GPIO Descriptor Consumer Interface

This document describes the consumer interface of the GPIO framework. Note that it describes the new descriptor-based interface. For a description of the deprecated integer-based GPIO interface please refer to gpio-legacy.txt.

Guidelines for GPIOs consumers

Drivers that can't work without standard GPIO calls should have Kconfig entries that depend on GPIOLIB or select GPIOLIB. The functions that allow a driver to obtain and use GPIOs are available by including the following file:

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

There are static inline stubs for all functions in the header file in the case where GPIOLIB is disabled. When these stubs are called they will emit warnings. These stubs are used for two use cases:

- Simple compile coverage with e.g. COMPILE_TEST - it does not matter that the current platform does not enable or select GPIOLIB because we are not going to execute the system anyway.
- Truly optional GPIOLIB support - where the driver does not really make use of the GPIOs on certain compile-time configurations for certain systems, but will use it under other compile-time configurations. In this case the consumer must make sure not to call into these functions, or the user will be met with console warnings that may be perceived as intimidating.

All the functions that work with the descriptor-based GPIO interface are prefixed with `gpiod_`. The `gpio_` prefix is used for the legacy interface. No other function in the kernel should use these prefixes. The use of the legacy functions is strongly discouraged, new code should use <linux/gpio/consumer.h> and descriptors exclusively.

Obtaining and Disposing GPIOs

With the descriptor-based interface, GPIOs are identified with an opaque, non-forgeable handler that must be obtained through a call to one of the gpiod_get() functions. Like many other kernel subsystems, gpiod_get() takes the device that will use the GPIO and the function the requested GPIO is supposed to fulfill:

```
struct gpio_desc *gpiod_get(struct device *dev, const char *con_id,
                            enum gpiod_flags flags)
```

If a function is implemented by using several GPIOs together (e.g. a simple LED device that displays digits), an additional index argument can be specified:

```
struct gpio_desc *gpiod_get_index(struct device *dev,
                                  const char *con_id, unsigned int idx,
                                  enum gpiod_flags flags)
```

For a more detailed description of the con_id parameter in the DeviceTree case see Documentation/driver-api/gpio/board.rst

The flags parameter is used to optionally specify a direction and initial value for the GPIO. Values can be:

- GPIOD_ASIS or 0 to not initialize the GPIO at all. The direction must be set later with one of the dedicated functions.
- GPIOD_IN to initialize the GPIO as input.
- GPIOD_OUT_LOW to initialize the GPIO as output with a value of 0.
- GPIOD_OUT_HIGH to initialize the GPIO as output with a value of 1.
- GPIOD_OUT_LOW_OPEN_DRAIN same as GPIOD_OUT_LOW but also enforce the line to be electrically used with open drain.
- GPIOD_OUT_HIGH_OPEN_DRAIN same as GPIOD_OUT_HIGH but also enforce the line to be electrically used with open drain.

Note that the initial value is *logical* and the physical line level depends on whether the line is configured active high or active low (see :ref:`active_low_semantics`).

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\gpio\(linux-master)(Documentation)(driver-api)(gpio)consumer.rst, line 75); *backlink*
>
> Unknown interpreted text role "ref".

The two last flags are used for use cases where open drain is mandatory, such as I2C: if the line is not already configured as open drain in the mappings (see board.txt), then open drain will be enforced anyway and a warning will be printed that the board configuration needs to be updated to match the use case.

Both functions return either a valid GPIO descriptor, or an error code checkable with IS_ERR() (they will never return a NULL pointer). -ENOENT will be returned if and only if no GPIO has been assigned to the device/function/index triplet, other error codes are used for cases where a GPIO has been assigned but an error occurred while trying to acquire it. This is useful to discriminate

between mere errors and an absence of GPIO for optional GPIO parameters. For the common pattern where a GPIO is optional, the gpiod_get_optional() and gpiod_get_index_optional() functions can be used. These functions return NULL instead of -ENOENT if no GPIO has been assigned to the requested function:

```
struct gpio_desc *gpiod_get_optional(struct device *dev,
                                     const char *con_id,
                                     enum gpiod_flags flags)

struct gpio_desc *gpiod_get_index_optional(struct device *dev,
                                           const char *con_id,
                                           unsigned int index,
                                           enum gpiod_flags flags)
```

Note that gpio_get*_optional() functions (and their managed variants), unlike the rest of gpiolib API, also return NULL when gpiolib support is disabled. This is helpful to driver authors, since they do not need to special case -ENOSYS return codes. System integrators should however be careful to enable gpiolib on systems that need it.

