GPIO Interfaces

This provides an overview of GPIO access conventions on Linux.

These calls use the gpio_* naming prefix. No other calls should use that prefix, or the related __gpio_* prefix.

What is a GPIO?
===============
A "General Purpose Input/Output" (GPIO) is a flexible software-controlled digital signal. They are provided from many kinds of chip, and are familiar to Linux developers working with embedded and custom hardware. Each GPIO represents a bit connected to a particular pin, or "ball" on Ball Grid Array (BGA) packages. Board schematics show which external hardware connects to which GPIOs. Drivers can be written generically, so that board setup code passes such pin configuration data to drivers.

System-on-Chip (SOC) processors heavily rely on GPIOs. In some cases, every non-dedicated pin can be configured as a GPIO; and most chips have at least several dozen of them. Programmable logic devices (like FPGAs) can easily provide GPIOs; multifunction chips like power managers, and audio codecs often have a few such pins to help with pin scarcity on SOCs; and there are also "GPIO Expander" chips that connect using the I2C or SPI serial busses. Most PC southbridges have a few dozen GPIO-capable pins (with only the BIOS firmware knowing how they're used).

The exact capabilities of GPIOs vary between systems. Common options:

  - Output values are writable (high=1, low=0). Some chips also have
    options about how that value is driven, so that for example only one
    value might be driven ... supporting "wire-OR" and similar schemes
    for the other value (notably, "open drain" signaling).

  - Input values are likewise readable (1, 0). Some chips support readback
    of pins configured as "output", which is very useful in such "wire-OR"
    cases (to support bidirectional signaling). GPIO controllers may have
    input de-glitch/debounce logic, sometimes with software controls.

  - Inputs can often be used as IRQ signals, often edge triggered but
    sometimes level triggered. Such IRQs may be configurable as system
    wakeup events, to wake the system from a low power state.

  - Usually a GPIO will be configurable as either input or output, as needed
    by different product boards; single direction ones exist too.

  - Most GPIOs can be accessed while holding spinlocks, but those accessed
    through a serial bus normally can't. Some systems support both types.

On a given board each GPIO is used for one specific purpose like monitoring MMC/SD card insertion/removal, detecting card writeprotect status, driving a LED, configuring a transceiver, bitbanging a serial bus, poking a hardware watchdog, sensing a switch, and so on.

GPIO conventions

================
Note that this is called a "convention" because you don't need to do it this
way, and it's no crime if you don't. There **are** cases where portability
is not the main issue; GPIOs are often used for the kind of board-specific
glue logic that may even change between board revisions, and can't ever be
used on a board that's wired differently. Only least-common-denominator
functionality can be very portable. Other features are platform-specific,
and that can be critical for glue logic.

Plus, this doesn't require any implementation framework, just an interface.
One platform might implement it as simple inline functions accessing chip
registers; another might implement it by delegating through abstractions
used for several very different kinds of GPIO controller. (There is some
optional code supporting such an implementation strategy, described later
in this document, but drivers acting as clients to the GPIO interface must
not care how it's implemented.)

That said, if the convention is supported on their platform, drivers should
use it when possible. Platforms must declare GENERIC_GPIO support in their
Kconfig (boolean true), and provide an <asm/gpio.h> file. Drivers that can't
work without standard GPIO calls should have Kconfig entries which depend
on GENERIC_GPIO. The GPIO calls are available, either as "real code" or as
optimized-away stubs, when drivers use the include file:

        #include <linux/gpio.h>

If you stick to this convention then it'll be easier for other developers to
see what your code is doing, and help maintain it.

Note that these operations include I/O barriers on platforms which need to
use them; drivers don't need to add them explicitly.

Identifying GPIOs

GPIOs are identified by unsigned integers in the range 0..MAX_INT. That
reserves "negative" numbers for other purposes like marking signals as
"not available on this board", or indicating faults. Code that doesn't
touch the underlying hardware treats these integers as opaque cookies.

Platforms define how they use those integers, and usually #define symbols
for the GPIO lines so that board-specific setup code directly corresponds
to the relevant schematics. In contrast, drivers should only use GPIO
numbers passed to them from that setup code, using platform_data to hold
board-specific pin configuration data (along with other board specific
data they need). That avoids portability problems.

