GPIO Driver Interface

This document serves as a guide for writers of GPIO chip drivers.

Each GPIO controller driver needs to include the following header, which defines the structures used to define a GPIO driver:

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

Internal Representation of GPIOs

A GPIO chip handles one or more GPIO lines. To be considered a GPIO chip, the lines must conform to the definition: General Purpose Input/Output. If the line is not general purpose, it is not GPIO and should not be handled by a GPIO chip. The use case is the indicative: certain lines in a system may be called GPIO but serve a very particular purpose thus not meeting the criteria of a general purpose I/O. On the other hand a LED driver line may be used as a GPIO and should therefore still be handled by a GPIO chip driver.

Inside a GPIO driver, individual GPIO lines are identified by their hardware number, sometime also referred to as `offset`, which is a unique number between 0 and n-1, n being the number of GPIOs managed by the chip.

The hardware GPIO number should be something intuitive to the hardware, for example if a system uses a memory-mapped set of I/O-registers where 32 GPIO lines are handled by one bit per line in a 32-bit register, it makes sense to use hardware offsets 0..31 for these, corresponding to bits 0..31 in the register.

This number is purely internal: the hardware number of a particular GPIO line is never made visible outside of the driver.

On top of this internal number, each GPIO line also needs to have a global number in the integer GPIO namespace so that it can be used with the legacy GPIO interface. Each chip must thus have a "base" number (which can be automatically assigned), and for each GPIO line the global number will be (base + hardware number). Although the integer representation is considered deprecated, it still has many users and thus needs to be maintained.

So for example one platform could use global numbers 32-159 for GPIOs, with a controller defining 128 GPIOs at a "base" of 32 ; while another platform uses global 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 legacy numbers need not be contiguous; either of those platforms could also use numbers 2000-2063 to identify GPIO lines in a bank of I2C GPIO expanders.

Controller Drivers: gpio_chip

In the gpiolib framework each GPIO controller is packaged as a "struct gpio_chip" (see <linux/gpio/driver.h> for its complete definition) with members common to each controller of that type, these should be assigned by the driver code:

- methods to establish GPIO line direction
- methods used to access GPIO line values
- method to set electrical configuration for a given GPIO line
- method to return the IRQ number associated to a given GPIO line
- flag saying whether calls to its methods may sleep
- optional line names array to identify lines
- optional debugfs dump method (showing extra state information)
- optional base number (will be automatically assigned if omitted)
- optional label for diagnostics and GPIO chip mapping using platform data

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

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

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

Realtime considerations: the GPIO driver should not use spinlock_t or any sleepable APIs (like PM runtime) in its gpio_chip implementation (.get/.set and direction control callbacks) if it is expected to call GPIO APIs from atomic context on realtime kernels (inside hard IRQ handlers and similar contexts). Normally this should not be required.

GPIO electrical configuration

GPIO lines can be configured for several electrical modes of operation by using the .set_config() callback. Currently this API supports setting:

- Debouncing

- Single-ended modes (open drain/open source)
- Pull up and pull down resistor enablement

These settings are described below.

The .set_config() callback uses the same enumerators and configuration semantics as the generic pin control drivers. This is not a coincidence: it is possible to assign the .set_config() to the function gpiochip_generic_config() which will result in pinctrl_gpio_set_config() being called and eventually ending up in the pin control back-end "behind" the GPIO controller, usually closer to the actual pins. This way the pin controller can manage the below listed GPIO configurations.

If a pin controller back-end is used, the GPIO controller or hardware description needs to provide "GPIO ranges" mapping the GPIO line offsets to pin numbers on the pin controller so they can properly cross-reference each other.

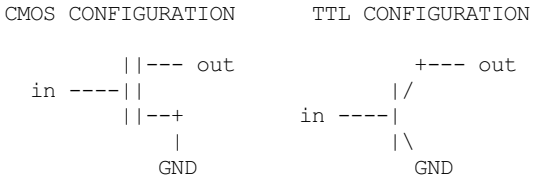
GPIO lines with debounce support

Debouncing is a configuration set to a pin indicating that it is connected to a mechanical switch or button, or similar that may bounce. Bouncing means the line is pulled high/low quickly at very short intervals for mechanical reasons. This can result in the value being unstable or irqs fireing repeatedly unless the line is debounced.

