

## 4. The use of ports

The configuration and the use of microcontroller ports is given.

### 4.1. Hardware – a port

The microcontroller STM32F407VG has five sets of pins (16 pins per set), each set is called a port. Each pin of every port can be individually configured as input or output for a digital signal, and for each of these options additional settings to accommodate the desired type of a signal are available. Alternatively, each pin can handle one from a set of 16 predefined alternate functions. Both settings and functions can be configured from within the user program; writing the appropriate values into registers inside of microcontroller defines the properties of pins and ports.

A basic block diagram of a pin and associated circuit is given in Fig. 4.1. The pin itself is shown on the right side of the figure (I/O pin). Two diodes are connected to ground and power supply respectively to protect (to some degree) the microcontroller from overvoltage connected by the unwary user. The microcontroller can write to a pin by writing the data into the 'Output data register' shown in the left part of the figure. This register is 16 bits wide, and each bit is connected to one pin of the port. The microcontroller can read the pin by reading the content of the Input data register, also shown in the left part of the figure. The register is again 16 bits wide, and each bit is linked to one of the pins of the port. The intermediate hardware between pin and data registers is shown in two dashed rectangles named 'Input driver' and 'Output driver'. The properties of these drivers are programmable.

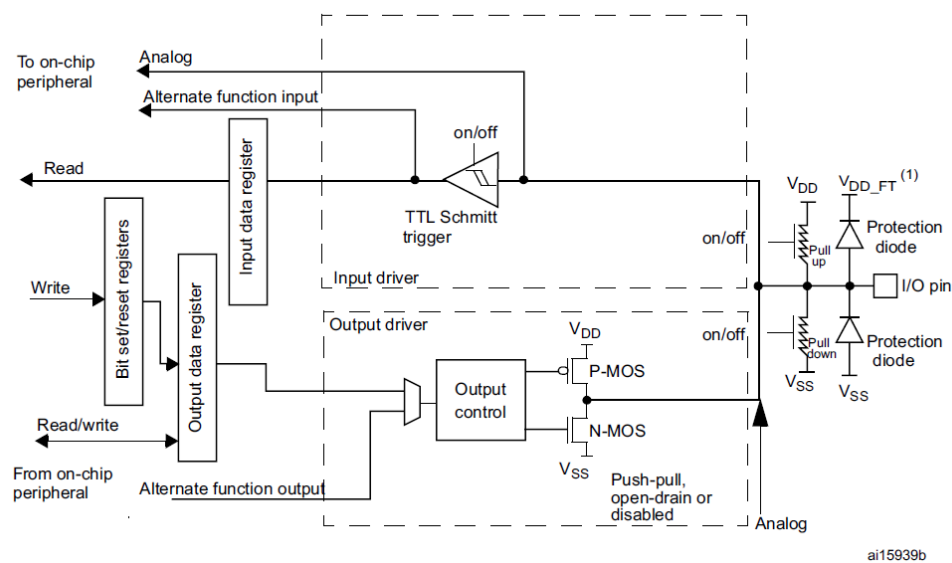


Figure 4.1: A basic block diagram of the circuit associated with a pin, RM0090, pg. 266

A pin can be used as an output; for instance when the microcontroller sends signal through the pin to a LED. In this case both or at least one of the two MOS transistor constituting 'Output driver' must

be active, and this mode of operation must be known in advance. Alternatively, a pin can be used as an input; for instance when accepting the value from a switch. In this case both MOS transistors must be disabled in advance not to interfere with the signal from the switch. The direction of the signal flow is configured during the programming.

When a pin is configured as an input, the default level at the input is not defined, i.e. the pin is floating. This might not be convenient. Consider the situation where a pushbutton is connected between a pin and a power supply. The signal at the pin is defined when the pushbutton is pressed, but left floating otherwise. This can be fixed by connecting a resistor from the pin to ground; this defines the signal when the pushbutton is not pressed. Two resistors to terminate the pin to either ground or power supply are available for every pin, and can be activated by configuration.

