# GPIO in the kernel: an introduction



By **Jonathan Corbet**
January 16, 2013

A GPIO (general-purpose I/O) device looks like the most boring sort of peripheral that a computer might offer. It is a single electrical signal that the CPU can either set to one of two values — zero or one, naturally — or read one of those values from (or both). Either way, a GPIO does not seem like a particularly expressive device. But, at their simplest, GPIOs can be used to control LEDs, reset lines, or pod-bay door locks. With additional "bit-banging" logic, GPIOs can be combined to implement higher-level protocols like i2c or DDC — a frequent occurrence on contemporary systems. GPIOs are thus useful in a lot of contexts.

GPIO lines seem to be especially prevalent in embedded systems; even so, there never seems to be enough of them. As one might expect, a system with dozens (or even hundreds) of GPIOs needs some sort of rational abstraction for managing them. The kernel has had such a mechanism since 2.6.21 (it was initially added by David Brownell). The API has changed surprisingly little since then, but that period of relative stasis may be about to come about to an end. The intended changes are best understood in the context of the existing API, though, so that is what this article will cover. Subsequent installments will look at how the GPIO API may evolve in the near future.

Naturally, there is an include file for working with GPIOs:

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

In current kernels, every GPIO in the system is represented by a simple unsigned integer. There is no provision for somehow mapping a desired function ("the sensor power line for the first camera device," say) onto a GPIO number; the code must come by that knowledge by other means. Often that is done through a long series of macro definitions; it is also possible to pass GPIO numbers through platform data or a device tree.

GPIOs must be allocated before use, though the current implementation does not enforce this requirement. The basic allocation function is:

```
int gpio_request(unsigned int gpio, const char *label);
```

The `gpio` parameter indicates which GPIO is required, while `label` associates a string with it that can later appear in sysfs. The usual convention applies: a zero return code indicates success; otherwise the return value will be a negative error number. A GPIO can be returned to the system with:

```
void gpio_free(unsigned int gpio);
```

There are some variants of these functions; `gpio_request_one()` can be used to set the initial configuration of the GPIO, and `gpio_request_array()` can request and configure a whole set of GPIOs with a single call. There are also "managed" versions (`devm_gpio_request()`, for example) that automatically handle cleanup if the developer forgets.

Some GPIOs are used for output, others for input. A suitably-wired GPIO can be used in either mode, though only one direction is active at any given time. Kernel code must inform the GPIO core of how a line is to be used; that is done with these functions:

```
int gpio_direction_input(unsigned int gpio);
int gpio_direction_output(unsigned int gpio, int value);
```

In either case, `gpio` is the GPIO number. In the output case, the value of the GPIO (zero or one) must also be specified; the GPIO will be set accordingly as part of the call. For both functions, the return value is again zero or a negative error number. The direction of (suitably capable) GPIOs can be changed at any time.

For input GPIOs, the current value can be read with:

```
int gpio_get_value(unsigned int gpio);
```

This function returns the value of the provided `gpio`; it has no provision for returning an error code. It is assumed (correctly in almost all cases) that any errors will be found when `gpio_direction_input()` is called, so checking the return value from that function is important.

Setting the value of output GPIOs can always be done using `gpio_direction_output()`, but, if the GPIO is known to be in output mode already, `gpio_set_value()` may be a bit more efficient:

```
void gpio_set_value(unsigned int gpio, int value);
```

Some GPIO controllers can generate interrupts when an input GPIO changes value. In such cases, code wishing to handle such interrupts should start by determining which IRQ number is associated with a given GPIO line:

```
int gpio_to_irq(unsigned int gpio);
```

The given `gpio` must have been obtained with `gpio_request()` and put into the input mode first. If there is an associated interrupt number, it will be passed back as the return value from `gpio_to_irq()`; otherwise a negative error number will be returned. Once obtained in this manner, the interrupt number can be passed to `request_irq()` to set up the handling of the interrupt.

