另外的代码来处理引脚配置（如控制 GPIO 使用的引脚、上拉/下拉）。

并且注意在释放 GPIO 前，你必须停止使用它。

考虑到大多数情况下声明 GPIO 之后就会立即配置它们，所以定义了以下三个辅助函数：

```
/* 申请一个 GPIO 信号，同时通过特定的'flags'初始化配置，
* 其他和 gpio_request()的参数和返回值相同
*
*/
int gpio_request_one(unsigned gpio, unsigned long flags, const char *label);

/* 在单个函数中申请多个 GPIO
*/
int gpio_request_array(struct gpio *array, size_t num);

/* 在单个函数中释放多个 GPIO
*/
void gpio_free_array(struct gpio *array, size_t num);
```

这里 'flags' 当前定义可指定以下属性：

```
* GPIOF_DIR_IN - 配置方向为输入
* GPIOF_DIR_OUT - 配置方向为输出

* GPIOF_INIT_LOW - 在作为输出时，初始值为低电平
* GPIOF_INIT_HIGH - 在作为输出时，初始值为高电平
```

因为 GPIOF_INIT_* 仅有在配置为输出的时候才存在，所以有效的组合为：

```
* GPIOF_IN - 配置为输入
* GPIOF_OUT_INIT_LOW - 配置为输出，并初始化为低电平
* GPIOF_OUT_INIT_HIGH - 配置为输出，并初始化为高电平
```

将来这些标志可能扩展到支持更多的属性，比如开漏状态。

更进一步，为了更简单地声明/释放多个 GPIO，'struct gpio'被引进来封装所有这三个领域：

```
struct gpio {
        unsigned gpio;
        unsigned long flags;
        const char *label;
};
```

一个典型的使用案例:

```
static struct gpio leds_gpios[] = {
            { 32, GPIOF_OUT_INIT_HIGH, "Power LED" },       /* default to ON */
            { 33, GPIOF_OUT_INIT_LOW, "Green LED" },        /* default to OFF */
            { 34, GPIOF_OUT_INIT_LOW, "Red LED" },          /* default to OFF */
            { 35, GPIOF_OUT_INIT_LOW, "Blue LED" },         /* default to OFF */
            { ... },
};

err = gpio_request_one(31, GPIOF_IN, "Reset Button");
if (err)
...

err = gpio_request_array(leds_gpios, ARRAY_SIZE(leds_gpios));
```

```
    if (err)
    ...

    gpio_free_array(leds_gpios, ARRAY_SIZE(leds_gpios));
```

GPIO 映射到 IRQ

GPIO 编号是无符号整数；IRQ 编号也是。这些构成了两个逻辑上不同的命名空间（GPIO 0 不一定使用 IRQ 0）。你可以通过以下函数在它们之间实现映射：

```
    /* 映射 GPIO 编号到 IRQ 编号 */
    int gpio_to_irq(unsigned gpio);

    /* 映射 IRQ 编号到 GPIO 编号 (尽量避免使用) */
    int irq_to_gpio(unsigned irq);
```

它们的返回值为对应命名空间的相关编号，或是负的错误代码（如果无法映射）。(例如，某些 GPIO 无法做为 IRQ 使用。)以下的编号错误是未经检测的：使用一个未通过 gpio_direction_input()配置为输入的 GPIO 编号，或者使用一个并非来源于 gpio_to_irq()的 IRQ 编号。

这两个映射函数可能会在信号编号的加减计算过程上花些时间。它们不可休眠。

gpio_to_irq()返回的非错误值可以传递给 request_irq()或者 free_irq()。它们通常通过板级特定的初始化代码存放到平台设备的 IRQ 资源中。注意：IRQ 触发选项是 IRQ 接口的一部分，比如 IRQF_TRIGGER_FALLING，系统唤醒能力也是如此。

irq_to_gpio()返回的非错误值大多数通常可以被 gpio_get_value()所使用，比如在 IRQ 是沿触发时初始化或更新驱动状态。注意某些平台不支持反映射，所以你应该尽量避免使用它。

模拟开漏信号

有时在只有低电平信号作为实际驱动结果（译者注：多个输出连接于一点，逻辑电平结果为所有输出的逻辑与）的时候，共享的信号线需要使用“开漏”信号。（该术语适用于 CMOS 管;而 TTL 用“集电极开路”。）一个上拉电阻使信号为高电平。这有时被称为“线与”。实际上，从负逻辑（低电平为真）的角度来看，这是一个“线或”。