So for example one platform uses numbers 32-159 for GPIOs; while another
uses numbers 0..63 with one set of GPIO controllers, 64-79 with another
type of GPIO controller, and on one particular board 80-95 with an FPGA.
The numbers need not be contiguous; either of those platforms could also
use numbers 2000-2063 to identify GPIOs in a bank of I2C GPIO expanders.

If you want to initialize a structure with an invalid GPIO number, use
some negative number (perhaps "-EINVAL"); that will never be valid. To
test if a number could reference a GPIO, you may use this predicate:

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

A number that's not valid will be rejected by calls which may request
or free GPIOs (see below). Other numbers may also be rejected; for
example, a number might be valid but unused on a given board.

Whether a platform supports multiple GPIO controllers is currently a
platform-specific implementation issue.

Using GPIOs

The first thing a system should do with a GPIO is allocate it, using
the gpio_request() call; see later.

One of the next things to do with a GPIO, often in board setup code when
setting up a platform_device using the GPIO, is mark its direction:

```
        /* set as input or output, returning 0 or negative errno */
        int gpio_direction_input(unsigned gpio);
        int gpio_direction_output(unsigned gpio, int value);
```

The return value is zero for success, else a negative errno. It should
be checked, since the get/set calls don't have error returns and since
misconfiguration is possible. You should normally issue these calls from
a task context. However, for spinlock-safe GPIOs it's OK to use them
before tasking is enabled, as part of early board setup.

For output GPIOs, the value provided becomes the initial output value.
This helps avoid signal glitching during system startup.

For compatibility with legacy interfaces to GPIOs, setting the direction
of a GPIO implicitly requests that GPIO (see below) if it has not been
requested already. That compatibility is being removed from the optional
gpiolib framework.

Setting the direction can fail if the GPIO number is invalid, or when
that particular GPIO can't be used in that mode. It's generally a bad
idea to rely on boot firmware to have set the direction correctly, since
it probably wasn't validated to do more than boot Linux. (Similarly,
that board setup code probably needs to multiplex that pin as a GPIO,
and configure pullups/pulldowns appropriately.)

Spinlock-Safe GPIO access

Most GPIO controllers can be accessed with memory read/write instructions.
That doesn't need to sleep, and can safely be done from inside IRQ handlers.
(That includes hardirq contexts on RT kernels.)

Use these calls to access such GPIOs:

```
        /* GPIO INPUT:  return zero or nonzero */
        int gpio_get_value(unsigned gpio);
```

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

The values are boolean, zero for low, nonzero for high. When reading the
value of an output pin, the value returned should be what's seen on the
pin ... that won't always match the specified output value, because of
issues including open-drain signaling and output latencies.

The get/set calls have no error returns because "invalid GPIO" should have
been reported earlier from gpio_direction_*(). However, note that not all
platforms can read the value of output pins; those that can't should always
return zero. Also, using these calls for GPIOs that can't safely be accessed
without sleeping (see below) is an error.

Platform-specific implementations are encouraged to optimize the two
calls to access the GPIO value in cases where the GPIO number (and for
output, value) are constant. It's normal for them to need only a couple
of instructions in such cases (reading or writing a hardware register),
and not to need spinlocks. Such optimized calls can make bitbanging
applications a lot more efficient (in both space and time) than spending
dozens of instructions on subroutine calls.

GPIO access that may sleep

Some GPIO controllers must be accessed using message based busses like I2C
or SPI. Commands to read or write those GPIO values require waiting to
get to the head of a queue to transmit a command and get its response.
This requires sleeping, which can't be done from inside IRQ handlers.

Platforms that support this type of GPIO distinguish them from other GPIOs
by returning nonzero from this call (which requires a valid GPIO number,
which should have been previously allocated with gpio_request):

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

To access such GPIOs, a different set of accessors is defined:

```
/* GPIO INPUT:  return zero or nonzero, might sleep */
int gpio_get_value_cansleep(unsigned gpio);

/* GPIO OUTPUT, might sleep */
void gpio_set_value_cansleep(unsigned gpio, int value);
```

Other than the fact that these calls might sleep, and will not be ignored
for GPIOs that can't be accessed from IRQ handlers, these calls act the
same as the spinlock-safe calls.

