4. The use of ports

The microcontroller has five sets of I/O pins (16 pins per set) called ports to be used as inputs or outputs for electrical signals. Each I/O pin can have different properties to accommodate the desired type of a signal. In addition, each I/O pin can have one from a set of 16 predefined alternative functions. Both properties and functions can be defined in user software by writing the appropriate values into registers inside of the microcontroller. We will explore the properties and setting of alternative functions for a single I/O pin in this chapter.

An I/O pin can be used to 00) pass the external digital signal to the microcontroller or to 01) pass the internally software-generated digital signal to the outer world. Additionally, the hardware blocks inside can use or generate a digital signal to be obtained from or passed to outer world through a pin; this is called 10) alternate function. The microcontroller also houses analog blocks, and digital signals can harm analog ones, so it is best to disable all digital units connected to the pin when it is used for 11) passing analog values. There are four possibilities for each pin; two bits inside a dedicated register (**MODER**) define the direction and nature of the signal at a pin. There are 16 pins per port, so the register MODER must have 32 bits in width. There will be a register MODER for each port individually, lower two bits define LSB pin of the port… Upon RESET these registers default to option 00).

An I/O pin can have different properties when used as input for external signals. Variants can be best described by commenting the block diagram in Fig. 1 (a copy from the RM0090, chapter 6.3, Fig.

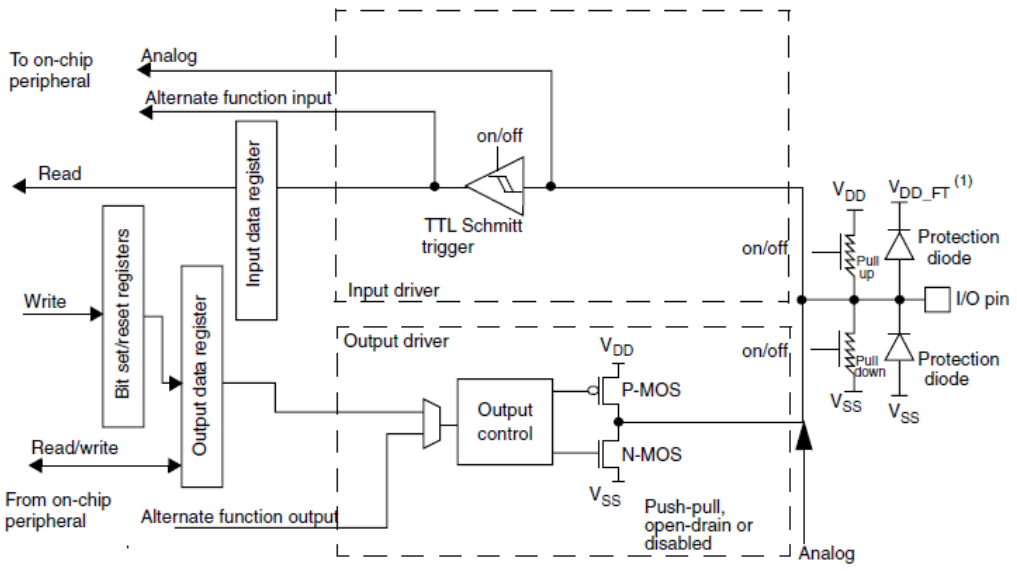

Figure 1: The configuration of hardware for a single pin of a port

13). In this configuration the Output driver circuit (lower dashed rectangle) is disabled and has no influence on the value at the I/O pin. The input signal is connected at I/O pin, and either of pull-up / pull-down resistors may be active here. There are three possibilities:

00) both pull-up and pull-down resistors are disabled, the input is called “floating” since the value on the pin depends solely on the value of input signal,
01) the pull-up resistor is enabled and the pull-down resistor is disabled, the input is called “pull-up” since the value on the pin is equal to the value of input signal when present, and is high when input signal is not defined, and
10) the pull-up resistor is disabled and the pull-down resistor is enabled, the input is called “pull-down” since the value on the pin is equal to the value of input signal when present, and is low when input signal is not defined.

There is a register (**PUPDR**) inside the microcontroller that holds two bits to define the properties for an individual pin. Since 16 pins form a port, and two bits are required for each pin, a register PUPDR is 32 bits in length. Lower two bits of register PUPDR define LSB pin of the port … There is a register PUPDR for each port individually, and these registers are reset to option 00).

The digital signal is fed to the TTL Schmidt trigger and enters the input data register (**IDR**), from where the software can read its value. The register IDR is 16 bits in length. Least significant bit of register IDR stands for the LSB pin of the port … There is a register IDR for each port individually.

An I/O pin can have different properties when used as output for signals from microcontroller, when both input and output drivers are enabled. The value at the input pin can still be read from the register IDR. The output pin can:

0) force the signal on the pin to go low or high (push-pull mode) or
1) force the signal on the pin to go low only (open drain mode).

There is a register (**OTYPER**) inside the microcontroller that holds a bit to define the capabilities for driving the pin for each pin individually; the register OTYPER is 16 bits wide. Least significant bit of this register defined LSB pin … There is a register OTYPER for each port individually, and these registers are reset to option 0).

The hardware can be very fast but this requires power. The speed of the hardware can be reduced in order to save power for each pin individually. Possible options are:

00) low speed, 2 MHz
01) medium speed, 25 MHz
10) fast speed, 50 MHz
11) high speed, 100 MHz

There is a register (**OSPEEDR**) inside the microcontroller that holds two bits to define the speed of each pin of the port individually. The register OSPEED is 32 bits wide, lower two bits define the LSB pin of the port … There is a register OSPEED for each port individually, and these registers are reset to option 00).

