CHAPTER 14

General Purpose I/O (GPIO)

This chapter illustrates how a processor uses a GPIO pin as a digital input or digital output. Example applications presented include lighting an LED, interfacing a push button and scanning a keypad.

14.1 Introduction to General Purpose I/O (GPIO)

The number of pins available on a processor is usually limited. A processor pin that can be configured by software at runtime to perform various functions is called a general-purpose input/output (GPIO) pin. GPIO provides high flexibility of use and enormous convenience of system design. It enables a processor to meet the needs of a broad range of embedded system applications. However, the flexibility comes with a price tag. The software has to perform a sophisticated initialization.

Software can program a GPIO pin as one of the following four different functions:

1. Digital input that detects whether an external voltage signal is higher or lower than a predetermined threshold
2. Digital output that controls the voltage on the pin
3. Analog functions that perform digital-to-analog or analog-to-digital conversion
4. Other complex functions such as PWM output, LCD driver, timer-based input capture, external interrupt, and interface of USART, SPI, I²C and USB communication

We call the last category of functions *alternative functions* (AF). The software can dynamically change the function of a GPIO pin at runtime. In this chapter, we focus on digital input and digital output, which are simply called input or output. Analog and other complex functions are introduced in the later chapters.

A GPIO port consists of a group of GPIO pins, typically 8 or 16, which share the same data and control registers.

- When a GPIO port is set as *digital input*, the binary data read from all pins of this GPIO group are combined into a word or halfword saved in the input data register (`IDR`). Each bit in `IDR` holds the digital input of the corresponding pin.
- When a GPIO port is configured as *digital output*, the output data register (`ODR`) holds the output of all pins of this port. Therefore, when changing the output of a GPIO pin, the programmer should only alter the value of the corresponding bit of `ODR`, without affecting the other bits of `ODR`. Chapter 4.6 introduces how to test, clear, set, and toggle a specific bit of a register in C and assembly.

14.2 GPIO Input Modes: Pull Up and Pull Down

When a GPIO pin is used as digital input, the pin has three states: high voltage, low voltage, or high impedance (also called floating or tri-stated).

Pull-up and pull-down are used to ensure the input pin has a valid high (logic `1`) or a valid low (logic `0`) when the external circuit does not drive the pin.

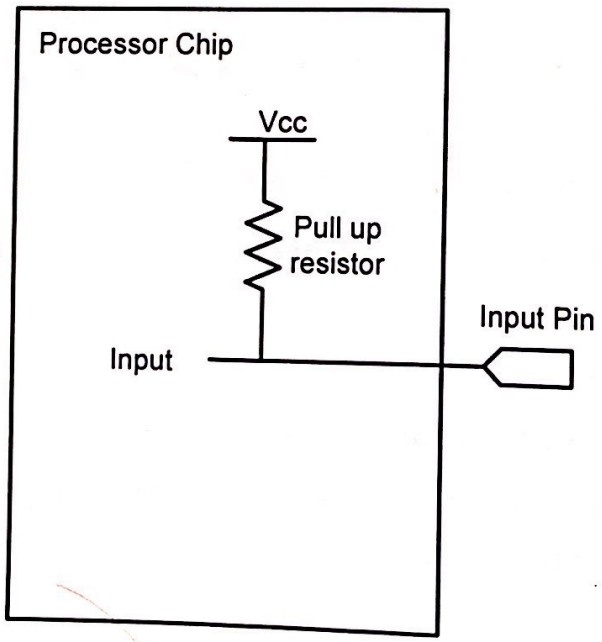

Figure 14-1. The GPIO pin is pulled up internally.