Claiming and Releasing GPIOs

To help catch system configuration errors, two calls are defined.

```
/* request GPIO, returning 0 or negative errno.
 * non-null labels may be useful for diagnostics.
 */
```

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

        /* release previously-claimed GPIO */
        void gpio_free(unsigned gpio);
```

Passing invalid GPIO numbers to gpio_request() will fail, as will requesting
GPIOs that have already been claimed with that call. The return value of
gpio_request() must be checked. You should normally issue these calls from
a task context. However, for spinlock-safe GPIOs it's OK to request GPIOs
before tasking is enabled, as part of early board setup.

These calls serve two basic purposes. One is marking the signals which
are actually in use as GPIOs, for better diagnostics; systems may have
several hundred potential GPIOs, but often only a dozen are used on any
given board. Another is to catch conflicts, identifying errors when
(a) two or more drivers wrongly think they have exclusive use of that
signal, or (b) something wrongly believes it's safe to remove drivers
needed to manage a signal that's in active use. That is, requesting a
GPIO can serve as a kind of lock.

Some platforms may also use knowledge about what GPIOs are active for
power management, such as by powering down unused chip sectors and, more
easily, gating off unused clocks.

Note that requesting a GPIO does NOT cause it to be configured in any
way; it just marks that GPIO as in use. Separate code must handle any
pin setup (e.g. controlling which pin the GPIO uses, pullup/pulldown).

Also note that it's your responsibility to have stopped using a GPIO
before you free it.

Considering in most cases GPIOs are actually configured right after they
are claimed, three additional calls are defined:

        /* request a single GPIO, with initial configuration specified by
         * 'flags', identical to gpio_request() wrt other arguments and
         * return value
         */
        int gpio_request_one(unsigned gpio, unsigned long flags, const char
*label);

        /* request multiple GPIOs in a single call
         */
        int gpio_request_array(struct gpio *array, size_t num);

        /* release multiple GPIOs in a single call
         */
        void gpio_free_array(struct gpio *array, size_t num);

where 'flags' is currently defined to specify the following properties:

        * GPIOF_DIR_IN   - to configure direction as input
        * GPIOF_DIR_OUT   - to configure direction as output

        * GPIOF_INIT_LOW   - as output, set initial level to LOW
        * GPIOF_INIT_HIGH   - as output, set initial level to HIGH
```

since GPIOF_INIT_* are only valid when configured as output, so group valid
combinations as:

        * GPIOF_IN              - configure as input
        * GPIOF_OUT_INIT_LOW    - configured as output, initial level LOW
        * GPIOF_OUT_INIT_HIGH   - configured as output, initial level HIGH

In the future, these flags can be extended to support more properties such
as open-drain status.

Further more, to ease the claim/release of multiple GPIOs, 'struct gpio' is
introduced to encapsulate all three fields as:

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

A typical example of usage:

        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));


GPIOs mapped to IRQs
--------------------
GPIO numbers are unsigned integers; so are IRQ numbers.  These make up
two logically distinct namespaces (GPIO 0 need not use IRQ 0).  You can
map between them using calls like:

        /* map GPIO numbers to IRQ numbers */
        int gpio_to_irq(unsigned gpio);

        /* map IRQ numbers to GPIO numbers (avoid using this) */
        int irq_to_gpio(unsigned irq);

Those return either the corresponding number in the other namespace, or
else a negative errno code if the mapping can't be done.  (For example,
some GPIOs can't be used as IRQs.)  It is an unchecked error to use a GPIO

number that wasn't set up as an input using gpio_direction_input(), or
to use an IRQ number that didn't originally come from gpio_to_irq().

These two mapping calls are expected to cost on the order of a single
addition or subtraction.  They're not allowed to sleep.

Non-error values returned from gpio_to_irq() can be passed to request_irq()
or free_irq().  They will often be stored into IRQ resources for platform
devices, by the board-specific initialization code.  Note that IRQ trigger
options are part of the IRQ interface, e.g. IRQF_TRIGGER_FALLING, as are
system wakeup capabilities.