Debouncing in practice involves setting up a timer when something happens on the line, wait a little while and then sample the line again, so see if it still has the same value (low or high). This could also be repeated by a clever state machine, waiting for a line to become stable. In either case, it sets a certain number of milliseconds for debouncing, or just "on/off" if that time is not configurable.

GPIO lines with open drain/source support

Open drain (CMOS) or open collector (TTL) means the line is not actively driven high: instead you provide the drain/collector as output, so when the transistor is not open, it will present a high-impedance (tristate) to the external rail:



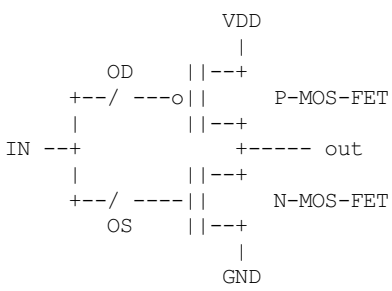
This configuration is normally used as a way to achieve one of two things:

- Level-shifting: to reach a logical level higher than that of the silicon where the output resides.
- Inverse wire-OR on an I/O line, for example a GPIO line, making it possible for any driving stage on the line to drive it low even if any other output to the same line is simultaneously driving it high. A special case of this is driving the SCL and SDA lines of an I2C bus, which is by definition a wire-OR bus.

Both use cases require that the line be equipped with a pull-up resistor. This resistor will make the line tend to high level unless one of the transistors on the rail actively pulls it down.

The level on the line will go as high as the VDD on the pull-up resistor, which may be higher than the level supported by the transistor, achieving a level-shift to the higher VDD.

Integrated electronics often have an output driver stage in the form of a CMOS "totem-pole" with one N-MOS and one P-MOS transistor where one of them drives the line high and one of them drives the line low. This is called a push-pull output. The "totem-pole" looks like so:



The desired output signal (e.g. coming directly from some GPIO output register) arrives at IN. The switches named "OD" and "OS" are normally closed, creating a push-pull circuit.

Consider the little "switches" named "OD" and "OS" that enable/disable the P-MOS or N-MOS transistor right after the split of the input. As you can see, either transistor will go totally numb if this switch is open. The totem-pole is then halved and give high impedance instead of actively driving the line high or low respectively. That is usually how software-controlled open drain/source works.

Some GPIO hardware come in open drain / open source configuration. Some are hard-wired lines that will only support open drain or open source no matter what: there is only one transistor there. Some are software-configurable: by flipping a bit in a register the output can be configured as open drain or open source, in practice by flicking open the switches labeled "OD" and "OS" in the drawing above.

By disabling the P-MOS transistor, the output can be driven between GND and high impedance (open drain), and by disabling the N-MOS transistor, the output can be driven between VDD and high impedance (open source). In the first case, a pull-up resistor is needed on the outgoing rail to complete the circuit, and in the second case, a pull-down resistor is needed on the rail.

Hardware that supports open drain or open source or both, can implement a special callback in the gpio_chip: .set_config() that takes a generic pinconf packed value telling whether to configure the line as open drain, open source or push-pull. This will happen in response to the GPIO_OPEN_DRAIN or GPIO_OPEN_SOURCE flag set in the machine file, or coming from other hardware descriptions.

If this state can not be configured in hardware, i.e. if the GPIO hardware does not support open drain/open source in hardware, the GPIO library will instead use a trick: when a line is set as output, if the line is flagged as open drain, and the IN output value is low, it will be driven low as usual. But if the IN output value is set to high, it will instead *NOT* be driven high, instead it will be switched to input, as input mode is high impedance, thus achieveing an "open drain emulation" of sorts: electrically the behaviour will be identical, with the exception of possible hardware glitches when switching the mode of the line.

For open source configuration the same principle is used, just that instead of actively driving the line low, it is set to input.

GPIO lines with pull up/down resistor support

A GPIO line can support pull-up/down using the .set_config() callback. This means that a pull up or pull-down resistor is available on the output of the GPIO line, and this resistor is software controlled.

