Table of Contents

1. Laboratory on GPIO use in Zephyr OS 2
1.1. Concept of the Device Tree 2
1.2. Blinky - how to use the LED 4
1.3. Direct register use 5
1.4. Button example 6
1.5. Tasks for the laboratory on GPIO 7

1. Laboratory on GPIO use in Zephyr OS

GPIOs in Nordic Semiconductor nRF52840 can be handled by software with polling, with interrupts or using direct hardware connection with PPI (Programmable Peripheral Interconnect). In the first part of the laboratory, we will handle GPIO events with the software. The first example named “blinky” [1)] is a simple program that blinks the onboard LED. It can be used to check if all settings are set properly, and if all hardware and software modules work. After compilation and uploading of the program one LED on the board (LED0) should blink. Using this example we can analyze the way of setting parameters and using the GPIO input ports in Zephyr RTOS. Blinky is accessible as the example in the PlatformIO set of examples for the Nordic nRF52 platform as “zephyr-blink”.

1.1. Concept of the Device Tree

In Zephyr RTOS the device tree [2)] concept is used to describe the hardware modules and their properties. The device tree file (*.dts) is specific for every board. Another file (*.dtsi) describes in detail the hardware peripherals built into the specific microchip. It contains names (labels) and addresses of registers of all peripherals in the SoC.

Let's focus on the device tree entry describing the LED connected to some GPIO output. This entry exists in the file that describes the board. In our case, it's the file for nRF52840DK located at the path:

```
C:\...\.platformio\packages\frameworkzephyr\boards\arm\nrf52840dk_nrf52840\nrf52840dk_nrf52840.dts
```

In the file, we can find the description of four LEDs. Let's show here one of them:

```
leds {
        compatible = "gpio-leds";
        led0: led_0 {
            gpios = <&gpio0 13 GPIO_ACTIVE_LOW>;
            label = "Green LED 0";
        };
             /*... details of other LEDs are here */
      };
```

The **compatible** key specifies compatibility with the hardware device. During the build system tries to bind the entry with the *.yaml file that describes the hardware from a specific vendor. In our example, we use a generic class of LED devices which do not have a specific vendor **gpio-leds**.

- **led0** is the node label that we can use to access the details of the device.
- **led_0** is the child node of gpio-leds. Here it has two properties: gpios and label.
- **gpios** property is required for gpio-leds class. It specifies the gpio port (here gpio0), bit number (here 13), and flags that define the behaviour of the GPIO (here active low output).
- **label** property is optional and contains human understandable name of the device.

Similarly, the buttons are described in the device tree file:

```
buttons {
        compatible = "gpio-keys";
        button0: button_0 {
            gpios = <&gpio0 11 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;
            label = "Push button switch 0";
        };
             /*... details of other buttons are here */
        };
```

The generic class of button devices is **gpio-keys**. Other options are similar to LED but the flags. For the button, they additionally enable a pull-up resistor with GPIO_PULL_UP value. For better compatibility, it is possible to define the aliases of the entries. In the *.dts file there are aliases defined for leds and buttons.

```
/* These aliases are provided for compatibility with samples */
aliases {
    led0 = &led0;
    led1 = &led1;
    ...
    sw0 = &button0;
    sw1 = &button1;
    sw2 = &button2;
    sw3 = &button3;
};
```

Notice that the description in the *.dts file does not specify the details of the internal peripheral built into SoC e.g. the addresses of the registers. Such information can be found in the *.dtsi file which holds the description of the chip itself. The file is located in the path:

```
C:\...\.platformio\packages\framework-zephyr\dts\arm\nordic\nrf52840.dtsi
```

The entry for gpio0 which is referred from leds and buttons entries looks like this:

```
        gpio0: gpio@50000000 {
            compatible = "nordic,nrf-gpio";
            gpio-controller;
            reg = <0x50000000 0x200
                   0x50000500 0x300>;
            #gpio-cells = <2>;
            label = "GPIO_0";
            status = "disabled";
            port = <0>;
        };
```