Finally, the GPIO subsystem is able to represent GPIO lines via a sysfs hierarchy, allowing user space to query (and possibly modify) them. Kernel code can cause a specific GPIO to appear in sysfs with:

```
int gpio_export(unsigned int gpio, bool direction_may_change);
```

The `direction_may_change` parameter controls whether user space is allowed to change the direction of the GPIO; in many cases, allowing that control would be asking for bad things to happen to the system as a whole. A GPIO can be removed from sysfs with `gpio_unexport()` or given another name with `gpio_export_link()`.

And that is an overview of the kernel's low-level GPIO interface. A number of details have naturally been left out; see Documentation/gpio.txt for a more thorough description. Also omitted is the low-level driver's side of the API, by which GPIO lines can be made available to the GPIO subsystem; covering that API may be the subject of a future article. The next installment, though, will look at a couple of perceived deficiencies in the above-described API and how they might be remedied.

---

**GPIO in the kernel: an introduction**
Posted Jan 17, 2013 9:19 UTC (Thu) by **linusw** (subscriber, #40300) [Link]

Fun that you're diggin into this Jon, I have just had a presentation for the ELC accepted where I try to give an overview of recent changes in GPIO and pinctrl.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Jan 17, 2013 18:12 UTC (Thu) by **dougg** (subscriber, #1894) [Link]

Here is a view from the trenches of the Linux gpio subsystem, seen from the Atmel AT91 family, specifically the AT91SAM9x5 sub-family.

The biggest mistake was introduced initially: using a single sequence of kernel gpio numbers to represent a two level hierarchy. For example the AT91SAM9G20 has 3 banks: PA0-31, PB0-31 and PC0-31. For some crazy reason (something about not interfering with interrupt numbers) they mapped PA0 to 32 and the gpio numbers followed in sequence from there (hence PC31 is 127). Well sanity finally prevailed and around lk 3.5 they remapped PA0 to 0. Great, lots of user space code to rewrite.

And in lk 3.8.0-rc1 they were at it again. A gpio pin that had a sysfs name of /sys/class/gpio/gpio32 suddenly became /sys/class/gpio/pioB0 . Well that's a better name but why not use /sys/class/gpio/pb0 so as to agree with Atmel's naming (apart from capitalisation)?? And more user space code needs to be rewritten.

Back around lk 2.6.28 they removed a gpio pass-through driver leaving sysfs as (almost) the only way to get to gpios. I say almost because for serious work with AT91s mmap() is your friend. With luck in lk 3.9 we might get a well-designed block gpio driver so it will probably get vetoed or bowdlerized.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Jan 18, 2013 23:58 UTC (Fri) by **jimparis** (guest, #38647) [Link]

It sounds like your concern is accessing GPIOs from userspace, but I thought the primary intended consumer of this particular ABI was kernel drivers, and that accessing GPIOs via /sys was intended just for debugging, in lieu of a proper driver for the hardware you're playing with.

Then again, Documentation/gpio.txt does claim that "for some tasks, simple userspace GPIO drivers could be all that the system really needs", which at least suggests that it should be considered a production API. So complaints about API breakes are probably worth bringing up when they happen.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Jan 17, 2013 11:25 UTC (Thu) by **notti** (subscriber, #67230) [Link]

I think you missed the value in the gpio_set_value function

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Jan 17, 2013 13:05 UTC (Thu) by **corbet** (editor, #1) [Link]

That's a little embarrassing...and I rechecked all those prototypes. Fixed, thanks.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Jan 17, 2013 21:08 UTC (Thu) by **zuki** (guest, #41808) [Link]

Frankly I don't see the point of such a stub of an article. It would be better to have all three instalments now, and nothing for two weeks, then to be teased like that.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Jan 18, 2013 8:44 UTC (Fri) by **russell** (guest, #10458) [Link]

Everyone thinks GPIO is easy, and that probably explains why the interfaces are incomplete. GPIO usually has a lot more functionality than can be accessed.