In discrete designs, a pull-up or pull-down resistor is simply soldered on the circuit board. This is not something we deal with or model in software. The most you will think about these lines is that they will very likely be configured as open drain or open source (see the section above).

The .set_config() callback can only turn pull up or down on and off, and will no have any semantic knowledge about the resistance used. It will only say switch a bit in a register enabling or disabling pull-up or pull-down.

If the GPIO line supports shunting in different resistance values for the pull-up or pull-down resistor, the GPIO chip callback .set_config() will not suffice. For these complex use cases, a combined GPIO chip and pin controller need to be implemented, as the pin config interface of a pin controller supports more versatile control over electrical properties and can handle different pull-up or pull-down resistance values.

GPIO drivers providing IRQs

It is custom that GPIO drivers (GPIO chips) are also providing interrupts, most often cascaded off a parent interrupt controller, and in some special cases the GPIO logic is melded with a SoC's primary interrupt controller.

The IRQ portions of the GPIO block are implemented using an irq_chip, using the header <linux/irq.h>. So this combined driver is utilizing two sub- systems simultaneously: gpio and irq.

It is legal for any IRQ consumer to request an IRQ from any irqchip even if it is a combined GPIO+IRQ driver. The basic premise is that gpio_chip and irq_chip are orthogonal, and offering their services independent of each other.

gpiod_to_irq() is just a convenience function to figure out the IRQ for a certain GPIO line and should not be relied upon to have been called before the IRQ is used.

Always prepare the hardware and make it ready for action in respective callbacks from the GPIO and irq_chip APIs. Do not rely on gpiod_to_irq() having been called first.

We can divide GPIO irqchips in two broad categories:

- CASCADED INTERRUPT CHIPS: this means that the GPIO chip has one common interrupt output line, which is triggered by any enabled GPIO line on that chip. The interrupt output line will then be routed to an parent interrupt controller one level up, in the most simple case the systems primary interrupt controller. This is modeled by an irqchip that will inspect bits inside the GPIO controller to figure out which line fired it. The irqchip part of the driver needs to inspect registers to figure this out and it will likely also need to acknowledge that it is handling the interrupt by clearing some bit (sometime implicitly, by just reading a status register) and it will often need to set up the configuration such as edge sensitivity (rising or falling edge, or high/low level interrupt for example).
- HIERARCHICAL INTERRUPT CHIPS: this means that each GPIO line has a dedicated irq line to a parent interrupt controller one level up. There is no need to inquire the GPIO hardware to figure out which line has fired, but it may still be necessary to acknowledge the interrupt and set up configuration such as edge sensitivity.

Realtime considerations: a realtime compliant GPIO driver should not use spinlock_t or any sleepable APIs (like PM runtime) as part of its irqchip implementation.

- spinlock_t should be replaced with raw_spinlock_t.[1]
- If sleepable APIs have to be used, these can be done from the .irq_bus_lock() and .irq_bus_unlock() callbacks, as these are the only slowpath callbacks on an irqchip. Create the callbacks if needed.[2]

Cascaded GPIO irqchips

Cascaded GPIO irqchips usually fall in one of three categories:

- CHAINED CASCADED GPIO IRQCHIPS: these are usually the type that is embedded on an SoC. This means that there is a fast IRQ flow handler for the GPIOs that gets called in a chain from the parent IRQ handler, most typically the system interrupt controller. This means that the GPIO irqchip handler will be called immediately from the parent irqchip, while holding the IRQs disabled. The GPIO irqchip will then end up calling something like this sequence in its interrupt handler:

  ```
  static irqreturn_t foo_gpio_irq(int irq, void *data)
      chained_irq_enter(...);
      generic_handle_irq(...);
      chained_irq_exit(...);
  ```

  Chained GPIO irqchips typically can NOT set the .can_sleep flag on struct gpio_chip, as everything happens directly in the callbacks: no slow bus traffic like I2C can be used.

  Realtime considerations: Note that chained IRQ handlers will not be forced threaded on -RT. As a result, spinlock_t or any sleepable APIs (like PM runtime) can't be used in a chained IRQ handler.

