Prof. Dr. Marco Zimmerling Max Granzow Jonas Kubicki
https://nes-lab.org/

Networked and Low-Power Embedded Systems

WS 2025/26

Lab 2: GPIOs and Clocks

Introduction

The programming of embedded systems varies depending on the platform and toolchains used. Popular systems like Arduino offer an entire ecosystem and abstract many hardware details to simplify and speed up application development. However, behind all the available operating systems, frameworks, and libraries, someone needs to know how to deal with the specific feature of the hardware. In this lab, we want you to work close to the hardware, as this will give you a deeper understanding of how things work and interact. At this level, the datasheets are your best friend. The datasheets will give you an overview of the internal structure of the hardware, as well as the peripherals available and how to configure and use them. Datasheets with several hundred pages can seem demotivating at first. However, you will find that after a short familiarization period, often only a few pages are needed to get the job done.

Download the datasheet "`nRF52840_PS_v1.8.pdf`" for our nRF52840 chip from Moodle. For a first overview just have a look at the table of contents. Sections 2.3 and 2.4 in particular will be relevant in the following tasks.

1 Light It On

The first task is to control the LED on the development kit, which is something like the "Hello World!" equivalent in the embedded world. The devkit has a green LED connected to GPIO port 0 pin 13 (i.e. P0.13) of the nRF52840 chip.

1.1 Juggling with Addresses

We start with the most bare metal approach and access the hardware directly via addresses (keyword: memory-mapped device). Follow the steps below and complete the code in the file `task1_1.c` to turn on the LED. Remember the steps from lab 1 to compile and flash your program. Compile the source code with `$ make task1_1` and flash it with `$ make flash1_1`.

1. Consult the datasheet and determine the base address of GPIO port 0 (P0) and the address offset that points to the configuration register to which the LED is connected.
2. How do we have to configure the GPIO pin in order to control the LED? In this case, the reset value is almost correct except for one field.
3. Look at the registers in section 6.9.2 of the datasheet. Which register do we have to use to turn on the LED? The LED on the devkit is "active low", that is, if the corresponding register value is set to `0`, the LED lights up. In general, depending on the circuit, a LED can also be "active high" and light up when the register value is `1`.

> In the embedded world, the term "set" often refers to assigning the value `1` while "clear" assigns the value `0`.

1.2 Common Microcontroller Software Interface Standard (CMSIS)

The use of addresses is a cumbersome and error-prone approach. More convenient access to the hardware functionality is provided by libraries, hardware abstraction layers (HAL), and other interfaces. For Cortex-M processors, CMSIS is a low-level interface that we will use in the lab. Take a look at `Include/nrf52840.h` (contains structures and addresses for processor and peripherals) and `Include/nrf52840_bitfields.h` (contains definitions for easier register access). Follow the steps below and complete the code in the file `task1_2.c`. Implement the same functionality as in task 1.1, but this time use the CMSIS definitions. Compile the source code with `$ make task1_2` and flash it with `$ make flash1_2`.

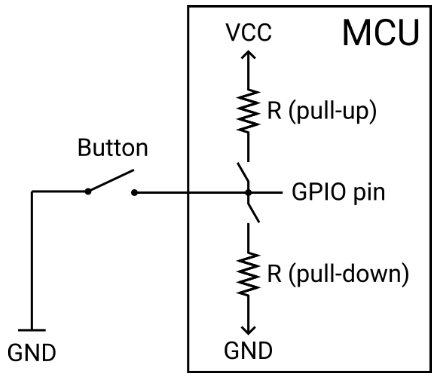
1. Access the GPIO peripheral via the GPIO port `0` structure defined in `nrf52840.h`. The required header files are included by the `nrf.h` file.
2. Use the definitions from `nrf52840_bitfields.h` to set the register values. You will see that the definitions match the register, field, and value names in the datasheet (i.e. <peripheral>_<register>_<field>_<value>). Please note that the register values have to be shifted to the right position in the register (see `xxx_Pos` definitions).