1. GPIO can be both output and input at the same time. Yep you may want to read back the value of an output. MCUs I've used in the past offered this. After all output are really just stronger versions of the pull up/down :)

2. GPIO have pull up/down resistors.

3. And this is the BIG one. GPIO has state BEFORE the kernel boots. Bootstrap code can set the state of GPIO during the first few cycles of operation. This can eliminate the need for extra external hardware. However, the kernel blows this away and you have to wait for userspace to boot before you get control!!!! This behaviour on for example RPi required extra hardware to be added to the circuit to prevent this kind of transient boot behaviour from glitching relays.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Jan 18, 2013 11:50 UTC (Fri) by **etienne** (guest, #25256) [Link]

> GPIO can be both output and input at the same time

Reading back output GPIO is described in Documentation/gpio.txt
My main problem is that people (even software manager responsible of software "quality") do think it is clean to use the C preprocessor, either in C or C++.
That leads to massive amount of #define, to define the address, the value when active, the value when inactive, the shift position for each GPIO and the mask for that GPIO (you never know if the mask is shifted or not).
Then you have a set of GPIO to represent a 3 bit value, and the preprocessor will never tell you that you are trying to write 0x15 in this 3 bit value.
Also those quality specialists tell you to use the macro:
#define IOWRITE32BYTE(addr, val) *((volatile unsigned *)(addr)) = (val)
(Hint: try to write the same addr twice with a newer compiler).
Anyway on newer architecture, those are memory mapped and IHMO it is a lot cleaner to use C to declare them:

> volatile struct my_IO_s {
> #ifdef BIG_ENDIAN
> enum { healthy, fail } power_state : 1;
> unsigned power_active : 1;
> #else
> unsigned power_active : 1;
> enum { healthy, fail } power_state : 1;
> #endif
> } * const my_IO = (volatile struct my_IO_s *)0xDC002000;
> Instead of pages and pages of #define.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Jan 19, 2013 0:33 UTC (Sat) by **jimparis** (guest, #38647) [Link]

> > Also those quality specialists tell you to use the macro:
> > #define IOWRITE32BYTE(addr, val) *((volatile unsigned *)(addr)) = (val)
> > (Hint: try to write the same addr twice with a newer compiler).
>
> The volatile should ensure that both writes occur in that case.
>
> > Anyway on newer architecture, those are memory mapped and IHMO it is a lot cleaner to use C
> > to declare them:
> > volatile struct my_IO_s {
> > enum { healthy, fail } power_state : 1;
> > unsigned power_active : 1;
> > } * const my_IO = (volatile struct my_IO_s *)0xDC002000;
> > Instead of pages and pages of #define.
>
> I agree, and I do the same in embedded development, but it's not perfect. The three biggest problems:
>
> - You lose control over the access size. It can vary between compilers, compiler versions, architectures, and even ABIs for particular architectures. For example, see GCC bug #23623
> - They don't map well to unusual registers. Consider an interrupt flag register which behaves like:
>
>     ```
>     read 0: interrupt has not occurred
>     read 1: interrupt has occurred
>     write 0: no action
>     write 1: clear interrupt flag
>     ```
>
>     To clear a flag, you can't just do:
>
>     ```
>     my_IO->timer_interrupt = 1;
>     ```
>
>     because this can become
>
>     ```
>     tmp = *my_IO_addr | (1 << TIMER_INTERRUPT_SHIFT);
>     *my_IO_addr = tmp;
>     ```
>
>     Instead, you'd need to use something like:
>
>     ```
>     *my_IO = (struct my_IO_s) {
>         .timer_interrupt = 1
>     };
>     ```

which is not much better than the equivalent:

```
*my_IO_addr = (1 << TIMER_INTERRUPT_SHIFT);
```

- Even for a more normal register like a GPIO port, it's quite difficult to set two bits simultaneously with a single write. You can't do:

  ```
  *my_IO_addr |= (1 << PIN0_SHIFT) | (1 << PIN1_SHIFT);
  ```

  without something like:

  ```
  struct my_IO_s tmp = *my_IO;
  tmp.pin0 = 1;
  tmp.pin1 = 1;
  *my_IO = tmp;
  ```

Really it's one of the more annoying parts of C when doing embedded development on a constrained system, as it's hard to avoid without adding some layer of performance-killing indirection (like Arduino does).