For a function using multiple GPIOs all of those can be obtained with one call:

```
struct gpio_descs *gpiod_get_array(struct device *dev,
                                   const char *con_id,
                                   enum gpiod_flags flags)
```

This function returns a struct gpio_descs which contains an array of descriptors. It also contains a pointer to a gpiolib private structure which, if passed back to get/set array functions, may speed up I/O proocessing:

```
struct gpio_descs {
        struct gpio_array *info;
        unsigned int ndescs;
        struct gpio_desc *desc[];
}
```

The following function returns NULL instead of -ENOENT if no GPIOs have been assigned to the requested function:

```
struct gpio_descs *gpiod_get_array_optional(struct device *dev,
                                            const char *con_id,
                                            enum gpiod_flags flags)
```

Device-managed variants of these functions are also defined:

```
struct gpio_desc *devm_gpiod_get(struct device *dev, const char *con_id,
                                 enum gpiod_flags flags)

struct gpio_desc *devm_gpiod_get_index(struct device *dev,
                                       const char *con_id,
                                       unsigned int idx,
                                       enum gpiod_flags flags)

struct gpio_desc *devm_gpiod_get_optional(struct device *dev,
                                          const char *con_id,
                                          enum gpiod_flags flags)

struct gpio_desc *devm_gpiod_get_index_optional(struct device *dev,
                                                const char *con_id,
                                                unsigned int index,
                                                enum gpiod_flags flags)

struct gpio_descs *devm_gpiod_get_array(struct device *dev,
                                        const char *con_id,
                                        enum gpiod_flags flags)

struct gpio_descs *devm_gpiod_get_array_optional(struct device *dev,
                                                 const char *con_id,
                                                 enum gpiod_flags flags)
```

A GPIO descriptor can be disposed of using the gpiod_put() function:

```
void gpiod_put(struct gpio_desc *desc)
```

For an array of GPIOs this function can be used:

```
void gpiod_put_array(struct gpio_descs *descs)
```

It is strictly forbidden to use a descriptor after calling these functions. It is also not allowed to individually release descriptors (using gpiod_put()) from an array acquired with gpiod_get_array().

The device-managed variants are, unsurprisingly:

```
void devm_gpiod_put(struct device *dev, struct gpio_desc *desc)

void devm_gpiod_put_array(struct device *dev, struct gpio_descs *descs)
```

Using GPIOs

Setting Direction

The first thing a driver must do with a GPIO is setting its direction. If no direction-setting flags have been given to gpiod_get*(), this is done by invoking one of the gpiod_direction_*() functions:

```
int gpiod_direction_input(struct gpio_desc *desc)
int gpiod_direction_output(struct gpio_desc *desc, int value)
```

The return value is zero for success, else a negative errno. It should be checked, since the get/set calls don't return errors and since misconfiguration is possible. You should normally issue these calls from a task context. However, for spinlock-safe GPIOs it is 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.

A driver can also query the current direction of a GPIO:

```
int gpiod_get_direction(const struct gpio_desc *desc)
```

This function returns 0 for output, 1 for input, or an error code in case of error.

Be aware that there is no default direction for GPIOs. Therefore, **using a GPIO without setting its direction first is illegal and will result in undefined behavior!**

Spinlock-Safe GPIO Access

Most GPIO controllers can be accessed with memory read/write instructions. Those don't need to sleep, and can safely be done from inside hard (non-threaded) IRQ handlers and similar contexts.

Use the following calls to access GPIOs from an atomic context:

```
int gpiod_get_value(const struct gpio_desc *desc);
void gpiod_set_value(struct gpio_desc *desc, 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 do not return errors because "invalid GPIO" should have been reported earlier from gpiod_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.

GPIO Access That May Sleep

Some GPIO controllers must be accessed using message based buses 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:

```
int gpiod_cansleep(const struct gpio_desc *desc)
```

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

