GPIO Generic API

v2.x.x | v1.6.0 — v1.9.1

To interact with the General-Purpose Input/Output (GPIO) peripheral, we can use the generic API `<drivers/gpio.h>`, which provides user-friendly functions to interact with GPIO peripherals. The GPIO peripheral can be used to interact with a variety of external components such as switches, buttons, and LEDs.

When using any driver in Zephyr, the first step is to initialize it by retrieving the device pointer. For a GPIO pin, the first necessary step after that is to configure the pin to be either an input or an output pin. Then you can write to an output pin or read from an input pin. In the following paragraphs, these four steps will be covered in detail.

Initializing the API

Some of the generic APIs in Zephyr have API-specific structs that contain the previously mentioned device pointer, as well as some other information about the device. In the GPIO API, this is the structure [`gpio_dt_spec`]. This structure has the device pointer `const struct device * port`, as well as the pin number on the device, `gpio_pin_t pin`, and the device's configuration flags, `gpio_dt_flags_t dt_flags`.

struct **gpio_dt_spec**

#include <gpio.h>

Container for GPIO pin information specified in devicetree.

This type contains a pointer to a GPIO device, pin number for a pin controlled by that device, and the subset of pin configuration flags which may be given in devicetree.

Public Members

const struct device * **port**

GPIO device controlling the pin

gpio_pin_t **pin**

The pin's number on the device

gpio_dt_flags_t **dt_flags**

The pin's configuration flags as specified in devicetree

To retrieve this structure, we need to use the API-specific function `GPIO_DT_SPEC_GET()`, which has the following signature:

GPIO_DT_SPEC_GET(node_id, prop)

Equivalent to GPIO_DT_SPEC_GET_BY_IDX(node_id, prop, 0).

See also

GPIO_DT_SPEC_GET_BY_IDX()

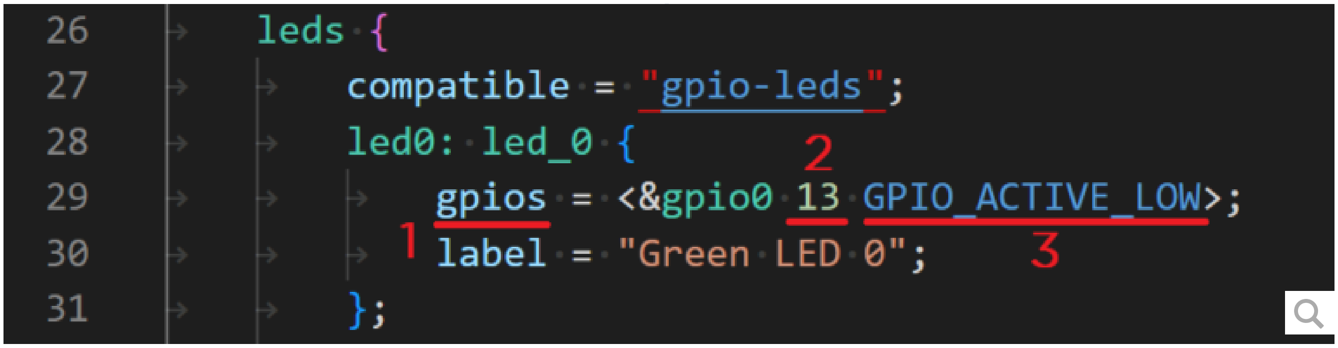
Parameters:
- **node_id** – devicetree node identifier
- **prop** – lowercase-and-underscores property name

Returns: static initializer for a struct gpio_dt_spec for the property

Similar to `DEVICE_DT_GET()`, `GPIO_DT_SPEC_GET()` also takes the devicetree node identifier. It also takes the property name of the node. The function will return a variable of type `gpio_dt_spec`, containing the device pointer as well as the pin number and configuration flags.

The advantage of this API-specific structure is that it encapsulates all the information needed to use the device in a single variable, instead of having to extract it from the devicetree line by line.