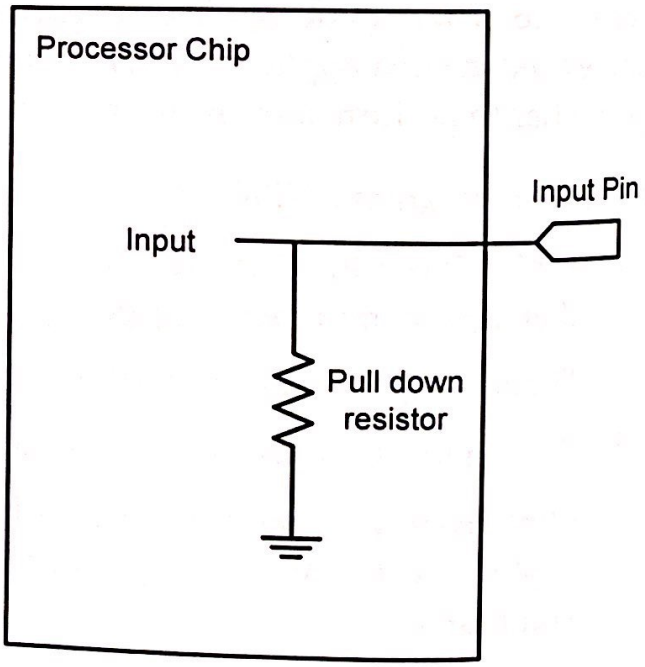

Figure 14-2. The GPIO pin is pulled down internally.

When software configures a pin as pull-up, the pin is internally connected to the power supply via a resistor, as shown in Figure 14-1. The pin is always read as high (logic 1) unless the external circuit drives this pin to low.

Similarly, when a pin is configured as pull-down, the pin is then internally connected to the ground via a resistor, as shown in Figure 14-2. The pin is always read as low (logic 0) unless the external circuit drives this pin to high.

When a pin is neither pulled up nor pulled down internally, then the pin has high impedance and the analog signal on the GPIO pin cannot reliably represent a logic value. Software can change the pull-up and pull-down setting of a GPIO pin dynamically at runtime.

When a pin is internally pulled up, but the external circuit drives the pin to low, a pull-up current is generated and is drawn internally from the processor chip. Similarly, when a pin is pulled down within the chip, but the external circuit drives the pin to high, a pull-down current is drawn to the processor chip. To limit the pull-up/pull-down current, the internal resistors usually have large impedance (> 10KΩ).

Strong vs Weak pull-up/pull-down

When an external circuit connected to a GPIO pin has a fair amount of capacitance, the process of pulling the pin voltage to the level of logic high or logic low takes a long time because the impedance of the pull-up and pull-down resistors is too large. We call pulling via large resistors *weak pull-up* or *weak pull-down*. The internal pulling often does not meet the speed requirement for fast communication protocols, such as I²C. In order to change the pin voltage rapidly, a GPIO pin can be externally pulled up or down via a smaller resistor (several KΩs). Pulling via small resistors is often called *strong pull-up* or *strong pull-down*.

14.3 GPIO Input: Schmitt Trigger

Each GPIO input module usually includes a Schmitt trigger. A Schmitt trigger uses a voltage comparator to convert a noisy or slow signal edge into a clean edge with instantaneous transition. In real systems, an input signal from external devices usually cannot change instantly. Such input signal tends to have a low slew rate (see definition in Chapter 14.5) because of inherent parasitic capacitance, resistance, or induction in the input data path. A processor chip usually has built-in Schmitt triggers to increase slew rate and enhance noise immunity for external input signals.

Figure 14-3 gives an example implementation of non-inverting Schmitt trigger with a reference voltage. The voltage comparator is an operational amplifier (op-amp) with positive feedback. The positive feedback is achieved by connecting the op-amp output to its non-inverting terminal (*i.e.*, the plus input lead).

The output voltage V_{out} responds rapidly to the difference between two input voltages V_+ and V_-. If V_+ is greater than V_-, V_{out} is quickly saturated to V_{SAT}; otherwise, V_{out} is zero in this example.