Non-error values returned from irq_to_gpio() would most commonly be used
with gpio_get_value(), for example to initialize or update driver state
when the IRQ is edge-triggered.  Note that some platforms don't support
this reverse mapping, so you should avoid using it.


Emulating Open Drain Signals
----------------------------
Sometimes shared signals need to use "open drain" signaling, where only the
low signal level is actually driven.  (That term applies to CMOS transistors;
"open collector" is used for TTL.)  A pullup resistor causes the high signal
level.  This is sometimes called a "wire-AND"; or more practically, from the
negative logic (low=true) perspective this is a "wire-OR".

One common example of an open drain signal is a shared active-low IRQ line.
Also, bidirectional data bus signals sometimes use open drain signals.

Some GPIO controllers directly support open drain outputs; many don't.  When
you need open drain signaling but your hardware doesn't directly support it,
there's a common idiom you can use to emulate it with any GPIO pin that can
be used as either an input or an output:

 LOW:    gpio_direction_output(gpio, 0) ... this drives the signal
         and overrides the pullup.

 HIGH:   gpio_direction_input(gpio) ... this turns off the output,
         so the pullup (or some other device) controls the signal.

If you are "driving" the signal high but gpio_get_value(gpio) reports a low
value (after the appropriate rise time passes), you know some other component
is driving the shared signal low.  That's not necessarily an error.  As one
common example, that's how I2C clocks are stretched:  a slave that needs a
slower clock delays the rising edge of SCK, and the I2C master adjusts its
signaling rate accordingly.


What do these conventions omit?
===============================
One of the biggest things these conventions omit is pin multiplexing, since
this is highly chip-specific and nonportable.  One platform might not need
explicit multiplexing; another might have just two options for use of any
given pin; another might have eight options per pin; another might be able
to route a given GPIO to any one of several pins.  (Yes, those examples all
come from systems that run Linux today.)

Related to multiplexing is configuration and enabling of the pullups or
pulldowns integrated on some platforms.  Not all platforms support them,
or support them in the same way; and any given board might use external
pullups (or pulldowns) so that the on-chip ones should not be used.
(When a circuit needs 5 kOhm, on-chip 100 kOhm resistors won't do.)
Likewise drive strength (2 mA vs 20 mA) and voltage (1.8V vs 3.3V) is a
platform-specific issue, as are models like (not) having a one-to-one
correspondence between configurable pins and GPIOs.

There are other system-specific mechanisms that are not specified here,
like the aforementioned options for input de-glitching and wire-OR output.
Hardware may support reading or writing GPIOs in gangs, but that's usually
configuration dependent:  for GPIOs sharing the same bank.  (GPIOs are
commonly grouped in banks of 16 or 32, with a given SOC having several such
banks.)  Some systems can trigger IRQs from output GPIOs, or read values
from pins not managed as GPIOs.  Code relying on such mechanisms will
necessarily be nonportable.

Dynamic definition of GPIOs is not currently standard; for example, as
a side effect of configuring an add-on board with some GPIO expanders.


GPIO implementor's framework (OPTIONAL)
=======================================
As noted earlier, there is an optional implementation framework making it
easier for platforms to support different kinds of GPIO controller using
the same programming interface.  This framework is called "gpiolib".

As a debugging aid, if debugfs is available a /sys/kernel/debug/gpio file
will be found there.  That will list all the controllers registered through
this framework, and the state of the GPIOs currently in use.


Controller Drivers: gpio_chip
-----------------------------
In this framework each GPIO controller is packaged as a "struct gpio_chip"
with information common to each controller of that type:

 - methods to establish GPIO direction
 - methods used to access GPIO values
 - flag saying whether calls to its methods may sleep
 - optional debugfs dump method (showing extra state like pullup config)
 - label for diagnostics

There is also per-instance data, which may come from device.platform_data:
the number of its first GPIO, and how many GPIOs it exposes.

The code implementing a gpio_chip should support multiple instances of the
controller, possibly using the driver model.  That code will configure each
gpio_chip and issue gpiochip_add().  Removing a GPIO controller should be
rare; use gpiochip_remove() when it is unavoidable.

