**EL-GY 6483**
**Homework 3**

---

---

In class, we talked about memory mapped I/O. In memory mapped I/O, peripherals are accessed by writing to certain addresses in memory, using the same instructions as for other memory access.

As embedded systems engineers, you may not always have the luxury of accessing GPIO peripherals using well-documented, mature libraries for popular microcontrollers. You may be asked to design a new embedded system, to write firmware libraries for accessing GPIO on a new embedded system, or to use poorly documented in-house libraries for accessing GPIO. The aim of this homework, therefore, is to help you gain a deeper understanding of how memory mapped I/O works so that you will be capable not only of using GPIO libraries, but also writing and documenting them.

1. Please refer to the Cortex M-4 Technical Reference Manual and the STM32F4 Discovery Reference Manual (attached to this HW) to answer this question.

   (a) Section 3.4 in the Cortex M-4 Technical Reference Manual defines the memory map for this processor. What range of memory addresses is reserved for use by peripherals?

   0x40000000 — 0x60000000

   (b) Section 2.3 of the STM32F4 Discovery Reference Manual further defines the memory used by each individual peripheral. The GPIO peripherals are listed by port (GPIOA, GPIOB, GPIOC, GPIOD, etc.). What is the address range used by the GPIOD port?

   0X40020C00 — 0X40020FF

   (c) On the STM32F4 Discovery board, each GPIO pin gets some dedicated configuration and data registers. These are described in Section 8, and especially Section 8.4, of the STM32F4 Discovery Technical Reference Manual. Please answer these questions for the following registers: `MODER`, `OTYPER`, `OSPEEDR`, `PUPDR`, `IDR`, `ODR`, and `BSRR`. For the `BSRR` register, you should further subdivide into `BSRRL` (BSRR low, used for bit set) and `BSRRH` (BSRR high, used for bit reset).

   - The functionality of the register; what is it used for (in one sentence)?
   - How much memory is used by this register for the entire port (typically 32 bits - 4 bytes - or 16 bits - 2 bytes)? How much memory is used for each individual pin (typically 2 bits or 1 bit)?
   - What is the address offset of this register? The address offset, together with the port address, tells us exactly where in memory the register is located. For example, if the address offset of a register is `0x02` and the port uses the memory range `0x40022800 - 0x40022BFF`, then the register is at address `0x40022800 + 0x02 = 0x40022802`.

**Solution:**

MODER

- Use: Configure the I/O direction mode
- Use 32 bits total, 2 bits for each pin
- Address offset: 0x00

OTYPER

- Use: configure the output type of the I/O port
- Use 32 bits total. However, the top 16 bits are reserved. 1 bit use for pin
- Address offset: 0x04

OSPEEDR

- Use: configure the configure the I/O output speed
- Use 32 bits total. 2 bits are used for each pins.
- Address offset: 0x08

PUPDR

- configure the I/O to use a pull-up or pull-down resistor
- Use 32 bit total. ~~However, the top 16 bits are reserved~~
- ~~0x10~~ 0x0C 2 bits are used for each pin. ~~1 bit is used for pin.~~

IDR

- read ~~port~~ input data
- Use 32 bits total. Top 16 bits are reserved. 1 bit is use for each pin
- ~~0x1~~ 0x10

ODR

- write port output data
- Use 32 bit total. Top 16 bits are reserved. 1 bit is used for each pin
- 0x14

BSRRL

- set bit with atomic operation
- Use 16 total, 1 bit for each pin
- 0x18

BSRRH

- reset bit with atonic opereration
- Use 16 bits total, 1 bit is used for each pin.
- 0x1a

2. This is a continuation of the previous question. Now that we know exactly where in memory each register is located, we can write C code to access these memory locations.

   When you define a structure in C, the memory for elements of the structure will be laid out in the same order in which you defined them. Given that the registers are located sequentially in memory, we can use a structure in C to map registers to human readable names.

   (a) We will use a `GPIO_TypeDef` struct to map memory for GPIO registers to human readable names. Here's a template, showing the struct definition and also how we define `GPIOD` using the struct:

   ```
   typedef struct
   {
    ...
   } GPIO_TypeDef;

   #define GPIOD_BASE     (0xAAAAAAAA)  // insert correct memory address here
   #define GPIOD          ((GPIO_TypeDef *) GPIOD_BASE)
   ```

   Inside the typedef, we will add a line for each register that defines its total size for the entire port (32 bits - `uint32_t` - or 16 bits - `uint16_t`), its name, and a comment giving its functionality. In order to make sure the struct element is located at the correct memory offset, we need to make sure to define them *in order*. We will also define each element as `volatile`, to signal to the compiler that they may be modified outside of our code (e.g., by peripherals) and shouldn't be optimized away.