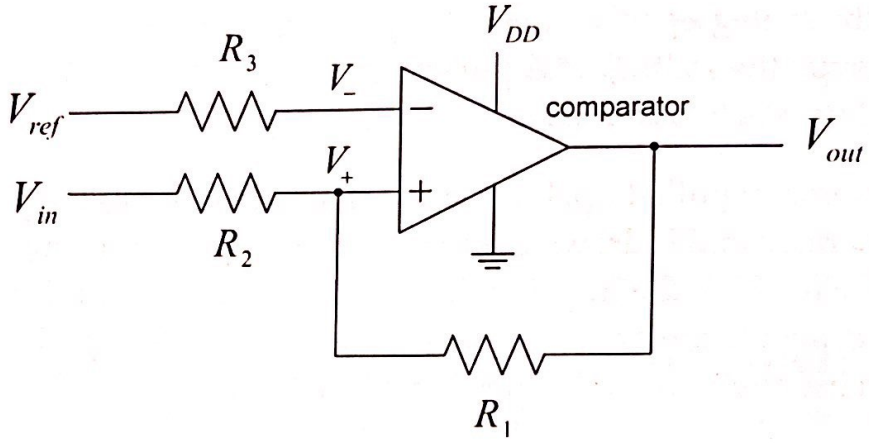

Figure 14-3. Circuit of non-inverting Schmitt trigger with hysteresis. Non-inverting means V_{in} is connected to the non-inverting terminal (*i.e.*, the plus input lead).

For an ideal op-amp, the current flowing through resistor R_3 is zero and thus we have

$$V_{ref} = V_-$$

The op-amp output V_{out} has two saturation values, as shown below

$$V_{out} = \begin{cases} V_{SAT} & if\ V_- < V_+ \\ 0 & if\ V_- > V_+ \end{cases}$$

However, V_+ depends on V_{out} and V_{in}. Therefore, V_{out} depends on both the input V_{in} and the recent history of V_{out}. Such an effect is called *hysteresis*.

Because the current flow into the positive input lead of the op-amp is assumed to be zero for an ideal op-amp, we can obtain the following equation by applying Kirchhoff's Current Law (KCL):

$$\frac{V_{in} - V_+}{R_2} = \frac{V_+ - V_{out}}{R_1}$$

Using the above equation, we can obtain the following expression:

$$V_+ = \frac{R_2}{R_1 + R_2} V_{out} + \frac{R_1}{R_1 + R_2} V_{in}$$

Figure 14-3 gives an example implementation of non-inverting Schmitt trigger with a reference voltage. The voltage comparator is an operational amplifier (op-amp) with positive feedback. The positive feedback is achieved by connecting the op-amp output to its non-inverting terminal (*i.e.*, the plus input lead).

The output voltage V_{out} responds rapidly to the difference between two input voltages V_+ and V_-. If V_+ is greater than V_-, V_{out} is quickly saturated to V_{SAT}; otherwise, V_{out} is zero in this example.

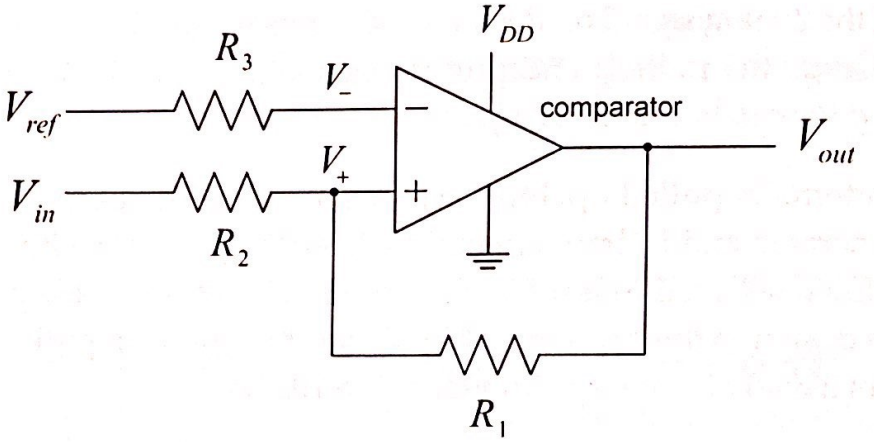

Figure 14-3. Circuit of non-inverting Schmitt trigger with hysteresis. Non-inverting means V_{in} is connected to the non-inverting terminal (*i.e.*, the plus input lead).