Bindings of the entries are defined in YAML files. The flags property is defined as one of the #gpio-cells in the file:

```
C:\...\.platformio\packages\framework-zephyr\dts\bindings\gpio\nordic,nrf-gpio.yaml
```

as one of the entries:

```
gpio-cells:
  - pin
  - flags
```

1.2. Blinky - how to use the LED

To use the hardware device in the program first there is a need to check if this device is defined in the device tree, and if so, set its main parameters. At the beginning of the blinky example, there is the fragment that gets the identifier of the led0 device and checks if the board used is capable of controlling the LED. If so takes the parameters of the LED from the device tree.

```
/* The devicetree node identifier for the "led0" alias. */
#define LED0_NODE DT_ALIAS(led0)

#if DT_NODE_HAS_STATUS(LED0_NODE, okay)
#define LED0    DT_GPIO_LABEL(LED0_NODE, gpios)
#define PIN DT_GPIO_PIN(LED0_NODE, gpios)
#define FLAGS   DT_GPIO_FLAGS(LED0_NODE, gpios)
#else
/* A build error here means your board isn't set up to blink an LED. */
#error "Unsupported board: led0 devicetree alias is not defined"
#define LED0    ""
#define PIN 0
#define FLAGS   0
#endif
```

- **DT_ALIAS** macro takes the identifier and converts it to c compatible string (special characters are replaced with underscore symbols and all characters are converted to lowercase).
- **DT_NODE_HAS_STATUS** checks if the status of the node is OK.

If the status is OK the label, the number of the pin, and flags are set as defined in the device tree. If not, all these elements are cleared. This may look like the overhead, but it is useful while we work with different hardware boards to prevent the compilation of the software for incompatible boards. The initial part of the code creates the structure which holds the parameters of the device and binds it with the LED0 identifier. At the end, the device is configured according to parameters taken from the device tree. After the configuration, the program jumps into the infinite while loop to blink the LED every 1000 ms.

```
/* ... */

/* 1000 msec = 1 sec */
#define SLEEP_TIME_MS   1000

/* ... */

void main(void)
{
```

```
    const struct device *dev;
    bool led_is_on = true;
    int ret;

    dev = device_get_binding(LED0);
    if (dev == NULL) {
        return;
    }

    ret = gpio_pin_configure(dev, PIN, GPIO_OUTPUT_ACTIVE | FLAGS);
    if (ret < 0) {
        return;
    }

    while (1) {
        gpio_pin_set(dev, PIN, (int)led_is_on);
        led_is_on = !led_is_on;
        k_msleep(SLEEP_TIME_MS);
    }
}
```

1.3. Direct register use

Using a device tree is universal but can be found somehow complex. To obtain the same results, certainly, with losing the portability of the program between different microcontrollers and hardware platforms, there is a possibility to write directly to registers of the microcontroller. We need to know what is the base address of the P0 port and offsets of the registers responsible for pin configuration and state. Also, the pin number where the LED is connected. The values are presented in the table 1.

Tab. 1: “Values needed for the program with direst register use”

base address	direction register	port register	pin number
0x50000000	0x514	0x504	14

The program can look like this.

```
#include <kernel.h>

#define SLEEP_TIME_MS      1000
#define P0_BASE_ADDR       0x50000000
#define GPIO_PORT_REG      0x504 //offset of the register for output state
#define GPIO_DIRECTION_REG 0x514 //offset of the register for defining pin
function
#define PIN_NUMBER         14    //led1 on the nRF52840 DK board

int main(void)
{
    volatile uint32_t *p0_dir_reg = (volatile uint32_t *) (P0_BASE_ADDR +
GPIO_DIRECTION_REG);
    *p0_dir_reg |= (1 << PIN_NUMBER);
```

```
    volatile uint32_t *p0_port_reg = (volatile uint32_t *) (P0_BASE_ADDR +
GPIO_PORT_REG);
    while (1) {
    *p0_port_reg |= (1 << PIN_NUMBER);
    k_msleep(SLEEP_TIME_MS);
    *p0_port_reg &= ~(1 << PIN_NUMBER);
    k_msleep(SLEEP_TIME_MS);
    }
    return 0;
}
```