   For example, the `MODER` register occupies 32 bits, and is located at an address offset of `0x00` (see 8.4.1 of the Technical Reference Manual). So we have to define it as the first element of the structure, and as a `uint32_t`. After adding `MODER`, our typedef would look like this:

   ```
   typedef struct
   {
     volatile uint32_t MODER;     // set port mode, e.g. input, output
     ...
   } GPIO_TypeDef;

   #define GPIOD_BASE     (0xAAAAAAAA)  // insert correct memory address here
   #define GPIOD          ((GPIO_TypeDef *) GPIOD_BASE)
   ```

   and we would be able to access the `GPIOD MODER` register as `GPIOD->MODER`.

   (b) Please fill out the following C template. Add an element to the structure for each of: `MODER`, `OTYPER`, `OSPEEDR`, `PUPDR`, `IDR`, `ODR`, `BSRRL`, and `BSRRH`. Also, set the correct value for the `GPIOD_BASE` address based on your answer to question 1(b).

**Solution:**

```
typedef struct
{ volatile uint32_t MODER; //0x00
  volatile uint32_t OTYPER; //0x04
     .        .     OSPEEDR; //0x08
     .        |     PUPDR ; //0x0C
     (        |     IDR; // 0x10    volatile uint32_t BSRRL ;//0x18
     |        |     ODR; //0x14
} GPIO_TypeDef;                                       BSRRH //0x1A

#define GPIOD_BASE     (0x40020C00)
#define GPIOD          ((GPIO_TypeDef *) GPIOD_BASE)
```

Note that the elements *must* be defined in the right order and be the right sizes so that they will be located at the right memory location (based on the offset defined in the datasheet). For example, given the definition above:

- The struct begins at the memory location defined with `GPIOD_BASE`, `0x40020C00`.
- The first element in the struct is at that memory location + `0x00` offset. So when we write to `GPIOD->MODER`, we are writing to the memory location `0x40020C00`.
- The second element in the struct (`OTYPER`) is at `0x40020C00` + the additional bytes used by the `MODER` register. Since `MODER` is defined to use 32 bits (4 bytes), `OTYPER` will be at `0x40020C00` + `0x04` offset = `0x40020C04`
- The third element in the struct (`PUPDR`) is at `0x40020C00` + the additional bytes used by the previous elements, the `MODER` and `OTYPER` registers. Since `MODER` and `OTYPER` are both defined to use 32 bits (4 bytes), 8 bytes total, `PUPDR` will be at `0x40020C00` + `0x08` offset = `0x40020C08`
- Following a similar procedure, you can see that all of the struct elements will be located at the correct memory addresses. (Note that some elements are defined as 16 bits).

If you look in `stm32f4xx.h` you'll see a similar struct definition. So now we've learned how to write a basic GPIO peripheral driver, using just the datasheet for reference.

(c) Which element(s) of the struct do you think you might declare as a `volatile const` instead of just a `volatile`? Why?

We might declare ~~IR~~ IDR as volatile const because it should be read-only to us

(d) Do we need to use `malloc()` in C to allocate memory for GPIOD? Why or why not?

No. We don't need to use malloc() to allocate memory for GPIOD malloc() is used to allocate memory dynamically on the heap We're using memory in the peripherals address range. We don't want memory on heap. We define our pointers to use the correct memory address, in the peripherals range.

(e) Please answer these questions for the following registers: `MODER`, `OTYPER`, `OSPEEDR`, `PUPDR`, `IDR`, `ODR`, `BSRRL`, and `BSRRH`:

- For all registers except `IDR`, write a line of C code that assigns the value '1' to this register for pin 3 of the GPIOD port, without affecting the values set for any other pins. For examples, see pages 12 and 14 of the lecture slides on peripherals.
- For all registers except `IDR`, write a line of C code that assigns the value '1' to this register for *all* pins on the GPIOD port.
- What does setting a value of '1' do on this register?

**Solution:** To set values only for pin 3:

```
GPIOD->MODER &= ~(3<<2*3);
GPIOD->MODER |= 1<<2*3 ;

GPIOD->OTYPER |= 1<<3;

GPIOD->OSPEEDR &= ~(3<<2*3);
GPIOD->OSPEEDR |=1<< 2*3 ;

GPIOD->PUPDR &= ~(3<<2* 3);
GPIOD->PUPDR |= 1<<2*3;

GPIOD->ODR |= 1<<3;
GPIOD->BSRRL |= 1<<3;
GPIOD->BSRRH |= 1<<3;
```

To set values for all pins on the port:

```
GPIOD->MODER = 0x55555555 ;
GPIOD->OTYPER |= 0xffff ;
GPIOD->PUPDR = 0x55555555 ;

GPIOD->ODR |= 0xffff ;

GPIOD->BSRRL =0xffff ;

GPIOD->BSRRH = 0xffff ;
```

MODER: Set pin to general purpose output mode.
OTYPER: Set output pin to open drain type.
OSPEEDR: Set speed to medium speed
PUPDR: Set a pull-up resistor on pins
ODR: Write a high value to output pins
BSRRL: Set a value of 1 on output pins
BSRRH: Reset on output pins.