GPIO Mappings

This document explains how GPIOs can be assigned to given devices and functions.

Note that it only applies to the new descriptor-based interface. For a description of the deprecated integer-based GPIO interface please refer to gpio-legacy.txt (actually, there is no real mapping possible with the old interface; you just fetch an integer from somewhere and request the corresponding GPIO).

All platforms can enable the GPIO library, but if the platform strictly requires GPIO functionality to be present, it needs to select GPIOLIB from its Kconfig. Then, how GPIOs are mapped depends on what the platform uses to describe its hardware layout. Currently, mappings can be defined through device tree, ACPI, and platform data.

Device Tree

GPIOs can easily be mapped to devices and functions in the device tree. The exact way to do it depends on the GPIO controller providing the GPIOs, see the device tree bindings for your controller.

GPIOs mappings are defined in the consumer device's node, in a property named <function>-gpios, where <function> is the function the driver will request through gpiod_get(). For example:

```
foo_device {
        compatible = "acme,foo";
        ...
        led-gpios = <&gpio 15 GPIO_ACTIVE_HIGH>, /* red */
                    <&gpio 16 GPIO_ACTIVE_HIGH>, /* green */
                    <&gpio 17 GPIO_ACTIVE_HIGH>; /* blue */

        power-gpios = <&gpio 1 GPIO_ACTIVE_LOW>;
};
```

Properties named <function>-gpio are also considered valid and old bindings use it but are only supported for compatibility reasons and should not be used for newer bindings since it has been deprecated.

This property will make GPIOs 15, 16 and 17 available to the driver under the "led" function, and GPIO 1 as the "power" GPIO:

```
struct gpio_desc *red, *green, *blue, *power;

red = gpiod_get_index(dev, "led", 0, GPIOD_OUT_HIGH);
green = gpiod_get_index(dev, "led", 1, GPIOD_OUT_HIGH);
blue = gpiod_get_index(dev, "led", 2, GPIOD_OUT_HIGH);

power = gpiod_get(dev, "power", GPIOD_OUT_HIGH);
```

The led GPIOs will be active high, while the power GPIO will be active low (i.e. gpiod_is_active_low(power) will be true).

The second parameter of the gpiod_get() functions, the con_id string, has to be the <function>-prefix of the GPIO suffixes ("gpios" or "gpio", automatically looked up by the gpiod functions internally) used in the device tree. With above "led-gpios" example, use the prefix without the "-" as con_id parameter: "led".

Internally, the GPIO subsystem prefixes the GPIO suffix ("gpios" or "gpio") with the string passed in con_id to get the resulting string (`snprintf(... "%s-%s", con_id, gpio_suffixes[]`).

ACPI

ACPI also supports function names for GPIOs in a similar fashion to DT. The above DT example can be converted to an equivalent ACPI description with the help of _DSD (Device Specific Data), introduced in ACPI 5.1:

```
Device (FOO) {
        Name (_CRS, ResourceTemplate () {
                GpioIo (Exclusive, PullUp, 0, 0, IoRestrictionOutputOnly,
                        "\\_SB.GPI0", 0, ResourceConsumer) { 15 } // red
                GpioIo (Exclusive, PullUp, 0, 0, IoRestrictionOutputOnly,
                        "\\_SB.GPI0", 0, ResourceConsumer) { 16 } // green
                GpioIo (Exclusive, PullUp, 0, 0, IoRestrictionOutputOnly,
                        "\\_SB.GPI0", 0, ResourceConsumer) { 17 } // blue
                GpioIo (Exclusive, PullNone, 0, 0, IoRestrictionOutputOnly,
                        "\\_SB.GPI0", 0, ResourceConsumer) { 1 } // power
        })

        Name (_DSD, Package () {
                ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
                Package () {
                        Package () {
                                "led-gpios",
                                Package () {
```

```
                                        ^FOO, 0, 0, 1,
                                        ^FOO, 1, 0, 1,
                                        ^FOO, 2, 0, 1,
                                }
                        },
                        Package () { "power-gpios", Package () { ^FOO, 3, 0, 0 } },
                }
        })
}
```

For more information about the ACPI GPIO bindings see Documentation/firmware-guide/acpi/gpio-properties.rst.