```
int gpiod_get_value_cansleep(const struct gpio_desc *desc)
void gpiod_set_value_cansleep(struct gpio_desc *desc, int value)
```

Accessing such GPIOs requires a context which may sleep, for example a threaded IRQ handler, and those accessors must be used instead of spinlock-safe accessors without the cansleep() name suffix.

Other than the fact that these accessors might sleep, and will work on GPIOs that can't be accessed from hardIRQ handlers, these calls act the same as the spinlock-safe calls.

The active low and open drain semantics

As a consumer should not have to care about the physical line level, all of the gpiod_set_value_xxx() or gpiod_set_array_value_xxx() functions operate with the *logical* value. With this they take the active low property into account. This means that they check whether the GPIO is configured to be active low, and if so, they manipulate the passed value before the physical line level is driven.

The same is applicable for open drain or open source output lines: those do not actively drive their output high (open drain) or low (open source), they just switch their output to a high impedance value. The consumer should not need to care. (For details read about open drain in driver.txt.)

With this, all the gpiod_set_(array)_value_xxx() functions interpret the parameter "value" as "asserted" ("1") or "de-asserted" ("0"). The physical line level will be driven accordingly.

As an example, if the active low property for a dedicated GPIO is set, and the gpiod_set_(array)_value_xxx() passes "asserted" ("1"), the physical line level will be driven low.

To summarize:

```
Function (example)                 line property          physical line
gpiod_set_raw_value(desc, 0);      don't care             low
gpiod_set_raw_value(desc, 1);      don't care             high
gpiod_set_value(desc, 0);          default (active high)  low
gpiod_set_value(desc, 1);          default (active high)  high
gpiod_set_value(desc, 0);          active low             high
gpiod_set_value(desc, 1);          active low             low
gpiod_set_value(desc, 0);          open drain             low
gpiod_set_value(desc, 1);          open drain             high impedance
gpiod_set_value(desc, 0);          open source            high impedance
gpiod_set_value(desc, 1);          open source            high
```

It is possible to override these semantics using the set_raw/get_raw functions but it should be avoided as much as possible, especially by system-agnostic drivers which should not need to care about the actual physical line level and worry about the logical value instead.

Accessing raw GPIO values

Consumers exist that need to manage the logical state of a GPIO line, i.e. the value their device will actually receive, no matter what lies between it and the GPIO line.

The following set of calls ignore the active-low or open drain property of a GPIO and work on the raw line value:

```
int gpiod_get_raw_value(const struct gpio_desc *desc)
void gpiod_set_raw_value(struct gpio_desc *desc, int value)
int gpiod_get_raw_value_cansleep(const struct gpio_desc *desc)
void gpiod_set_raw_value_cansleep(struct gpio_desc *desc, int value)
int gpiod_direction_output_raw(struct gpio_desc *desc, int value)
```

The active low state of a GPIO can also be queried and toggled using the following calls:

```
int gpiod_is_active_low(const struct gpio_desc *desc)
void gpiod_toggle_active_low(struct gpio_desc *desc)
```

Note that these functions should only be used with great moderation; a driver should not have to care about the physical line level or open drain semantics.

Access multiple GPIOs with a single function call

The following functions get or set the values of an array of GPIOs:

```
int gpiod_get_array_value(unsigned int array_size,
                          struct gpio_desc **desc_array,
                          struct gpio_array *array_info,
                          unsigned long *value_bitmap);
int gpiod_get_raw_array_value(unsigned int array_size,
                              struct gpio_desc **desc_array,
                              struct gpio_array *array_info,
                              unsigned long *value_bitmap);
int gpiod_get_array_value_cansleep(unsigned int array_size,
                                   struct gpio_desc **desc_array,
                                   struct gpio_array *array_info,
                                   unsigned long *value_bitmap);
int gpiod_get_raw_array_value_cansleep(unsigned int array_size,
                                   struct gpio_desc **desc_array,
                                   struct gpio_array *array_info,
                                   unsigned long *value_bitmap);