Most often a gpio_chip is part of an instance-specific structure with state
not exposed by the GPIO interfaces, such as addressing, power management,
and more.  Chips such as codecs will have complex non-GPIO state.

Any debugfs dump method should normally ignore signals which haven't been
requested as GPIOs.  They can use gpiochip_is_requested(), which returns
either NULL or the label associated with that GPIO when it was requested.


Platform Support
----------------
To support this framework, a platform's Kconfig will "select" either
ARCH_REQUIRE_GPIOLIB or ARCH_WANT_OPTIONAL_GPIOLIB
and arrange that its <asm/gpio.h> includes <asm-generic/gpio.h> and defines
three functions: gpio_get_value(), gpio_set_value(), and gpio_cansleep().
They may also want to provide a custom value for ARCH_NR_GPIOS.

ARCH_REQUIRE_GPIOLIB means that the gpio-lib code will always get compiled
into the kernel on that architecture.

ARCH_WANT_OPTIONAL_GPIOLIB means the gpio-lib code defaults to off and the user
can enable it and build it into the kernel optionally.

If neither of these options are selected, the platform does not support
GPIOs through GPIO-lib and the code cannot be enabled by the user.

Trivial implementations of those functions can directly use framework
code, which always dispatches through the gpio_chip:

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

Fancier implementations could instead define those as inline functions with
logic optimizing access to specific SOC-based GPIOs. For example, if the
referenced GPIO is the constant "12", getting or setting its value could
cost as little as two or three instructions, never sleeping. When such an
optimization is not possible those calls must delegate to the framework
code, costing at least a few dozen instructions. For bitbanged I/O, such
instruction savings can be significant.

For SOCs, platform-specific code defines and registers gpio_chip instances
for each bank of on-chip GPIOs. Those GPIOs should be numbered/labeled to
match chip vendor documentation, and directly match board schematics. They
may well start at zero and go up to a platform-specific limit. Such GPIOs
are normally integrated into platform initialization to make them always be
available, from arch_initcall() or earlier; they can often serve as IRQs.

Board Support

For external GPIO controllers -- such as I2C or SPI expanders, ASICs, multi
function devices, FPGAs or CPLDs -- most often board-specific code handles
registering controller devices and ensures that their drivers know what GPIO
numbers to use with gpiochip_add(). Their numbers often start right after
platform-specific GPIOs.

For example, board setup code could create structures identifying the range
of GPIOs that chip will expose, and passes them to each GPIO expander chip

using platform_data. Then the chip driver's probe() routine could pass that
data to gpiochip_add().

Initialization order can be important. For example, when a device relies on
an I2C-based GPIO, its probe() routine should only be called after that GPIO
becomes available. That may mean the device should not be registered until
calls for that GPIO can work. One way to address such dependencies is for
such gpio_chip controllers to provide setup() and teardown() callbacks to
board specific code; those board specific callbacks would register devices
once all the necessary resources are available, and remove them later when
the GPIO controller device becomes unavailable.

Sysfs Interface for Userspace (OPTIONAL)
==
Platforms which use the "gpiolib" implementors framework may choose to
configure a sysfs user interface to GPIOs. This is different from the
debugfs interface, since it provides control over GPIO direction and
value instead of just showing a gpio state summary. Plus, it could be
present on production systems without debugging support.

Given appropriate hardware documentation for the system, userspace could
know for example that GPIO #23 controls the write protect line used to
protect boot loader segments in flash memory. System upgrade procedures
may need to temporarily remove that protection, first importing a GPIO,
then changing its output state, then updating the code before re-enabling
the write protection. In normal use, GPIO #23 would never be touched,
and the kernel would have no need to know about it.

Again depending on appropriate hardware documentation, on some systems
userspace GPIO can be used to determine system configuration data that
standard kernels won't know about. And for some tasks, simple userspace
GPIO drivers could be all that the system really needs.

Note that standard kernel drivers exist for common "LEDs and Buttons"
GPIO tasks: "leds-gpio" and "gpio_keys", respectively. Use those
instead of talking directly to the GPIOs; they integrate with kernel
frameworks better than your userspace code could.