Platform Data

Finally, GPIOs can be bound to devices and functions using platform data. Board files that desire to do so need to include the following header:

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

GPIOs are mapped by the means of tables of lookups, containing instances of the gpiod_lookup structure. Two macros are defined to help declaring such mappings:

```
GPIO_LOOKUP(key, chip_hwnum, con_id, flags)
GPIO_LOOKUP_IDX(key, chip_hwnum, con_id, idx, flags)
```

where

- key is either the label of the gpiod_chip instance providing the GPIO, or the GPIO line name
- chip_hwnum is the hardware number of the GPIO within the chip, or U16_MAX to indicate that key is a GPIO line name
- con_id is the name of the GPIO function from the device point of view. It can be NULL, in which case it will match any function.
- idx is the index of the GPIO within the function.
- flags is defined to specify the following properties:
    - GPIO_ACTIVE_HIGH - GPIO line is active high
    - GPIO_ACTIVE_LOW - GPIO line is active low
    - GPIO_OPEN_DRAIN - GPIO line is set up as open drain
    - GPIO_OPEN_SOURCE - GPIO line is set up as open source
    - GPIO_PERSISTENT - GPIO line is persistent during suspend/resume and maintains its value
    - GPIO_TRANSITORY - GPIO line is transitory and may loose its electrical state during suspend/resume

In the future, these flags might be extended to support more properties.

Note that:

1. GPIO line names are not guaranteed to be globally unique, so the first match found will be used.
2. GPIO_LOOKUP() is just a shortcut to GPIO_LOOKUP_IDX() where idx = 0.

A lookup table can then be defined as follows, with an empty entry defining its end. The 'dev_id' field of the table is the identifier of the device that will make use of these GPIOs. It can be NULL, in which case it will be matched for calls to gpiod_get() with a NULL device.

```
struct gpiod_lookup_table gpios_table = {
        .dev_id = "foo.0",
        .table = {
                GPIO_LOOKUP_IDX("gpio.0", 15, "led", 0, GPIO_ACTIVE_HIGH),
                GPIO_LOOKUP_IDX("gpio.0", 16, "led", 1, GPIO_ACTIVE_HIGH),
                GPIO_LOOKUP_IDX("gpio.0", 17, "led", 2, GPIO_ACTIVE_HIGH),
                GPIO_LOOKUP("gpio.0", 1, "power", GPIO_ACTIVE_LOW),
                { },
        },
};
```

And the table can be added by the board code as follows:

```
gpiod_add_lookup_table(&gpios_table);
```

The driver controlling "foo.0" will then be able to obtain its GPIOs as follows:

```
struct gpio_desc *red, *green, *blue, *power;

red = gpiod_get_index(dev, "led", 0, GPIOD_OUT_HIGH);
```

```
green = gpiod_get_index(dev, "led", 1, GPIOD_OUT_HIGH);
blue = gpiod_get_index(dev, "led", 2, GPIOD_OUT_HIGH);

power = gpiod_get(dev, "power", GPIOD_OUT_HIGH);
```

Since the "led" GPIOs are mapped as active-high, this example will switch their signals to 1, i.e. enabling the LEDs. And for the "power" GPIO, which is mapped as active-low, its actual signal will be 0 after this code. Contrary to the legacy integer GPIO interface, the active-low property is handled during mapping and is thus transparent to GPIO consumers.

A set of functions such as gpiod_set_value() is available to work with the new descriptor-oriented interface.

Boards using platform data can also hog GPIO lines by defining GPIO hog tables.

```
struct gpiod_hog gpio_hog_table[] = {
        GPIO_HOG("gpio.0", 10, "foo", GPIO_ACTIVE_LOW, GPIOD_OUT_HIGH),
        { }
};
```

And the table can be added to the board code as follows:

```
gpiod_add_hogs(gpio_hog_table);
```

The line will be hogged as soon as the gpiochip is created or - in case the chip was created earlier - when the hog table is registered.

Arrays of pins

In addition to requesting pins belonging to a function one by one, a device may also request an array of pins assigned to the function. The way those pins are mapped to the device determines if the array qualifies for fast bitmap processing. If yes, a bitmap is passed over get/set array functions directly between a caller and a respective .get/set_multiple() callback of a GPIO chip.

In order to qualify for fast bitmap processing, the array must meet the following requirements:

- pin hardware number of array member 0 must also be 0,
- pin hardware numbers of consecutive array members which belong to the same chip as member 0 does must also match their array indexes.

Otherwise fast bitmap processing path is not used in order to avoid consecutive pins which belong to the same chip but are not in hardware order being processed separately.

If the array applies for fast bitmap processing path, pins which belong to different chips than member 0 does, as well as those with indexes different from their hardware pin numbers, are excluded from the fast path, both input and output. Moreover, open drain and open source pins are excluded from fast bitmap output processing.