2 Push the Button

In our next task, we want to control the behavior of the LED (P0.13, see back of devkit) by pressing a button. The devkit has a push button connected to GPIO port `0` pin 11 (i.e. P0.11, see back of devkit). If you want to inspect the signal, you have to connect the LA to pin P0.11 of the expansion header. Pay attention to the following steps and finish the implementation in `task2.c` so that the LED lights up when the button is pressed and is off otherwise. Compile the source code with `$ make task2`.

1. How do we have to configure the GPIO pin of the push button? To determine the correct pull configuration, have a look at the following schematic.

   In order to avoid a so called "floating" pin, pull-up or pull-down resistors are used. A pin is called floating when there is no fixed voltage level connected to the pin, which can lead to unexpected behaviour when the pin is read and increasing energy consumption. A pull-up resistor ensures that the pin is at a high voltage level when the button is not pressed (open circuit), and at a low level when the button is pressed (short circuit). The pull-down resistor works in reverse.



2. Which register do we have to read to determine the current state of the button? To check for the correct bit in the register, you can make use of the corresponding `xxx_Msk` definitions in `nrf52840_bitfields.h`, for example, `(reg & xxx_Msk) == 1`.

3 LED Control via Timer Polling

We will now use the timer peripheral to control the LED. Timers are an important peripheral in embedded systems because they give you access to a clock and allow you to time your actions, for example, to periodically read a sensor. Your task is to toggle the LED approximately every second so that it is on for 1 second, then off for 1 second and so on. Note the following steps and finish the implementation in `task3.c` so that the LED behaves as described. Compile the source code with `$ make task3`.

1. Get an overview of the timer peripheral in the datasheet. The nRF chip has five different timers. We will use `TIMER0` in this lab. Determine a suitable configuration of the `MODE`, `BITMODE`, and `PRESCALER` registers.

> The timer counts with frequency f_{TIMER}. The `PRESCALER` essentially controls how fast the timer counts, while `BITMODE` determines how high we can count before the counter overflows (i.e. starts at 0 again). Choose suitable values so that at least one second can be counted without the timer counter overflowing.

2. Start the timer and continuously check its count (i.e. poll the timer) to implement the correct timing behavior of the LED. However, the nRF chip is special in that you cannot inspect the timer counter directly. Read about the “capture” functionality to obtain the current count.

> If you want to analyze/debug your code with the LA, you can simply connect to the LED pin. Alternatively, you could mirror the behavior of the LED pin with another GPIO pin and connect the LA to it instead, for example Port 1 Pin 8.

3. Use the bitwise XOR operator (^) to toggle the LED. Toggling means switching it on when it is off and off when it is on. To toggle the correct bit in the register, you can again use the corresponding `xxx_Msk` definitions.

4 LED Control via Timer Interrupts

In the previous task, we continuously polled the timer to control the behavior of the LED. However, this keeps the CPU busy all the time and wastes valuable energy. We now take a look at interrupts and how they help us to avoid wasting energy. Toggle the LED approximately every second, but this time the CPU should sleep in between. Follow the steps below and finish the implementation in `task4.c`. Compile the source code with `$ make task4`.

1. Setup the GPIO pin of the LED and the timer peripheral as in task 3. Find out about the `compare` functionality of the timer peripheral to generate events. To be able to react to these events, we need to enable interrupts. There are two levels of interrupt control. We need to enable interrupts in the peripheral, for example, to trigger interrupts for specific pins of a GPIO port, and we have to enable interrupts for the peripheral in the interrupt controller (`NVIC`).

> The timer peripheral features five different capture/compare (`CC`) registers that can be used for different timings in parallel. Interrupts can be enabled separately for each `CC` register.

2. Reset the timer counter, set the time in the corresponding register at which the interrupt should trigger, and start the timer.
3. If an interrupt occurs, the current program execution state is saved (or the CPU wakes up) and the program jumps to the corresponding interrupt service routine. Implement the interrupt service routine for the timer (`TIMER0_IRQHandler`). Toggle the LED, update the corresponding register to trigger the ISR again after one second, and don't forget to clear the event that caused the interrupt.