For an ideal op-amp, the current flowing through resistor R_3 is zero and thus we have

$$V_{ref} = V_-$$

The op-amp output V_{out} has two saturation values, as shown below

$$V_{out} = \begin{cases} V_{SAT} & if\ V_- < V_+ \\ 0 & if\ V_- > V_+ \end{cases}$$

However, V_+ depends on V_{out} and V_{in}. Therefore, V_{out} depends on both the input V_{in} and the recent history of V_{out}. Such an effect is called *hysteresis*.

Because the current flow into the positive input lead of the op-amp is assumed to be zero for an ideal op-amp, we can obtain the following equation by applying Kirchhoff's Current Law (KCL):

$$\frac{V_{in} - V_+}{R_2} = \frac{V_+ - V_{out}}{R_1}$$

Using the above equation, we can obtain the following expression:

$$V_+ = \frac{R_2}{R_1 + R_2} V_{out} + \frac{R_1}{R_1 + R_2} V_{in}$$

At the time instant when V_{out} transits from one saturation value to the other saturation value, we have

$$V_+ = V_{ref}$$

Thus

$$\frac{R_2}{R_1 + R_2} V_{out} + \frac{R_1}{R_1 + R_2} V_{in} = V_{ref}$$

Solving the above equation, we have

$$V_{in} = \left(1 + \frac{R_2}{R_1}\right) V_{ref} - \frac{R_2}{R_1} V_{out}$$

As discussed earlier, V_{out} has only two possible values. If $V_{out} = 0$ initially and V_{in} increases, we can obtain the trigger high threshold V_{TH} at which V_{out} transits to V_{SAT}:

$$V_{TH} = \left(1 + \frac{R_2}{R_1}\right) V_{ref} - \frac{R_2}{R_1} \times 0 = \left(1 + \frac{R_2}{R_1}\right) V_{ref}$$

On the other hand, if $V_{out} = V_{SAT}$ initially and V_{in} decreases, we can obtain the trigger low threshold V_{TL} at which V_{out} transits to 0:

$$V_{TL} = \left(1 + \frac{R_2}{R_1}\right) V_{ref} - \frac{R_2}{R_1} V_{SAT}$$

Therefore, V_{out} can be determined by comparing it with two thresholds V_{TH} and V_{TL}. Figure 14-4 shows the relationship of V_{out} and V_{in}. When V_{in} climbs through V_{TH}, V_{out} is rapidly switched to the upper limit V_{SAT}. Conversely, once V_{in} falls below V_{TL}, V_{out} makes a transition to the lower limit. Note that $V_{TH} > V_{TL}$, *i.e.* the threshold for switching to high is greater than the threshold of switching to low.

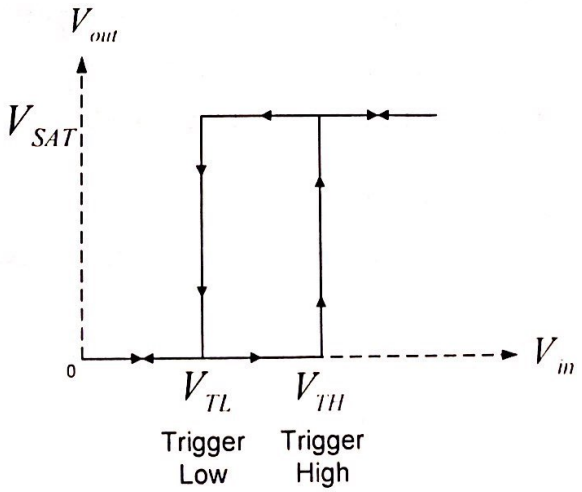

Figure 14-4. Relationship between V_{out} and V_{in} of inverting Schmitt trigger with reference voltage. V_{TL} and V_{TH} are the low and high switching thresholds.

Figure 14-5 compares the output voltage V_{out} of Schmitt trigger and a simple comparator when the input signal varies irregularly. Compared with a simple comparator, Schmitt trigger provides better noise rejection. The threshold of Schmitt trigger is larger than that of a simple comparator for switching high, and lower for switching low. If the input signal fluctuates slightly, the output of Schmitt trigger does not change. For this reason, Schmitt trigger is immune to undesired noise.