一个开漏信号的常见例子是共享的低电平使能 IRQ 信号线。此外，有时双向数据总线信号也使用漏极开路信号。

某些 GPIO 控制器直接支持开漏输出，还有许多不支持。当你需要开漏信号但你的硬件又不直接支持的时候，一个常用的方法是用任何即可作输入也可作输出的 GPIO 引脚来模拟：

```
    LOW: gpio_direction_output(gpio, 0) ... 这代码驱动信号并覆盖上拉配置
    HIGH: gpio_direction_input(gpio) ... 这代码关闭输出，所以上拉电阻（或其他的一些器件）控制了信号。
```

如果你将信号线“驱动”为高电平，但是 gpio_get_value(gpio)报告了一个低电平（在适当的上升时间后），你就可以知道是其他的一些组件将共享信号线拉低了。这不一定是错误的。一个常见的例子就是 I2C 时钟的延长：一个需要较慢时钟的从设备延迟 SCK 的上升沿，而 I2C 主设备相应地调整其信号传输速率。

这些公约忽略了什么？

这些公约忽略的最大一件事就是引脚复用，因为这属于高度芯片特定的属性且没有可移植性。某个平台可能不需要明确的复用信息；有的对于任意给定的引脚可能只有两个功能选项；有的可能每个引脚有八个功能选项；有的可能可以将几个引脚中的任何一个作为给定的 GPIO。(是的，这些例子都来自于当前运行 Linux 的系统。)

在某些系统中，与引脚复用相关的是配置和使能集成的上、下拉模式。并不是所有平台都支持这种模式，或者不会以相同的方式来支持这种模式；且任何给定的电路板可能使用外置的上拉（或下拉）电阻，这时芯片上的就不应该使用。（当一个电路需要 5kOhm 的拉动电阻，芯片上的 100 kOhm 电阻就不能做到。）同样的，驱动能力(2 mA vs 20 mA)和电压(1.8V vs 3.3V)是平台特定问题，就像模型一样在可配置引脚和 GPIO 之间（没）有一一对应的关系。

还有其他一些系统特定的机制没有在这里指出，例如上述的输入去毛刺和线与输出选项。硬件可能支持批量读或写 GPIO，但是那一般是配置相关的：对于处于同一块区（bank）的 GPIO。(GPIO 通常以 16 或 32 个组成一个区块，一个给定的片上系统

一般有几个这样的区块。)某些系统可以通过输出 GPIO 触发 IRQ，或者从并非以 GPIO 管理的引脚取值。这些机制的相关代码没有必要具有可移植性。

当前，动态定义 GPIO 并不是标准的，例如作为配置一个带有某些 GPIO 扩展器的附加电路板的副作用。

GPIO 实现者的框架（可选）

前面提到了，有一个可选的实现框架，让平台使用相同的编程接口，更加简单地支持不同种类的 GPIO 控制器。这个框架称为"gpiolib"。

作为一个辅助调试功能，如果 debugfs 可用，就会有一个 /sys/kernel/debug/gpio 文件。通过这个框架，它可以列出所有注册的控制器，以及当前正在使用中的 GPIO 的状态。

控制器驱动：gpio_chip

在框架中每个 GPIO 控制器都包装为一个 "struct gpio_chip" ，他包含了该类型的每个控制器的常用信息：

- 设置 GPIO 方向的方法

- 用于访问 GPIO 值的方法

- 告知调用其方法是否可能休眠的标志

- 可选的 debugfs 信息导出方法 (显示类似上拉配置一样的额外状态)

- 诊断标签

也包含了来自 device.platform_data 的每个实例的数据：它第一个 GPIO 的编号和它可用的 GPIO 的数量。

实现 gpio_chip 的代码应该支持多控制器实例，可能使用驱动模型。那些代码要配置每个 gpio_chip，并发起 gpiochip_add()。卸载一个 GPIO 控制器很少见，但在必要的时候可以使用 gpiochip_remove()。

大部分 gpio_chip 是一个实例特定结构体的一部分，而并不将 GPIO 接口单独暴露出来，比如编址、电源管理等。类似编解码器这样的芯片会有复杂的非 GPIO 状态。

任何一个 debugfs 信息导出方法通常应该忽略还未申请作为 GPIO 的信号线。他们可以使用 gpiochip_is_requested()测试，当这个 GPIO 已经申请过了就返回相关的标签，否则返回 NULL。

平台支持

为了支持这个框架，一个平台的 Kconfig 文件将会 "select"（选择） ARCH_REQUIRE_GPIOLIB 或 ARCH_WANT_OPTIONAL_GPIOLIB ，并让它的 <asm/gpio.h> 包含 <asm-generic/gpio.h> ，并定义三个方法：gpio_get_value()、gpio_set_value()和 gpio_cansleep()。