  If required (and if it can't be converted to the nested threaded GPIO irqchip, see below) a chained IRQ handler can be converted to generic irq handler and this way it will become a threaded IRQ handler on -RT and a hard IRQ handler on non-RT (for example, see [3]).

  The generic_handle_irq() is expected to be called with IRQ disabled, so the IRQ core will complain if it is called from an IRQ handler which is forced to a thread. The "fake?" raw lock can be used to work around this problem:

  ```
  raw_spinlock_t wa_lock;
  static irqreturn_t omap_gpio_irq_handler(int irq, void *gpiobank)
      unsigned long wa_lock_flags;
      raw_spin_lock_irqsave(&bank->wa_lock, wa_lock_flags);
      generic_handle_irq(irq_find_mapping(bank->chip.irq.domain, bit));
      raw_spin_unlock_irqrestore(&bank->wa_lock, wa_lock_flags);
  ```

- GENERIC CHAINED GPIO IRQCHIPS: these are the same as "CHAINED GPIO irqchips", but chained IRQ handlers are not used. Instead GPIO IRQs dispatching is performed by generic IRQ handler which is configured using request_irq(). The GPIO irqchip will then end up calling something like this sequence in its interrupt handler:

  ```
  static irqreturn_t gpio_rcar_irq_handler(int irq, void *dev_id)
      for each detected GPIO IRQ
          generic_handle_irq(...);
  ```

  Realtime considerations: this kind of handlers will be forced threaded on -RT, and as result the IRQ core will complain that generic_handle_irq() is called with IRQ enabled and the same work-around as for "CHAINED GPIO irqchips" can be applied.

- NESTED THREADED GPIO IRQCHIPS: these are off-chip GPIO expanders and any other GPIO irqchip residing on the other side of a sleeping bus such as I2C or SPI.

  Of course such drivers that need slow bus traffic to read out IRQ status and similar, traffic which may in turn incur other IRQs to happen, cannot be handled in a quick IRQ handler with IRQs disabled. Instead they need to spawn a thread and then mask the parent IRQ line until the interrupt is handled by the driver. The hallmark of this driver is to call something like this in its interrupt handler:

  ```
  static irqreturn_t foo_gpio_irq(int irq, void *data)
      ...
      handle_nested_irq(irq);
  ```

  The hallmark of threaded GPIO irqchips is that they set the .can_sleep flag on struct gpio_chip to true, indicating that this chip may sleep when accessing the GPIOs.

  These kinds of irqchips are inherently realtime tolerant as they are already set up to handle sleeping contexts.

Infrastructure helpers for GPIO irqchips

To help out in handling the set-up and management of GPIO irqchips and the associated irqdomain and resource allocation callbacks. These are activated by selecting the Kconfig symbol GPIOLIB_IRQCHIP. If the symbol IRQ_DOMAIN_HIERARCHY is also selected, hierarchical helpers will also be provided. A big portion of overhead code will be managed by gpiolib, under the assumption that your interrupts are 1-to-1-mapped to the GPIO line index:

GPIO line offset	Hardware IRQ
0	0
1	1
2	2
...	...
ngpio-1	ngpio-1

If some GPIO lines do not have corresponding IRQs, the bitmask valid_mask and the flag need_valid_mask in gpio_irq_chip can be used to mask off some lines as invalid for associating with IRQs.

The preferred way to set up the helpers is to fill in the struct gpio_irq_chip inside struct gpio_chip before adding the gpio_chip. If you do this, the additional irq_chip will be set up by gpiolib at the same time as setting up the rest of the GPIO functionality. The following is a typical example of a chained cascaded interrupt handler using the gpio_irq_chip:

```
/* Typical state container with dynamic irqchip */
struct my_gpio {
    struct gpio_chip gc;
    struct irq_chip irq;
};

int irq; /* from platform etc */
struct my_gpio *g;
struct gpio_irq_chip *girq;

/* Set up the irqchip dynamically */
g->irq.name = "my_gpio_irq";
g->irq.irq_ack = my_gpio_ack_irq;
g->irq.irq_mask = my_gpio_mask_irq;
g->irq.irq_unmask = my_gpio_unmask_irq;
g->irq.irq_set_type = my_gpio_set_irq_type;

/* Get a pointer to the gpio_irq_chip */
girq = &g->gc.irq;
girq->chip = &g->irq;
girq->parent_handler = ftgpio_gpio_irq_handler;
girq->num_parents = 1;
girq->parents = devm_kcalloc(dev, 1, sizeof(*girq->parents),
                             GFP_KERNEL);
if (!girq->parents)
    return -ENOMEM;
girq->default_type = IRQ_TYPE_NONE;
girq->handler = handle_bad_irq;
girq->parents[0] = irq;

return devm_gpiochip_add_data(dev, &g->gc, g);
```