As seen in the Fig. 1 the value of the signal at pin comes from a register Output Data Register (**ODR**). The register is 16 bits wide, the least significant bit of the register defines the value at LSB pin of the port … There is a register ODR for each port individually, and these registers are reset to value 0. The content of this register is written by the user software. There is an alternative way for

changing the content of register ODR besides writing the new value into it directly. A register (**BSSR**) has 32 bits, and writing a high in a bit 0 (LSB) causes bit 0 of the register ODR to become high regardless of its previous value. Alternatively, writing a high in a bit 16 (LSB in the upper half of the register) causes bit 0 of the register ODR to become low. This can be done for more bits simultaneously and provides a nice way to change values at output of a port without doing any logical functions inside the microcontroller.

When alternate functions are selected in register MODE, the output driver is connected to the output of a multiplexer with 16 inputs. The multiplexer selects one of 16 possible alternative signal sources within the hardware of the microcontroller; 4 bits are needed to define the multiplexer's selection. Altogether 4 bits x 16 pins = 64 bits are needed to define alternative functions for one port, and these bits are stored in two 32 bit registers (**AFRL**, **AFRH**). Each port requires it own register AFRL and AFRH. The details on the possibilities will be presented in relevant chapters to follow. For now it is only important that these registers default to all zeros which enable standard I/O function under the control of software. In the meantime the interested reader might want to read the complete Chapter 6, RM0090.

In order to demonstrate the use of ports a simple program will be presented. The program reads the value of the pushbutton S370 (connected to port E, LSB), and turns on the LED D390 (connected to port A, bit 6) when the pushbutton is pressed. The program initially enables the clock for ports A and E, then defines the use of port pins and finally enters a loop to read the pushbutton and write the LED. The listing is presented in Fig. 2.

```
#include "stm32f4xx.h"                  // the required include

void main (void)  {                     // program start
char In;                                // define a variable to hold input value

  RCC->AHB1ENR |= 0x10 + 0x01;          // Enable clock for ports A &  E

  GPIOA->MODER |= 0x00001000;           // MODE == OUT: LED at PA06

  while (1) {                           // infinite loop
    In = GPIOE->IDR & 0x01;             // read bit 0, port E into variable In
    If (In & 0x01)                      // if In == 1 then
       GPIOA->ODR |= 0x0040;            // turn on LED
    else                                // else
       GPIOA->ODR &= ~0x0040;           // turn off LED
  };                                    // end of infinite loop
}                                       // end of program
```

Figure 2: The listing of a simple program

The naming of register is follows the definitions in the included files. It consists of a port name (GPIOA for instance), followed by an arrow (indicating a structure in "c") and a register name (MODER for instance). The naming of bits does not follow the definitions given in the included files. The reason is simple: the author sees no benefit in learning names for bits in each individual register by hart. Instead, he prefers to check the function of bits in each individual register in RM0090 and uses hex notation to set/reset the desired bits. However, feel free to use other notation if desired.

The program starts with a write to a not yet mentioned register **AHB1ENR** (AHB stands for AH bus inside the microprocessor, 1 is the index of the register, and then comes ENable Register). There is a complex clock circuitry built into the microcontroller, and this register defines the distribution of clock signals throughout the microcontroller. There actually several registers involved in enabling and defining the clocks, and all are located in a so called Reset & Clock Control (hence RCC) block. The

details can be seen in Chapter 5, RM0090, and will be disclosed in due time also in these examples. For now it is enough to say that no peripheral can change its state if a clock for that peripheral is not enabled. Register AHB1ENR lower 9 bits are responsible for enabling of the clock to nine ports available in the largest version of STM32F4xx microcontroller. The LBS is responsible for port A …

There were more register associated with ports at the beginning of this chapter. However, their default value is good to be used for this simple example and is therefore not touched here.

Please note that the content of register is modified not by direct writing but rather by setting the appropriate bits using function OR (for instance: RCC->AHB1ENR **|=** 0x10 + 0x01;) . This technique keeps the other, for now not important bits, in their previous states. The resetting of bits should also be performed with caution using function AND and the inverted version of the argument, again to avoid tempering with bits that are not to be affected. There are also shortcuts defined in include files to ease avoiding of such pitfalls, but the author prefers not to memorize additional shortcut names and prefers to have the content of registers under complete control.

Figure 3 gives another simple example to blink a LED connected to port A, bit 7.

```
#include "stm32f4xx.h"

void main (void)  {

  RCC->AHB1ENR |= 0x08;                        // Enable clock for GPIOD

  GPIOD->MODER |= 0x00100000;                  // MODE Register: bit 10 == out

  while (1) {
    for (int i=0; i<0x1000000; i++) {};        // waste some time
    GPIOD->ODR ^= 0x0400;                      // Toggle PD10
  };
}
```

Figure 3: A program to blink the LED connected to port D, bit 10

More than one bit can be changed at a time. Program in Fig. 4 sends the content of an 8-bit counter to port D.

```
#include "stm32f4xx.h"

void main (void)  {
unsigned int i=0;                       // define variable to count

  RCC->AHB1ENR |= 0x08;                 // Enable clock for GPIOD

  GPIOD->MODER   |= 0x00005555;         // MODE: lower 8 bits are outputs

  while (1) {
    GPIOD->ODR = i & 0x00ff;            // send lower 8 bits to port
    i++;                                // increment counter
  };
}
```

Figure 4: The content of a software counter is sent to port D, lower eight bits