Paths in Sysfs

There are three kinds of entry in /sys/class/gpio:

    -    Control interfaces used to get userspace control over GPIOs;

    -    GPIOs themselves; and

    -    GPIO controllers ("gpio_chip" instances).

That's in addition to standard files including the "device" symlink.

The control interfaces are write-only:

    /sys/class/gpio/

"export" ... Userspace may ask the kernel to export control of
        a GPIO to userspace by writing its number to this file.

        Example: "echo 19 > export" will create a "gpio19" node
        for GPIO #19, if that's not requested by kernel code.

"unexport" ... Reverses the effect of exporting to userspace.

        Example: "echo 19 > unexport" will remove a "gpio19"
        node exported using the "export" file.

GPIO signals have paths like /sys/class/gpio/gpio42/ (for GPIO #42)
and have the following read/write attributes:

    /sys/class/gpio/gpioN/

        "direction" ... reads as either "in" or "out". This value may
                normally be written. Writing as "out" defaults to
                initializing the value as low. To ensure glitch free
                operation, values "low" and "high" may be written to
                configure the GPIO as an output with that initial value.

                Note that this attribute *will not exist* if the kernel
                doesn't support changing the direction of a GPIO, or
                it was exported by kernel code that didn't explicitly
                allow userspace to reconfigure this GPIO's direction.

        "value" ... reads as either 0 (low) or 1 (high). If the GPIO
                is configured as an output, this value may be written;
                any nonzero value is treated as high.

        "edge" ... reads as either "none", "rising", "falling", or
                "both". Write these strings to select the signal edge(s)
                that will make poll(2) on the "value" file return.

                This file exists only if the pin can be configured as an
                interrupt generating input pin.

        "active_low" ... reads as either 0 (false) or 1 (true). Write
                any nonzero value to invert the value attribute both
                for reading and writing. Existing and subsequent
                poll(2) support configuration via the edge attribute
                for "rising" and "falling" edges will follow this
                setting.

GPIO controllers have paths like /sys/class/gpio/gpiochip42/ (for the
controller implementing GPIOs starting at #42) and have the following
read-only attributes:

    /sys/class/gpio/gpiochipN/

        "base" ... same as N, the first GPIO managed by this chip

        "label" ... provided for diagnostics (not always unique)

        "ngpio" ... how many GPIOs this manges (N to N + ngpio - 1)

Board documentation should in most cases cover what GPIOs are used for
what purposes. However, those numbers are not always stable; GPIOs on
a daughtercard might be different depending on the base board being used,
or other cards in the stack. In such cases, you may need to use the
gpiochip nodes (possibly in conjunction with schematics) to determine
the correct GPIO number to use for a given signal.

Exporting from Kernel code

Kernel code can explicitly manage exports of GPIOs which have already been
requested using gpio_request():

```
        /* export the GPIO to userspace */
        int gpio_export(unsigned gpio, bool direction_may_change);

        /* reverse gpio_export() */
        void gpio_unexport();

        /* create a sysfs link to an exported GPIO node */
        int gpio_export_link(struct device *dev, const char *name,
                unsigned gpio)

        /* change the polarity of a GPIO node in sysfs */
        int gpio_sysfs_set_active_low(unsigned gpio, int value);
```

After a kernel driver requests a GPIO, it may only be made available in
the sysfs interface by gpio_export(). The driver can control whether the
signal direction may change. This helps drivers prevent userspace code
from accidentally clobbering important system state.

This explicit exporting can help with debugging (by making some kinds
of experiments easier), or can provide an always-there interface that's
suitable for documenting as part of a board support package.

After the GPIO has been exported, gpio_export_link() allows creating
symlinks from elsewhere in sysfs to the GPIO sysfs node. Drivers can
use this to provide the interface under their own device in sysfs with
a descriptive name.

Drivers can use gpio_sysfs_set_active_low() to hide GPIO line polarity
differences between boards from user space. This only affects the
sysfs interface. Polarity change can be done both before and after
gpio_export(), and previously enabled poll(2) support for either
rising or falling edge will be reconfigured to follow this setting.