它也应该提供一个 ARCH_NR_GPIOS 的定义值，这样可以更好地反映该平台 GPIO 的实际数量，节省静态表的空间。（这个定义值应该包含片上系统内建 GPIO 和 GPIO 扩展器中的数据。）

ARCH_REQUIRE_GPIOLIB 意味着 gpiolib 核心在这个构架中将总是编译进内核。

ARCH_WANT_OPTIONAL_GPIOLIB 意味着 gpiolib 核心默认关闭，且用户可以使能它，并将其编译进内核（可选）。

如果这些选项都没被选择，该平台就不通过 GPIO-lib 支持 GPIO，且代码不可以被用户使能。

以下这些方法的实现可以直接使用框架代码，并总是通过 gpio_chip 调度：

```
#define gpio_get_value __gpio_get_value
#define gpio_set_value __gpio_set_value
#define gpio_cansleep __gpio_cansleep
```

这些定义可以用更理想的实现方法替代，那就是使用经过逻辑优化的内联函数来访问基于特定片上系统的 GPIO。例如，若引用 GPIO 的（寄存器地址）是常量“12”，读取或设置它可能只需少则两或三个指令，且不会休眠。当这样的优化无法实现时，那些函数必须使用框架提供的代码，那就至少要几十条指令才可以实现。对于用 GPIO 模拟的 I/O 接口，如此精简指令是很有意义的。

对于片上系统，平台特定代码为片上 GPIO 每个区（bank）定义并注册 gpio_chip 实例。那些 GPIO 应该根据芯片厂商的文档进行编码/标签，并直接和电路板原理图对应。他们应该开始于零并终止于平台特定的限制。这些 GPIO（代码）通常从 arch_initcall()或者更早的地方集成进平台初始化代码，使这些 GPIO 总是可用，且他们通常可以作为 IRQ 使用。

板级支持

对于外部 GPIO 控制器（例如 I2C 或 SPI 扩展器、专用芯片、多功能器件、FPGA 或 CPLD），大多数常用板级特定代码都可以注册控制器设备，并保证他们的驱动知道 gpiochip_add()所使用的 GPIO 编号。他们的起始编号通常跟在平台特定的 GPIO 编号之后。

例如板级启动代码应该创建结构体指明芯片公开的 GPIO 范围，并使用 platform_data 将其传递给每个 GPIO 扩展器芯片。然后芯片驱动中的 probe()例程可以将这个数据传递给 gpiochip_add()。

初始化顺序很重要。例如，如果一个设备依赖基于 I2C 的（扩展）GPIO，那么它的 probe()例程就应该在那个 GPIO 有效以后才可以被调用。这意味着设备应该在 GPIO 可以工作之后才可被注册。解决这类依赖的的一种方法是让这种 gpio_chip 控制器向板级特定代码提供 setup()和 teardown()回调函数。一旦所有必须的资源可用之后，这些板级特定的回调函数将会注册设备，并可以在这些 GPIO 控制器设备变成无效时移除它们。

用户空间的 Sysfs 接口(可选)

使用"gpiolib"实现框架的平台可以选择配置一个 GPIO 的 sysfs 用户接口。这不同于 debugfs 接口，因为它提供的是对 GPIO 方向和值的控制，而不只显示一个 GPIO 的状态摘要。此外，它可以出现在没有调试支持的产品级系统中。

例如，通过适当的系统硬件文档，用户空间可以知道 GIOP #23 控制 Flash 存储器的写保护（用于保护其中 Bootloader 分区）。产品的系统升级可能需要临时解除这个保护：首先导入一个 GPIO，改变其输出状态，然后在重新使能写保护前升级代码。通常情况下，GPIO #23 是不会被触及的，并且内核也不需要知道他。

根据适当的硬件文档，某些系统的用户空间 GPIO 可以用于确定系统配置数据，这些数据是标准内核不知道的。在某些任务中，简单的用户空间 GPIO 驱动可能是系统真正需要的。

注意：标准内核驱动中已经存在通用的“LED 和按键”GPIO 任务，分别是："leds-gpio" 和 "gpio_keys"。请使用这些来替代直接访问 GPIO，因为集成在内核框架中的这类驱动比你在用户空间的代码更好。