The output driver can work at different speeds. The speed is related with the current consumption: faster action requires more current. The speed can also be configured.

In addition to simple passing of signals to and from the microcontroller pins can be used for so-called alternate functions. This includes the passing of analog signals to ADCs and DACs, and passing of digital signals to and from timers for instance. In all alternate functions cases input and output drivers must be adjusted to the task. Again, this ought to be configured prior to the use.

The reference manual RM0090 gives a detailed description of ports and settings available in chapter 8. There is a table with all registers associated with the properties of ports, and meaning of all bits within these registers. The properties of a port can be configured by writing in these registers. However, a detailed knowledge on individual bits of within the registers is mandatory, and this can be an overwhelming task. The CMSIS standard provides a convenient set of functions and keywords to ease the setting-up of the ports.

The operation of any peripheral or the CPU is timed by a designated clock signal. In order to conserve power most of the clock signals for peripherals are disabled on reset, and must be enabled before the peripheral can be used.

## 4.2. Software – the CMSIS header and include files

The CMSIS functions to define the operation of the port are given in “stm32f4xx\_gpio.c” (GPIO == ‘General Purpose Input Output’), while the CMSIS functions to enable the clock signal for individual peripherals are available in “stm32f4xx\_rcc.c” (RCC == ‘Reset and Clock Control’). Note the name of these files, especially the part after the ‘\_’; it reflects the peripheral referred to by the file. Both files must be included from the user program to use functions from them. The keywords and definitions used by these functions are stored in header files “stm32f4xx\_gpio.h” and “stm32f4xx\_rcc.h” respectively. These two files are automatically included in the compilation process from the header file “stm32f4xx.h”, which must be included from within the user program.

A user program to manipulate any of the ports using the CMSIS functions must therefore start with three “#include” commands:

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
```

The first of the files to include is modified from the original in the CMSIS library, and is stored locally in the project directory. The other two files are originals from the CMSIS library, and pointers are set to them in “Options...” before the compilation, see chapter 3 for details.

A brief instruction on the use of CMSIS functions for manipulating a port is given at the beginning of the file “stm32f4xx\_gpio.c” in section “How to use this driver”. The recommended procedure is listed in steps marked with ‘(#)’:

- enable the clock for the port using function “RCC\_AHB1PeriphClockCmd”,
- configure the port (possible configurations are listed) using the function “GPIO\_Init”,
- configure alternate functions if applicable, and
- use the port.

Similar sections with brief instructions can be found in all “stm32f4xx\_\$.c” files containing functions for other peripherals.

A simple example follows. The microcontroller is used as a counter; it increments a variable, and the value of this variable is sent to port E. The STM32F4-Discovery board has most of microprocessor pins accessible, but many of them are best used for special purposes, like ADCs, DAC, counters, communication, etc. The BaseBoard exposes port E, pins 8 to 16, at connector K440 as general purpose IO pins, and these will be used to pass the content of the counter. Due to the selection of pins used it is best to increment the counter by LSB of the exposed bits, therefore by 0x100. The listing of the core of the program is given below:

```
while (1) {                                // about 50ns per iteration
    number += 0x100;                         // increment at BIT_8
    GPIOE->ODR = number;                     // send to PortE
};
```

An infinite ‘while’ loop is used, and within the loop the content of the variable ‘number’ is first incremented by 0x100 (100<sub>HEX</sub> == 256<sub>10</sub>, increment by one at bit 8), and then sent to the port E. The name ‘GPIOE->ODR’ points to port E, output data register. Alternatively one could use here the CMSIS function “GPIO\_Write” to do the same (CMSIS function performs syntax checking), but the direct writing into the output data register used here is faster.

As stated in brief instructions above the user must enable the clock for the port E and configure the port before the use. Both actions are best enclosed into a function named “GPIOEinit” that is called prior to the use of the port:

```
void GPIOEinit (void) {
    GPIO_InitTypeDef  GPIO_InitStructure;                                // 2