Some explanation of the “volatile” qualifier present in the example could help to clear some doubts. In general compilation and run of the above example without “volatile” should work properly. The troubles could start if there is some other part of the code that modifies the same registers. Because of optimization compiler could not read the port state but set it only, which could result in unexpected modification of other outputs. In general volatile qualifier should be used whenever the value of the variable could change unexpectedly. In practice, only three types of variables could change:

- Shared global variable modified by an interrupt service routine, used also in the main program
- Shared global variable within a multi-threaded application used by more than one thread
- Memory-mapped peripheral registers

In our example, we have that third situation.

1.4. Button example

A similar example named “button” [5] uses one of the onboard buttons to generate interrupts and, at the same time, to control the onboard LED. The button example is not a part of the distribution of the Zephyr package, but can be downloaded from GitHub: https://github.com/zephyrproject-rtos [3]. Because of this, some paths for including libraries should be adjusted after import (usually it is enough to delete the “zephyr/” from paths marked erroneous). The button is included in the program similar way as LED.

```
#define SW0_NODE DT_ALIAS(sw0)
#if DT_NODE_HAS_STATUS(SW0_NODE, okay)
#define SW0_GPIO_LABEL DT_GPIO_LABEL(SW0_NODE, gpios)
#define SW0_GPIO_PIN DT_GPIO_PIN(SW0_NODE, gpios)
#define SW0_GPIO_FLAGS (GPIO_INPUT | DT_GPIO_FLAGS(SW0_NODE, gpios))
#endif
```

Configuration of the button is done with the gpio_pin_configure() function:

```
gpio_pin_configure(button, SW0_GPIO_PIN, SW0_GPIO_FLAGS);
```

The button is configured to signal the interrupt.

```
gpio_pin_interrupt_configure(button, SW0_GPIO_PIN, GPIO_INT_EDGE_TO_ACTIVE);
```

As the cause of the press or release the callback function is started. The definition of this function can

be like in the following code:

```
static struct gpio_callback button_cb_data;
void button_pressed(const struct device *dev, struct gpio_callback *cb,
uint32_t pins)
{
 printk("Button pressed at %" PRIu32 "\n", k_cycle_get_32());
}
```

The callback function must be initialised and configured in the main:

```
gpio_init_callback(&button_cb_data, button_pressed, BIT(SW0_GPIO_PIN));
gpio_add_callback(button, &button_cb_data);
```

Note that to be able to observe the UART port there is a need to enable the serial port in the file prj.conf:

```
CONFIG_SERIAL=y
```

and set the speed of transmission in platformio.ini:

```
monitor_speed = 115200
```

1.5. Tasks for the laboratory on GPIO

1. Import the Blinky (zephyr_blink) example in the VS Code, as shown in the introduction to PlatformIO document.
2. Find a zephyr-button example. It's not importable as the project, so the simplest solution is to replace files main.c and prj.conf in the already imported project (e.g. Blinky). Files can be found in the folder

   ```
   ..\.platformio\packages\framework-zephyr\samples\basic\button\src
   ```

3. Merge two projects “blink”, and “button” together to obtain the program that blinks LED0 continuously, calls the callback function with the interrupt generated with Button0, and controls the state of LED1 with Button1.

1)
https://github.com/zephyrproject-rtos/zephyr/blob/main/samples/basic/blinky/src/main.c
2)
https://docs.zephyrproject.org/latest/build/dts/index.html
3)
https://github.com/zephyrproject-rtos

From:
https://dokuwiki.zmitac.aei.polsl.pl/ - **RAU6 Dokuwiki**

Permanent link:
https://dokuwiki.zmitac.aei.polsl.pl/en:edu:zmitac:s2:aih:lab:nrf52gpio?rev=1701037474

Last update: **2023/11/26 23:24**