Let's take `led_0` as an example, which has the devicetree implementation shown below:



From the image above, we can see that the property containing all this information is called `gpios`, and is the property name to pass to `GPIO_DT_SPEC_GET()`:


```
struct gpio_dt_spec led = GPIO_DT_SPEC_GET(DT_NODELABEL(led0), gpios,,
```

This function will return a struct of type `gpio_dt_spec` with the device pointer for the GPIO controller, `&gpio0`, the pin number `led.pin = 13` and the flag `led.dt_flags = GPIO_ACTIVE_LOW`.

Before using the device pointer contained in `gpio_dt_spec led`, we need to check if it's ready using [`device_is_ready()`].


```
if (!device_is_ready(led.port)) {
        return;
}
```

Configure a single pin

This is done by calling the function [`gpio_pin_configure_dt()`], which has the following signature:

static inline int **gpio_pin_configure_dt**(*const struct* gpio_dt_spec *spec, gpio_flags_t extra_flags)

Configure a single pin from a `gpio_dt_spec` and some extra flags.

This is equivalent to:

```
gpio_pin_configure(spec->port, spec->pin, spec->dt_flags | extra_flags);
```

Parameters:
- **spec** – GPIO specification from devicetree
- **extra_flags** – additional flags

Returns: a value from gpio_pin_configure()

With this function, you can configure a pin to be an input `GPIO_INPUT` or an output `GPIO_OUTPUT` through the second parameter `flags` as shown in the examples below.

The following line configures the pin associated with `gpio_dt_spec led`, which can be denoted as `led.pin`, as an output pin:

Copy

```
gpio_pin_configure_dt(&led, GPIO_OUTPUT);
```

You can also specify other hardware characteristics to a pin like the drive strength, pull up/pull down resistors, active high or active low. Different hardware characteristics can be combined through the | operator. Again, this is done using the parameter `flags`.

The following line configures the pin `led.pin` as an output that is active low.

Copy

```
gpio_pin_configure_dt(&led, GPIO_OUTPUT | GPIO_ACTIVE_LOW);
```

All GPIO `flags` are documented here.

Write to an output pin

Writing to an output pin is straightforward by using the function `gpio_pin_set_dt()`, which has the following signature:

static inline int gpio_pin_set_dt(const struct gpio_dt_spec *spec, int value)

Set logical level of a output pin from a `gpio_dt_spec`.

This is equivalent to:

```
gpio_pin_set(spec->port, spec->pin, value);
```

Parameters:
- **spec** – GPIO specification from devicetree
- **value** – Value assigned to the pin.

Returns: a value from gpio_pin_set()

For example, the following line sets the pin associated with `gpio_dt_spec led`, which can be denoted as `led.pin`, to logic 1 “active state”:

Copy

```
gpio_pin_set_dt(&led, 1);
```

For instance, for the node `led_0` on the nRF52833DK, this would set pin `13` to logic 1 “active state”.

You can also use the gpio_pin_toggle_dt() function to toggle an output pin.

static inline int gpio_pin_toggle_dt(const struct gpio_dt_spec *spec)

Toggle pin level from a `gpio_dt_spec`.

This is equivalent to:

```
gpio_pin_toggle(spec->port, spec->pin);
```

Parameters:
- **spec** – GPIO specification from devicetree

Returns: a value from gpio_pin_toggle()

Read from an input pin

Reading a pin configured as an input is not as straightforward as writing to a pin configured as an output. There are two possible methods to read the status of an input pin:

Polling method

Polling means continuously reading the status of the pin to check if it has changed. To read the current status of a pin, all you need to do is to call the function `gpio_pin_get_dt()`, which has the following signature:

```
static inline int gpio_pin_get_dt(const struct gpio_dt_spec *spec)
```

Get logical level of an input pin from a `gpio_dt_spec`.

This is equivalent to:

```
gpio_pin_get(spec->port, spec->pin);
```

Parameters: • **spec** – GPIO specification from devicetree

Returns: a value from gpio_pin_get()

For example, the following line reads the current status of `led.pin` saves it in a variable called `val`.

