Bern University
of Applied Sciences

Exercise - 04

CMSIS Driver Introduction

Andreas Habegger | Adrian Steiner
BTS3230 | Version 1.0.0 of 17.03.2025

Please be aware that the content is subject to change at any time. For the latest version, please check the website.

In this exercise you will redo the Edge Detection Exercise by using the CMSIS driver. First read the theory concerning the CMSIS driver, then use it to reimplement the Ex-03.

Objectives

- CMSIS bit manipulation
- CMSIS register access

Outcomes

- Implement software using the CMSIS driver
- Understand the structure of the CMSIS driver

CMSIS Driver

CMSIS description

Writing directly custom hardware addresses, as in the previous exercises, is error-prone, unreadable and impractical for reusing code. The MCU manufacturer ARM solve these problems with a driver library. The **C**ommon **M**icrocontroller **S**oftware **I**nterface **S**tandard (CMSIS) enables device support and simple software interface to the processor and its peripherals. To use this driver with our MCU, two header files must be included:

```
#include "stm32f4xx.h"
#include "stm32f446xx.h"
```

The first include is a general include for stm32f4 controllers and the second is the specific include for stm32f446 controllers including all the addresses and register types.

CMSIS macros

The structure of these macros are always the same.

Template

```
<PERIPHERAL>_<REGISTER>_<BITFIELD>_<ADDITIONALS>
```

Example

```
GPIO_IDR_ID1
```

The description of the structure:

- **PERIPHERAL**

  The peripherals are the functional parts of the MCU such as GPIO, TIM, ADC, etc.

- **REGISTER**

  The name of the register in the peripheral device according to the reference manual RM0390 reference manual.

- **BITFIELD**

  The bit or bits to be changed. For example, the input register of a GPIO uses one bit for one input. The GPIO port mode register uses two bits for one pin.

- **ADDITIONAL**

  There are four possibilities:

  1. Nothing: Use directly the masked bitfield.
  2. Bit number: The specific bit in the bitfield. For example, in the GPIO port mode register, bit 0 and 1 are included in the bitfield of a pin.
  3. Pos: The bitfield's position in the register.

4. Msk: The masked bitfield as you have used in previous exercises.

CMSIS TypeDefs

In addition to the macros, CMSIS provides a struct of all the registers for each peripheral in the MCU. The struct is exactly the size of all the registers in a peripheral and the members are on the physical registers. This has the advantage that a pointer can be used to refer to the appropriate register with a readable name. CMSIS defines peripheral type macros at the correct address.

Register Access with CMSIS

Here is an example of how to access the clock control register of the reset and clock control peripherals:

```
RCC->CR
```

CMSIS bit manipulation

In previous exercises you learned how to manipulate the registers of a peripheral with bit-wise operators. For a real-world implementation, we will probably use a set of macros to do this, rather than writing the plain bit operations. The code snippet below shows the most commonly used bit manipulation macros defined within the **“stm32f4xx.h”** header file.

Extract of most important bit manipulations

```
#define SET_BIT(REG, BIT)     ((REG) |= (BIT))
#define CLEAR_BIT(REG, BIT)   ((REG) &= ~(BIT))
#define READ_BIT(REG, BIT)    ((REG) & (BIT))
#define CLEAR_REG(REG)        ((REG) = (0x0))
#define WRITE_REG(REG, VAL)   ((REG) = (VAL))
#define READ_REG(REG)         ((REG))
#define MODIFY_REG(REG, CLEARMASK, SETMASK)  WRITE_REG((REG), (((READ_REG(REG)) &
(~(CLEARMASK))) | (SETMASK)))
```

Hint

As a little help, here are some examples of how to use the macros with the help of CMSIS:

Bit Manipulations

SET_BIT()

```
SET_BIT(RCC->AHB1ENR, RCC_AHB1ENR_GPIODEN);
```

Set the desired bit in the register. In this case, the GPIOD peripheral clock is enabled

CLEAR_BIT()

```
CLEAR_BIT(GPIOB->ODR, GPIO_ODR_OD4);
```

Clears the desired bit in the register. In this example the output PB4 is turned off.

READ_BIT()

```
READ_BIT(GPIOC->IDR, GPIO_IDR_ID12);
```

Read only the state of the desired bit in the register. The input PC12 is read in this example.

Register Manipulations

CLEAR_REG()

```
CLEAR_REG(GPIOC->PUPDR);
```

Sets the hole register to zero. All pull-up and pull-down resistors of peripheral GPIOC are turned off in this case.

WRITE_REG()

```
WRITE_REG(SDIO->ARG, 0x2);
```

Writes the value into the register. In this case the 0x2 hex-value is written into the SDIO argument register.

READ_REG()

```
READ_REG(SPI1->DR);
```

Read the hole register. Here the SPI data register is read.

MODIFY_REG()

```
MODIFY_REG(GPIOB->MODER, GPIO_MODER_MODE4_Msk, 0b01 << GPIO_MODER_MODE4_Pos);
```

With modify register you can set to a bitfield a desired set of bit. In this example the GPIOB port mode register for I/O pin 4 is set as an output.

> **Note**
>
> Have a look to set the bitfield on the correct position to not modify other bits.

Tasks

The aim is to reimplement the edge detection from ex-03, therefore refer to the task section in ex-03.

> **Tip**
>
> You don't need to reimplement the complete code. Only change the implementation where register addresses are used and bit manipulations take place.