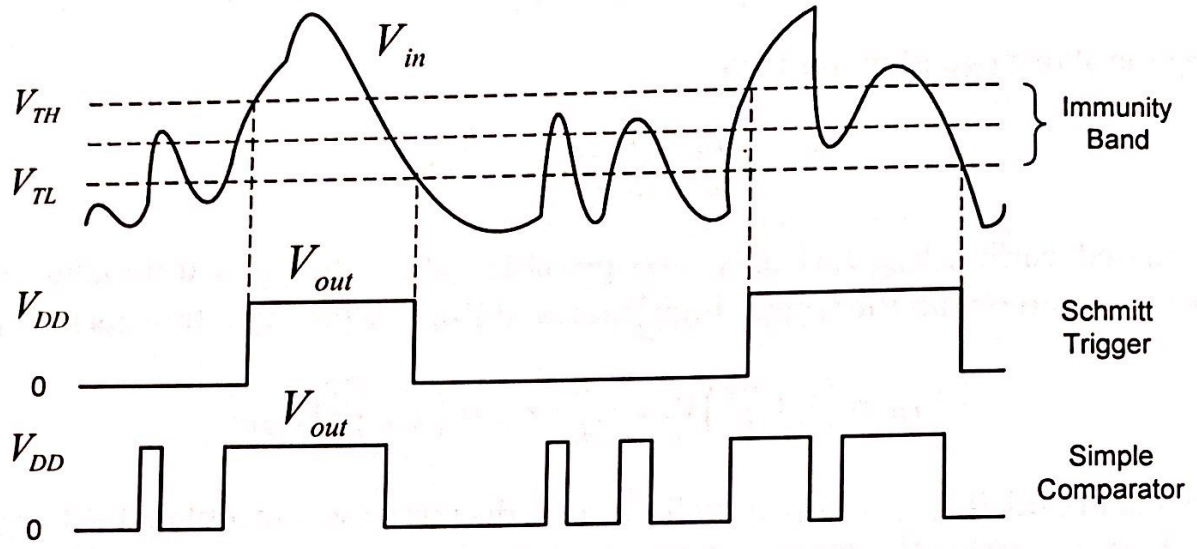

Figure 14-5. Comparing the voltage output of Schmitt trigger with a simple comparator. Schmitt trigger converts an irregular-shaped signal V_{in} into a square wave V_{out} based on two switching thresholds. The simple comparator uses a single threshold (the dotted line between V_{TL} and V_{TH}) to generate the output.

14.4 GPIO Output Modes: Push-Pull and Open-Drain

Software can configure a GPIO output pin as either push-pull or open-drain. The push-pull mode allows the pin to supply and absorb current. However, a GPIO pin in the open-drain (also called collector) mode can only absorb current.

14.4.1 GPIO Push-Pull Output

A push-pull output consists of a pair of complementary transistors, as shown in Figure 14-6. Only one of them is turned on at any time.

- When logic `0` is outputted, the transistor connected to the ground is turned on to sink an electric current from the external circuit, as shown in Figure 14-7.
- When the pin outputs logic `1`, the transistor connected to the power supply is turned on, and it provides an electric current to the external circuit connected to the output pin, as shown in Figure 14-8.