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Jan 21, 2013 11:36 UTC (Mon) by **etienne** (guest, #25256) [Link]

>> #define IOWRITE32BITS(addr, val) *((volatile unsigned *)(addr)) = (val)
> The volatile should ensure that both writes occur in that case.

Last time I tried few days ago it did not work, you have to do:
#define IOWRITE32BITS(addr, val) do { \
volatile unsigned *ptr = (volatile unsigned *)(addr); \
*ptr = (val); \
} while (0)

> You lose control over the access size

Yes, I would also like to have bitfields accessed like they are declared, i.e. "char abit : 2;" accessed as a byte and "unsigned abit :2;" accessed as a 32 bits, because it is usual to have devices which cannot do short word access (lots of system on chip).

> [... using "(struct my_IO_s) {}" extension ...]

My system is more complex than Arduino, I have arrays of structures containing other arrays in these I/O areas, and all those #define blessed by software-quality people drive me mad.
#define FPGA1_FILTER_2_PARAMETER_4_ACTIVE 1
How to do a loop for each filters?
In short, in C or C++, unlike BASIC, the linker is made to manage addresses, the compiler manages offsets into these addresses; the C pre-processor has nothing to do there.
IHMO even GPIO in the kernel shall be identified with an address, not a #define.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Jan 18, 2013 14:42 UTC (Fri) by **dougg** (subscriber, #1894) [Link]

To your second and third points, welcome to pinctrl introduced in lk 3.7 and 3.8 . And while you are at it, look at the device tree stuff (aka "open firmware" and hence lots of "of_" adorned names). gpio pin states can and should be set before the Linux kernel gets a look in (e.g. in uboot or earlier), and when it does the kernel should resist the temptation to set those lines to some arbitrary state.

Things are moving so fast in the device tree/pinctrl area that it might be worth LWN writing another article on the subject.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Jan 28, 2013 21:27 UTC (Mon) by **BenHutchings** (subscriber, #37955) [Link]

> 1. GPIO can be both output and input at the same time. Yep you may want to read back the value of an output.

Indeed this is an absolute requirement for a GPIO used as an I2C SCL line.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Jan 18, 2013 14:20 UTC (Fri) by **kvaml** (subscriber, #61841) [Link]

Just in time to help with my new Raspberry Pi. Thanks.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Jul 20, 2016 7:52 UTC (Wed) by **codename_y** (guest, #109837) [Link]

Hi, I have tried to make a driver that utilize linux/gpio.h, but when I try to insmod it, I keep getting this error:

[ 3030.112044] gpiotest: Unknown symbol gpio_direction_output (err 0)
[ 3030.118537] gpiotest: Unknown symbol gpio_request (err 0)

Could you give me some insight about why this happened? I have tried to find by myself but no luck so far.

Thanks for your help

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Jul 20, 2016 8:05 UTC (Wed) by **mjg59** (subscriber, #23239) [Link]

They're EXPORT_SYMBOL_GPL - make sure that your MODULE_LICENSE tag is GPL compatible.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Oct 3, 2016 19:12 UTC (Mon) by **annonch** (guest, #111527) [Link]

Is it possible to register an interrupt to both falling edge and rising edge and then distinguish which case it is inside the interrupt handler?

When I try to register two interrupts I get an error resource is busy, I think because it is the same irq_number (-16)

When I try to register the interrupts with the IRQF_SHARED | IRQF_TRIGGER_RISING flags I get Invalid argument (-22)

So I create a interrupt with flags IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING but when the interrupt is called, I want to distinguish the two conditions separately.

It is an arm device with kernel 4.4.21

Any advice? :)

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Oct 3, 2016 19:27 UTC (Mon) by **nybble41** (subscriber, #55106) [Link]

> Is it possible to register an interrupt to both falling edge and rising edge and then distinguish which case it is inside the interrupt handler?

Generally speaking, no. I'm not familiar with your specific platform, but typically one configures the interrupt controller to look for either a rising edge or a falling edge (or low/high level) for each input and it generates a single interrupt when that event occurs. As a result, there is no way to statically configure it to watch for both rising and falling edges.