The helper supports using threaded interrupts as well. Then you just request the interrupt separately and go with it:

```
/* Typical state container with dynamic irqchip */
struct my_gpio {
    struct gpio_chip gc;
    struct irq_chip irq;
};

int irq; /* from platform etc */
struct my_gpio *g;
struct gpio_irq_chip *girq;

/* Set up the irqchip dynamically */
g->irq.name = "my_gpio_irq";
g->irq.irq_ack = my_gpio_ack_irq;
g->irq.irq_mask = my_gpio_mask_irq;
g->irq.irq_unmask = my_gpio_unmask_irq;
g->irq.irq_set_type = my_gpio_set_irq_type;

ret = devm_request_threaded_irq(dev, irq, NULL,
            irq_thread_fn, IRQF_ONESHOT, "my-chip", g);
if (ret < 0)
      return ret;

/* Get a pointer to the gpio_irq_chip */
girq = &g->gc.irq;
girq->chip = &g->irq;
/* This will let us handle the parent IRQ in the driver */
girq->parent_handler = NULL;
girq->num_parents = 0;
girq->parents = NULL;
girq->default_type = IRQ_TYPE_NONE;
girq->handler = handle_bad_irq;

return devm_gpiochip_add_data(dev, &g->gc, g);
```

The helper supports using hierarchical interrupt controllers as well. In this case the typical set-up will look like this:

```
/* Typical state container with dynamic irqchip */
struct my_gpio {
    struct gpio_chip gc;
    struct irq_chip irq;
    struct fwnode_handle *fwnode;
};
```

```
int irq; /* from platform etc */
struct my_gpio *g;
struct gpio_irq_chip *girq;

/* Set up the irqchip dynamically */
g->irq.name = "my_gpio_irq";
g->irq.irq_ack = my_gpio_ack_irq;
g->irq.irq_mask = my_gpio_mask_irq;
g->irq.irq_unmask = my_gpio_unmask_irq;
g->irq.irq_set_type = my_gpio_set_irq_type;

/* Get a pointer to the gpio_irq_chip */
girq = &g->gc.irq;
girq->chip = &g->irq;
girq->default_type = IRQ_TYPE_NONE;
girq->handler = handle_bad_irq;
girq->fwnode = g->fwnode;
girq->parent_domain = parent;
girq->child_to_parent_hwirq = my_gpio_child_to_parent_hwirq;

return devm_gpiochip_add_data(dev, &g->gc, g);
```

As you can see pretty similar, but you do not supply a parent handler for the IRQ, instead a parent irqdomain, an fwnode for the hardware and a funcion .child_to_parent_hwirq() that has the purpose of looking up the parent hardware irq from a child (i.e. this gpio chip) hardware irq. As always it is good to look at examples in the kernel tree for advice on how to find the required pieces.

If there is a need to exclude certain GPIO lines from the IRQ domain handled by these helpers, we can set .irq.need_valid_mask of the gpiochip before devm_gpiochip_add_data() or gpiochip_add_data() is called. This allocates an .irq.valid_mask with as many bits set as there are GPIO lines in the chip, each bit representing line 0..n-1. Drivers can exclude GPIO lines by clearing bits from this mask. The mask can be filled in the init_valid_mask() callback that is part of the struct gpio_irq_chip.

To use the helpers please keep the following in mind:

- Make sure to assign all relevant members of the struct gpio_chip so that the irqchip can initialize. E.g. .dev and .can_sleep shall be set up properly.
- Nominally set gpio_irq_chip.handler to handle_bad_irq. Then, if your irqchip is cascaded, set the handler to handle_level_irq() and/or handle_edge_irq() in the irqchip .set_type() callback depending on what your controller supports and what is requested by the consumer.