Sysfs 中的路径

在/sys/class/gpio 中有 3 类入口：

- 用于在用户空间控制 GPIO 的控制接口；

- GPIOs 本身；以及

- GPIO 控制器 ("gpio_chip" 实例)。

除了这些标准的文件，还包含“device”符号链接。

控制接口是只写的：

/sys/class/gpio/

"export" ... 用户空间可以通过写其编号到这个文件，要求内核导出一个 GPIO 的控制到用户空间。

例如: 如果内核代码没有申请 GPIO #19，"echo 19 > export" 将会为 GPIO #19 创建一个 "gpio19" 节点。

"unexport" ... 导出到用户空间的逆操作。

例如: "echo 19 > unexport" 将会移除使用"export"文件导出的 "gpio19" 节点。

GPIO 信号的路径类似 /sys/class/gpio/gpio42/ (对于 GPIO #42 来说)，并有如下的读/写属性：

/sys/class/gpio/gpioN/

"direction" ... 读取得到 "in" 或 "out"。这个值通常运行写入。写入"out" 时，其引脚的默认输出为低电平。为了确保无故障运行，"low" 或 "high" 的电平值应该写入 GPIO 的配置，作为初始输出值。

注意：如果内核不支持改变 GPIO 的方向，或者在导出时内核代码没有明确允许用户空间可以重新配置 GPIO 方向，那么这个熟悉将不存在。

"value" ... 读取得到 0 (低电平) 或 1 (高电平)。如果 GPIO 配置为输出，这个值允许写操作。任何非零值都以高电平看待。

如果引脚可以配置为中断信号，且如果已经配置了产生中断的模式（见"edge"的描述），你可以对这个文件使用轮询操作（poll(2)），且轮询操作会在任何中断触发时返回。如果你使用轮询操作（poll(2)），请在 events 中设置 POLLPRI 和 POLLERR。如果你使用轮询操作（select(2)），请在 exceptfds 设置你期望的文件描述符。在轮询操作（poll(2)）返回之后，既可以通过 lseek(2)操作读取 sysfs 文件的开始部分，也可以关闭这个文件并重新打开它来读取数据。

"edge" ... 读取得到 "none"、"rising"、"falling"或者"both"。将这些字符串写入这个文件可以选择沿触发模式，会使得轮询操作（select(2)）在"value"文件中返回。

这个文件仅有在这个引脚可以配置为可产生中断输入引脚时，才存在。

"active_low" ... 读取得到 0 (假) 或 1 (真)。 写入任何非零值可以翻转这个属性的（读写）值。已存在或之后通过"edge"属性设置了"rising" 和 "falling" 沿触发模式的轮询操作（poll(2)）将会遵循这个设置。

GPIO 控制器的路径类似 /sys/class/gpio/gpiochip42/ (对于从#42 GPIO 开始实现控制的控制器)，并有着以下只读属性：

/sys/class/gpio/gpiochipN/

"base" ... 与以上的 N 相同，代表此芯片管理的第一个 GPIO 的编号

"label" ... 用于诊断 (并不总是只有唯一值)

"ngpio" ... 此控制器所管理的 GPIO 数量(而 GPIO 编号从 N 到 N + ngpio - 1)

大多数情况下，电路板的文档应当标明每个 GPIO 的使用目的。但是那些编号并不总是固定的，例如在扩展卡上的 GPIO 会根据所使用的主板或所在堆叠架构中其他的板子而有所不同。在这种情况下，你可能需要使用 gpiochip 节点（尽可能地结合电路图）来确定给定信号所用的 GPIO 编号。

从内核代码中导出

内核代码可以明确地管理那些已通过 gpio_request()申请的 GPIO 的导出：

```
/* 导出 GPIO 到用户空间 */
int gpio_export(unsigned gpio, bool direction_may_change);

/* gpio_export()的逆操作 */
void gpio_unexport();

/* 创建一个 sysfs 连接到已导出的 GPIO 节点 */
```

```
int gpio_export_link(struct device *dev, const char *name,
unsigned gpio)

/* 改变 sysfs 中的一个 GPIO 节点的极性 */
int gpio_sysfs_set_active_low(unsigned gpio, int value);
```

在一个内核驱动申请一个 GPIO 之后， 它可以通过 gpio_export()使其在 sysfs 接口中可见。该驱动可以控制信号方向是否可修改。这有助于防止用户空间代码无意间破坏重要的系统状态。

这个明确的导出有助于（通过使某些实验更容易来）调试，也可以提供一个始终存在的接口，与文档配合作为一个板级支持包的一部分。

在 GPIO 被导出之后，gpio_export_link()允许在 sysfs 文件系统的任何地方创建一个到这个 GPIO sysfs 节点的符号链接。这样驱动就可以通过一个描述性的名字，在 sysfs 中他们所拥有的设备下提供一个（到这个 GPIO sysfs 节点的）接口。

驱动可以使用 gpio_sysfs_set_active_low() 来在用户空间隐藏电路板之间 GPIO 线的极性差异。这个仅对 sysfs 接口起作用。极性的改变可以在 gpio_export()前后进行，且之前使能的轮询操作（poll(2)）支持（上升或下降沿）将会被重新配置来遵循这个设置。