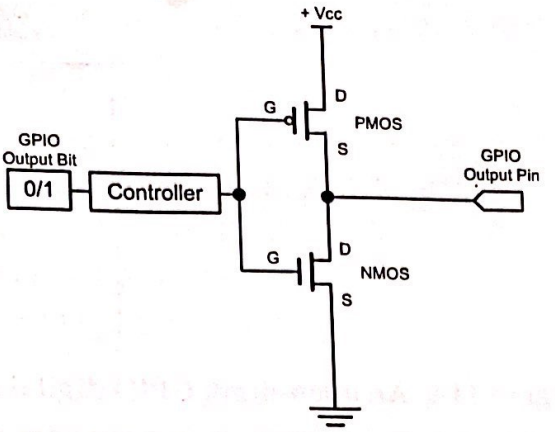

Figure 14-6. A push-pull GPIO digital output

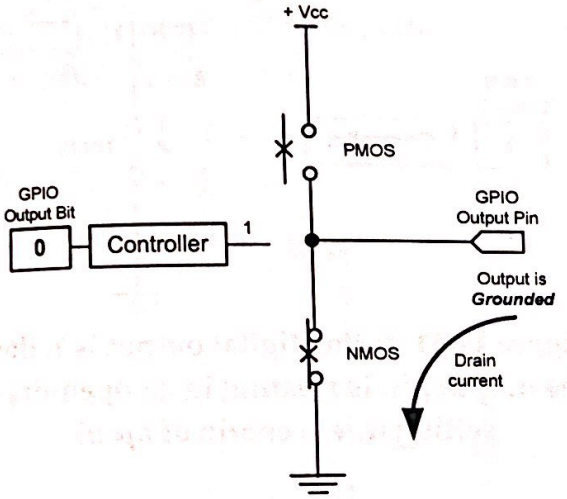

Figure 14-7. If the digital output is 0, then the GPIO output pin is pulled down to the ground in a push-pull setting.

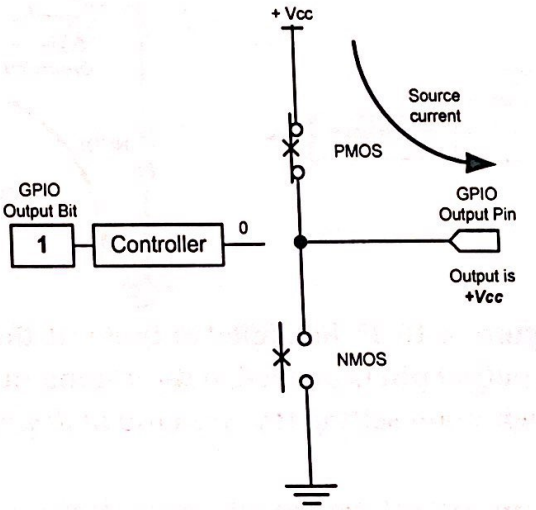

Figure 14-8. If the digital output is 1, then the GPIO output pin is pulled up to the Vcc in a push-pull setting.

14.4.2 GPIO Open-Drain Output

An open-drain output consists of a pair of the same type of CMOS or transistors, as shown in Figure 14-9.

- When software outputs a logic `0`, the open-drain circuit can sink an electric current from the external load connected to the GPIO pin.
- However, when software outputs a logic 1, it cannot supply any electric current to the external load because the output pin is floating, connected to neither the power supply nor the ground.

An open-drain output has only two states: low voltage (logic `0`), and high impedance (logic 1). It often has an external pull-up resistor.

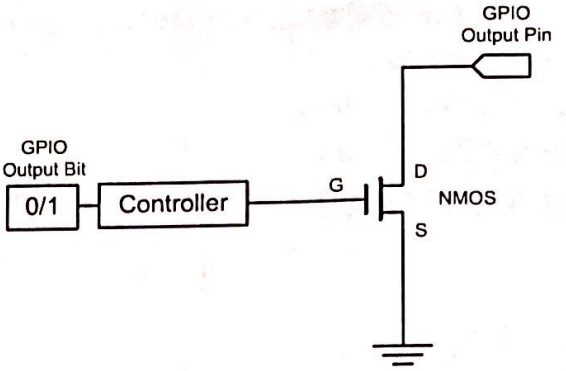

Figure 14-9. An open-drain GPIO digital output

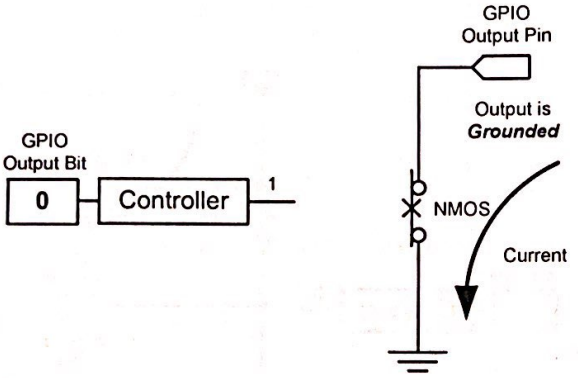

Figure 14-10. If the digital output is 0, then the output pin is pushed to the ground in an open-drain setting (the scenario of *drain*).