Locking IRQ usage

Since GPIO and irq_chip are orthogonal, we can get conflicts between different use cases. For example a GPIO line used for IRQs should be an input line, it does not make sense to fire interrupts on an output GPIO.

If there is competition inside the subsystem which side is using the resource (a certain GPIO line and register for example) it needs to deny certain operations and keep track of usage inside of the gpiolib subsystem.

Input GPIOs can be used as IRQ signals. When this happens, a driver is requested to mark the GPIO as being used as an IRQ:

```
int gpiochip_lock_as_irq(struct gpio_chip *chip, unsigned int offset)
```

This will prevent the use of non-irq related GPIO APIs until the GPIO IRQ lock is released:

```
void gpiochip_unlock_as_irq(struct gpio_chip *chip, unsigned int offset)
```

When implementing an irqchip inside a GPIO driver, these two functions should typically be called in the .startup() and .shutdown() callbacks from the irqchip.

When using the gpiolib irqchip helpers, these callbacks are automatically assigned.

Disabling and enabling IRQs

In some (fringe) use cases, a driver may be using a GPIO line as input for IRQs, but occasionally switch that line over to drive output and then back to being an input with interrupts again. This happens on things like CEC (Consumer Electronics Control).

When a GPIO is used as an IRQ signal, then gpiolib also needs to know if the IRQ is enabled or disabled. In order to inform gpiolib about this, the irqchip driver should call:

```
void gpiochip_disable_irq(struct gpio_chip *chip, unsigned int offset)
```

This allows drivers to drive the GPIO as an output while the IRQ is disabled. When the IRQ is enabled again, a driver should call:

```
void gpiochip_enable_irq(struct gpio_chip *chip, unsigned int offset)
```

When implementing an irqchip inside a GPIO driver, these two functions should typically be called in the .irq_disable() and .irq_enable() callbacks from the irqchip.

When using the gpiolib irqchip helpers, these callbacks are automatically assigned.

Real-Time compliance for GPIO IRQ chips

Any provider of irqchips needs to be carefully tailored to support Real-Time preemption. It is desirable that all irqchips in the GPIO subsystem keep this in mind and do the proper testing to assure they are real time-enabled.

So, pay attention on above realtime considerations in the documentation.

The following is a checklist to follow when preparing a driver for real-time compliance:

- ensure spinlock_t is not used as part irq_chip implementation
- ensure that sleepable APIs are not used as part irq_chip implementation If sleepable APIs have to be used, these can be done from the .irq_bus_lock() and .irq_bus_unlock() callbacks
- Chained GPIO irqchips: ensure spinlock_t or any sleepable APIs are not used from the chained IRQ handler
- Generic chained GPIO irqchips: take care about generic_handle_irq() calls and apply corresponding work-around
- Chained GPIO irqchips: get rid of the chained IRQ handler and use generic irq handler if possible
- regmap_mmio: it is possible to disable internal locking in regmap by setting .disable_locking and handling the locking in the GPIO driver
- Test your driver with the appropriate in-kernel real-time test cases for both level and edge IRQs

- [1] http://www.spinics.net/lists/linux-omap/msg120425.html
- [2] https://lore.kernel.org/r/1443209283-20781-2-git-send-email-grygorii.strashko@ti.com
- [3] https://lore.kernel.org/r/1443209283-20781-3-git-send-email-grygorii.strashko@ti.com

Requesting self-owned GPIO pins

Sometimes it is useful to allow a GPIO chip driver to request its own GPIO descriptors through the gpiolib API. A GPIO driver can use the following functions to request and free descriptors:

```
struct gpio_desc *gpiochip_request_own_desc(struct gpio_desc *desc,
                                            u16 hwnum,
                                            const char *label,
                                            enum gpiod_flags flags)

void gpiochip_free_own_desc(struct gpio_desc *desc)
```

Descriptors requested with gpiochip_request_own_desc() must be released with gpiochip_free_own_desc().

These functions must be used with care since they do not affect module use count. Do not use the functions to request gpio descriptors not owned by the calling driver.