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);           // 4

    GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10 | GPIO_Pin_11; // 6
    GPIO_InitStructure.GPIO_Pin  |= GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15; // 7
    GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_OUT;                    // 8
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;                    // 9
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;                // 10
    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;                  // 11
    GPIO_Init(GPIOE, &GPIO_InitStructure);                             // 12
}
```

The syntax of statements follows the CMSIS standard.

- The first action is performed by the statement in the fourth line of the listing above where the function “RCC\_AHB1PeriphClockCmd” with two arguments is called. The first argument is the name of the port to manipulate the clock for (‘RCC\_AHB1Periph\_GPIOE’), and the second is the actual action (‘ENABLE’ or ‘DISABLE’). Please note that the last letter of the first argument defines the port to be one from A to E.
- The second action is performed by a call to a function “GPIO\_Init” in the last line of the function. This function requires two arguments, the first one being the name of the port, GPIOE in our case; please note that the last letter of this argument defines the port to manipulate. The second argument is a pointer to the data structure named ‘GPIO\_InitStructure’ containing the desired settings. This data structure is declared in the second line of the listing above, and then

initialized in lines 6 to 11. The structure has five members as defined in the “stm23f4xx\_gpio.h”, line 124. Each member can have different values as defined in the same file, lines 64 to 168. The names used tend to be human friendly and understandable. For reference:

- Lines 6 and 7 in the listing define the pins to be affected by initializing the member called ‘GPIO\_Pin’. In this example eight pins are used, and their names are or-ed together in two consecutive lines of the function. The names of pins are defined as hex values in “stm23f4xx\_gpio.h”, lines 151 to 168, and or-ing them builds a hex number that can be directly written into a designated register of the microprocessor.
- Line 8 of the listing defines the direction of the signal flow by initializing member called ‘Mode’ for pins selected by previous member of the structure. The options are input, output, alternate function or analog, and are declared in “stm23f4xx\_gpio.h”, lines 64 to 71. The user-friendly version reads as ‘GPIO\_Mode\_IN’, ‘GPIO\_Mode\_OUT’, ‘GPIO\_Mode\_AF’, and ‘GPIO\_Mode\_AN’ respectively.
- Line 9 defines the use of both MOS transistors in output driver by initializing the member ‘GPIO\_OType’. The options are: ‘GPIO\_OType\_PP’ (push-pull configuration, both transistors are used) and ‘GPIO\_OType\_OD’ (open-drain, only the lower transistor is used).
- Line 10 defines the speed of the selected pins by initializing the member ‘GPIO\_Speed’. The options are ‘GPIO\_Speed\_2MHz’, ‘GPIO\_Speed\_25MHz’, ‘GPIO\_Speed\_50MHz’, and ‘GPIO\_Speed\_100MHz’, as defined at lines 88 to 96 of “stm23f4xx\_gpio.h”.
- Line 11 defines the use of termination resistors by initializing the member ‘GPIO\_PuPd’. Valid options are ‘GPIO\_PuPd\_NOPULL’ (resistors not used), ‘GPIO\_PuPd\_UP’ (pull-up resistor used), and ‘GPIO\_PuPd\_DOWN’ (pull-down resistor used), as defined in lines 101 to 106 of the “stm23f4xx\_gpio.h”.

Similar use of header and source files can be traced throughout configuration of all peripherals using the CMSIS standard. Not all members of structure need to be initialized, since for certain combinations some settings are not possible; this will be demonstrated soon. A source file named “stm32f4xx\_\$\$\$\$.c” contains brief instructions on the use of functions and functions to manipulate the peripheral themselves, and the header file “stm32f4xx\_\$\$\$\$.h” defines the required data structures and options.

The complete listing of the program to increment the counter and send its content to port E, bits 8 to 16, is given below.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"

void GPIOEinit (void) {
GPIO_InitTypeDef GPIO_InitStructure;

RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10 | GPIO_Pin_11;
GPIO_InitStructure.GPIO_Pin |= GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOE, &GPIO_InitStructure);
}
```

```

void main (void) {
unsigned int number = 0;

GPIOEinit();

while (1) {                // about 50ns per iteration
    number += 0x100;        // increment at BIT_8
    GPIOE->ODR = number;    // send to PortE
};
}

```

The next example demonstrates the use of pushbuttons mounted onto the BaseBoard and LEDs available on the STM32F4-Discovery board. The program periodically reads the state of pushbuttons, and turns the LED that corresponds to the pushbutton ON when the pushbutton is pressed.