Copy

```
val = gpio_pin_get_dt(&led);
```

The drawback of the polling method is that you have to repeatedly call `gpio_pin_get_dt()` to keep track of the status of a pin. This is usually not optimal from performance and power perspectives as it requires the CPU's constant attention. It's a simple method, yet not power-efficient.

We will use this method in Exercise 1 of this lesson for demonstration purposes.

Interrupt method

Note

> You can only configure an interrupt on a GPIO pin configured as an input.

The following are the general steps needed to set up an interrupt on a GPIO pin.

1. Configure the interrupt on a pin.

This is done by calling the function `gpio_pin_interrupt_configure_dt()`, which has the signature shown below:

static inline int gpio_pin_interrupt_configure_dt(const struct gpio_dt_spec *spec, gpio_flags_t flags)

Configure pin interrupts from a `gpio_dt_spec`.

This is equivalent to:

```
gpio_pin_interrupt_configure(spec->port, spec->pin, flags);
```

The `spec->dt_flags` value is not used.

Parameters:
- **spec** – GPIO specification from devicetree
- **flags** – interrupt configuration flags

Returns: a value from gpio_pin_interrupt_configure()

Through the second parameter `flags`, you can configure whether you want to trigger the interrupt on rising edge, falling edge, or both. Or change to logical level 1, logical level 0, or both.

The following line will configure an interrupt on `dev.pin` on the change to logical level 1.

Copy

```
gpio_pin_interrupt_configure_dt(&button,GPIO_INT_EDGE_TO_ACTIVE);
```

All interrupt flag options are documented here.

Definition

Callback function: Also known as an interrupt handler or an Interrupt Service Routine (ISR). It runs asynchronously in response to a hardware or software interrupt. In general, ISRs have higher priority than all threads (covered in Lesson 7). It preempts the execution of the current thread, allowing an action to take place immediately. Thread execution resumes only once all ISR work has been completed.

The signature (prototype) of the callback function is shown below:

Copy

```
void pin_isr(const struct device *dev, struct gpio_callback *cb,
gpio_port_pins_t pins);
```

What you put inside the body of an ISR is highly application-dependent. For instance, the following ISR toggles a LED every time the interrupt is triggered.

Copy

```
void pin_isr(const struct device *dev, struct gpio_callback *cb,
uint32_t pins)
{
        gpio_pin_toggle_dt(&led);
}
```

3. Define a variable of type `static struct` [`gpio_callback`]() as shown in the code line below.

Copy

```
static struct gpio_callback pin_cb_data;
```

The `pin_cb_data` gpio callback variable will hold information such as the pin number and the function to be called when an interrupt occurs (callback function).

4. Initialize the gpio callback variable `pin_cb_data` using `gpio_init_callback()`.

This `gpio_callback` struct variable stores the address of the callback function and the `static inline void gpio_init_callback(struct gpio_callback *callback, gpio_callback_handler_t handler, gpio_port_pins_t pin_mask)`

Helper to initialize a struct gpio_callback properly.

Parameters:
- **callback** – A valid Application's callback structure pointer.
- **handler** – A valid handler function pointer.
- **pin_mask** – A bit mask of relevant pins for the handler

For example, the following line will initialize the `pin_cb_data` variable with the callback function `pin_isr` and the bit mask of pin `dev.pin`. Note the use of the macro `BIT(n)`, which simply gets an unsigned integer with bit position `n` set.

Copy

```
gpio_init_callback(&pin_cb_data, pin_isr, BIT(dev.pin));
```

5. The final step is to add the callback function through the function [`gpio_add_callback()`].

`static inline int gpio_add_callback(const struct device *port, struct gpio_callback *callback)`

Add an application callback.

Note: enables to add as many callback as needed on the same port.

Note

Callbacks may be added to the device from within a callback handler invocation, but whether they are invoked for the current GPIO event is not specified.

Parameters:
- **port** – Pointer to the device structure for the driver instance.
- **callback** – A valid Application's callback structure pointer.

Returns: 0 if successful, negative errno code on failure.

For example, the following line adds the callback function that we set up in the previous steps.

Copy

```
gpio_add_callback(button.port, &pin_cb_data);
```

The full API documentation for GPIO generic interface is available [here]