int gpiod_set_array_value(unsigned int array_size,
                          struct gpio_desc **desc_array,
                          struct gpio_array *array_info,
                          unsigned long *value_bitmap)
int gpiod_set_raw_array_value(unsigned int array_size,
                              struct gpio_desc **desc_array,
                              struct gpio_array *array_info,
                              unsigned long *value_bitmap)
int gpiod_set_array_value_cansleep(unsigned int array_size,
                                   struct gpio_desc **desc_array,
                                   struct gpio_array *array_info,
                                   unsigned long *value_bitmap)
int gpiod_set_raw_array_value_cansleep(unsigned int array_size,
                                       struct gpio_desc **desc_array,
                                       struct gpio_array *array_info,
                                       unsigned long *value_bitmap)
```

The array can be an arbitrary set of GPIOs. The functions will try to access GPIOs belonging to the same bank or chip simultaneously if supported by the corresponding chip driver. In that case a significantly improved performance can be expected. If simultaneous access is not possible the GPIOs will be accessed sequentially.

The functions take four arguments:

- array_size - the number of array elements
- desc_array - an array of GPIO descriptors
- array_info - optional information obtained from gpiod_get_array()
- value_bitmap - a bitmap to store the GPIOs' values (get) or a bitmap of values to assign to the GPIOs (set)

The descriptor array can be obtained using the gpiod_get_array() function or one of its variants. If the group of descriptors returned by that function matches the desired group of GPIOs, those GPIOs can be accessed by simply using the struct gpio_descs returned by gpiod_get_array():

```
struct gpio_descs *my_gpio_descs = gpiod_get_array(...);
gpiod_set_array_value(my_gpio_descs->ndescs, my_gpio_descs->desc,
                      my_gpio_descs->info, my_gpio_value_bitmap);
```

It is also possible to access a completely arbitrary array of descriptors. The descriptors may be obtained using any combination of gpiod_get() and gpiod_get_array(). Afterwards the array of descriptors has to be setup manually before it can be passed to one of the above functions. In that case, array_info should be set to NULL.

Note that for optimal performance GPIOs belonging to the same chip should be contiguous within the array of descriptors.

Still better performance may be achieved if array indexes of the descriptors match hardware pin numbers of a single chip. If an array passed to a get/set array function matches the one obtained from gpiod_get_array() and array_info associated with the array is also passed, the function may take a fast bitmap processing path, passing the value_bitmap argument directly to the respective .get/set_multiple() callback of the chip. That allows for utilization of GPIO banks as data I/O ports without much loss of performance.

The return value of gpiod_get_array_value() and its variants is 0 on success or negative on error. Note the difference to gpiod_get_value(), which returns 0 or 1 on success to convey the GPIO value. With the array functions, the GPIO values are stored in value_array rather than passed back as return value.

GPIOs mapped to IRQs

GPIO lines can quite often be used as IRQs. You can get the IRQ number corresponding to a given GPIO using the following call:

```
int gpiod_to_irq(const struct gpio_desc *desc)
```

It will return an IRQ number, or a negative errno code if the mapping can't be done (most likely because that particular GPIO cannot be used as IRQ). It is an unchecked error to use a GPIO that wasn't set up as an input using gpiod_direction_input(), or to use an IRQ number that didn't originally come from gpiod_to_irq(). gpiod_to_irq() is not allowed to sleep.

Non-error values returned from gpiod_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.

GPIOs and ACPI

On ACPI systems, GPIOs are described by GpioIo()/GpioInt() resources listed by the _CRS configuration objects of devices. Those resources do not provide connection IDs (names) for GPIOs, so it is necessary to use an additional mechanism for this purpose.

Systems compliant with ACPI 5.1 or newer may provide a _DSD configuration object which, among other things, may be used to provide connection IDs for specific GPIOs described by the GpioIo()/GpioInt() resources in _CRS. If that is the case, it will be handled by the GPIO subsystem automatically. However, if the _DSD is not present, the mappings between GpioIo()/GpioInt() resources and GPIO connection IDs need to be provided by device drivers.

For details refer to Documentation/firmware-guide/acpi/gpio-properties.rst

Interacting With the Legacy GPIO Subsystem

Many kernel subsystems and drivers still handle GPIOs using the legacy integer-based interface. It is strongly recommended to update these to the new gpiod interface. For cases where both interfaces need to be used, the following two functions allow to convert a GPIO descriptor into the GPIO integer namespace and vice-versa:

```
int desc_to_gpio(const struct gpio_desc *desc)
struct gpio_desc *gpio_to_desc(unsigned gpio)
```

The GPIO number returned by desc_to_gpio() can safely be used as a parameter of the gpio_*() functions for as long as the GPIO descriptor *desc* is not freed. All the same, a GPIO number passed to gpio_to_desc() must first be properly acquired using e.g. gpio_request_one(), and the returned GPIO descriptor is only considered valid until that GPIO number is released using gpio_free().

Freeing a GPIO obtained by one API with the other API is forbidden and an unchecked error.