The pushbuttons are named S370 to S373, as shown in chapter 2, Fig. 2.2.2. They are connected to port E, pins 3 to 6. In order to read the value from pushbuttons the port E, pins 3 to 6 must be configured as inputs. When the pushbutton is pressed, the pin reads as logic high, otherwise the pin is left floating. However, we can use the pull-down resistors for port E, pins 3 to 6 to terminate pins to ground and un-float them. It is convenient again to prepare a function to initialize pins associated with pushbuttons, such function “SWITCHinit” is given below.

```

void SWITCHinit (void) {
GPIO_InitTypeDef      GPIO_InitStructure;

RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);

GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6;
GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_IN;
GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_DOWN;
GPIO_Init(GPIOE, &GPIO_InitStructure);
}

```

Please note that the ‘.GPIO\_mode’ member is defined as ‘GPIO\_Mode\_IN’, and that ‘.GPIO\_PuPd’ member is defined as ‘GPIO\_PuPd\_DOWN’. Members ‘GPIO\_Type’ and ‘.GPIO\_Speed’ have no meaning here since pins are used as inputs, and are not initialized.

A function for the configuration of pins associated with LEDs can be prepared similarly. LEDs are available at the STM32F4-Discovery board, and are wired to port D, pins 12 (green), 13 (orange), 14 (red), and 15 (blue). These pins must be configured as outputs, the function “LEDinit” is listed below.

```

void LEDinit (void)    {
GPIO_InitTypeDef      GPIO_InitStructure;

RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);                // 4

GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_OUT;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;                       // 9
GPIO_Init(GPIOD, &GPIO_InitStructure);
}

```

Please note the differences from the listing to initialize port E, pins 8 to 16 as outputs. Here the clock is enabled for port D, see line 4 of the listing. The direction of the signal flow is ‘OUT’, and both MOS transistors are used in push-pull configuration. The terminating resistors need not be specified, since these are output pins. The speed is reduced to 2MHz in line 9 thus saving power.

The rest of the program is straightforward, and is given below.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "dd.h"

void main (void) {
char switches = 0;
  SWITCHinit();
  LEDinit();

  while (1) {
    switches = GPIOE->IDR; // 12
    if (switches & S370) LED_GR_ON; else LED_GR_OFF;
    if (switches & S371) LED_OR_ON; else LED_OR_OFF;
    if (switches & S372) LED_RD_ON; else LED_RD_OFF;
    if (switches & S373) LED_BL_ON; else LED_BL_OFF;
  };
}
```

The program starts with three familiar include statements. The fourth line includes one addition file with define statements for pushbuttons and LEDs. Next come the functions “SWITCHinit” and “LEDinit” as listed above but not shown here again to avoid redundancy.

Within the “main” function a variable ‘switches’ is declared and initialized first, next two functions for the configuration of pins for pushbuttons and LEDs are called. Finally the endless ‘while’ loop is entered, where the microprocessor reads the status of pushbuttons into variable ‘switches’, line 12. The name of the Input data register for port E is ‘GPIOE->IDR’. After reading the input the microprocessor checks the variable ‘switches’ against four different patterns, and lights the corresponding LED. It is worth to mention that commands written in capital letters are defined in file “dd.h”. Two special output registers are used; by setting a bit in register GPIOD->BSRRL the corresponding pin of port D is set, and by setting a bit in output register GPIOD->BSRRH the corresponding bit is reset. This can streamline setting or resetting of individual bits over writing a completely new pattern into the data output register, as well as avoids corrupting the state of other pins when only one pin is to be affected.