You could try dynamically reconfiguring the interrupt—set it as falling-edge in the rising-edge handler, and vice-versa—but performance would probably become an issue and you might miss some events depending on how long the handler is delayed. A better option, if you control the wiring and have a spare interrupt-capable GPIO, would be to route the same signal to two separate pins and set up a handler for each pin, one rising-edge and one falling-edge.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Oct 3, 2016 22:58 UTC (Mon) by **annonch** (guest, #111527) [Link]

Wow thank you for the suggestion, it seems to work perfectly! I have configured an additional pin fed with the same signal and I think this method will work well for my project. :)

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Dec 13, 2016 14:20 UTC (Tue) by **bla** (guest, #112986) [Link]

Anything speaking against using gpio_get_value in the isr to discern between rising and falling?

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Dec 13, 2016 14:56 UTC (Tue) by **nybble41** (subscriber, #55106) [Link]

> Anything speaking against using gpio_get_value in the isr to discern between rising and falling?

The ISR will only run on the rising edge _or_ the falling edge, not both, so gpio_get_value will generally return a known result: either high for a rising-edge-triggered interrupt, or low for falling-edge.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Dec 13, 2016 16:46 UTC (Tue) by **bla** (guest, #112986) [Link]

Why wouldn't IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING with request_irq work?

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Dec 13, 2016 18:28 UTC (Tue) by **nybble41** (subscriber, #55106) [Link]

> Why wouldn't IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING with request_irq work?

Things may be different on ARM or x86, but in my work on embedded PowerPC targets I have yet to see an interrupt controller than can be configured to trigger on both rising and falling edges for the same signal. Typically there are four possible trigger modes, of which only one can be active at a time: rising edge, falling edge, high level, or low level.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Dec 13, 2016 19:11 UTC (Tue) by **bla** (guest, #112986) [Link]

Ah, probably a gpio-controller/IRQ line limitation. I'll check if it works on a Raspberry (should, as Wiring has a BOTH option).

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Dec 15, 2016 11:07 UTC (Thu) by **bla** (guest, #112986) [Link]

Ok, you c a n set a combined interrupt mask and query the pin state in the ISR on an RPi, but gpio_get_value seems to have latency, which makes this approach pretty much useless in conjunction with switch bounce and the ISR time constraints.
For implementing long button presses, detecting the first flank and reading the level with a thread after some delay seems to be the better approach, it also doesn't require a state change to trigger the long press signal.

Reply to this comment

**GPIO in the kernel: an introduction**

Posted Dec 15, 2016 20:05 UTC (Thu) by **BlueLightning** (subscriber, #38978) [Link]

Surely the answer to switch bounce is to do the debouncing in hardware?

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Dec 15, 2016 20:32 UTC (Thu) by **pizza** (subscriber, #46) [Link]

But hardware debounce would require adding extra $0.02 worth of components and growing your board size by 2 square millimeters.

Why spent that money on every unit when it can be done once, "for free" in software?

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Dec 17, 2016 6:44 UTC (Sat) by **flussence** (subscriber, #85566) [Link]

Fun trivia: this is exactly what the NES Classic does. In their GPL code dump there's a kernel driver for the controller - it has the same wire protocol the Wiimote uses (internally!), but leaves the kernel to deal with all that hardware-related insanity and more.

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Feb 13, 2017 10:36 UTC (Mon) by **asmo** (guest, #114090) [Link]

Hi,
I need to use the same GPIO output in two different drivers.
Is it possible to use the set of apis gpio_request/gpio_get_value/gpio_set_value in parallel in both device drivers?

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Feb 13, 2019 16:51 UTC (Wed) by **Circuits** (guest, #130419) [Link]

Great information! Too bad this post was made in 2013 cause I have some questions here in 2018. Is the author or another authority still using LWN.net?

Reply to this comment

**GPIO in the kernel: an introduction**
Posted Feb 18, 2019 9:38 UTC (Mon) by **jem** (subscriber, #24231) [Link]

I do think the author is still using LWN.net. :) But he may be busy with other things.

Reply to this comment