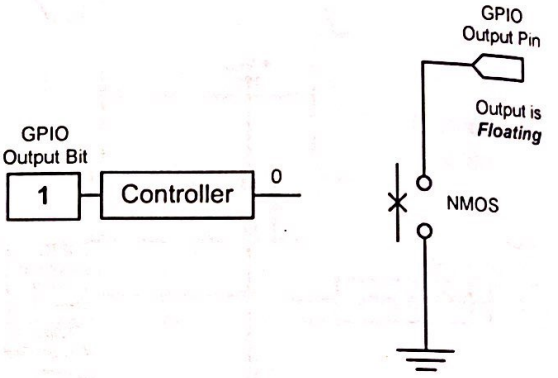

Figure 14-11. If the digital output is 1, then the output pin is floating in an open-drain setting (the scenario of *open*).

One important usage of open-drain outputs is to connect directly several outputs together and implement wired logic AND (active high) or OR (active low) circuit in a simple way. If multiple open-drain output pins are connected and are pulled up via a shared resistor, any output pin can drive the output voltage to low. The pin voltage is high if and only if all pins output a high voltage level.

- If a high voltage level represents logic state 1 (*i.e.* active high), it implements a wired-AND function. The final output is 1 (high) only if all outputs of connected pins are 1 (high).
- If a low voltage level represents logic state 1 (*i.e.* active low), it implements a wired-OR function. The final output is 1 (low) if the output of any pins is 1 (low).

For example, the I2C communication protocol uses wired-OR to allow multiple master devices to operate on the same bus.

Compared to open-drain, the push-pull mode has the advantage of faster speed, because it can change the pin voltage faster if the external circuit has some capacitance. Another advantage is that it can supply current and simplify the circuit. For example, a push-pull

output can directly control an external LED while an open-drain output cannot light up an LED without external voltage source.

However, the wired-OR characteristics can only be provided in open-drain outputs. Usually, push-pull output pins cannot be directly connected, because it might cause a potential short circuit. Additionally, open-drain output allows the pin to be pulled up to any voltage. This feature can be helpful when a GPIO pin is used as an input to another system that requires a higher level of input voltage.

14.5 GPIO Output Speed: Slew Rate

The slew rate of a GPIO pin is the rate of change of its output voltage per unit of time, as defined as follows.

$$Slew\ Rate = \frac{\Delta V}{\Delta t}$$

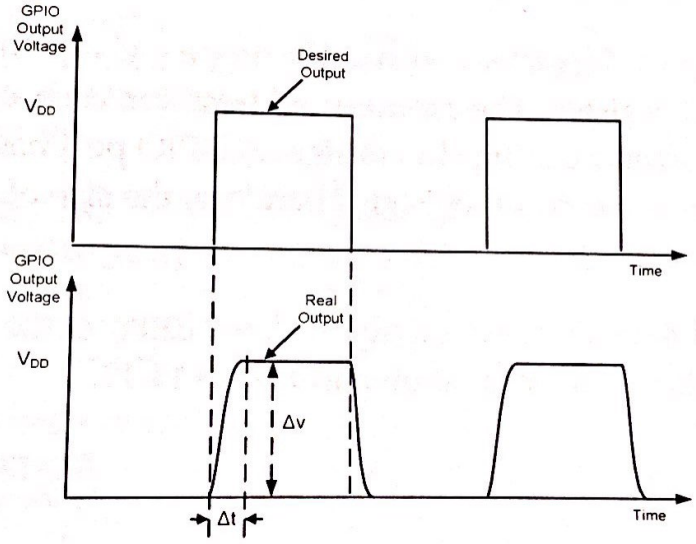

Figure 14-12. Comparing a desired square wave voltage output with the real GPIO output

If the logic output of a GPIO pin changes from `0` to `1` and accordingly the voltage output of this pin rises from 0V to 3V in 3μs, then the slew rate is 1 volt per μs. Figure 14-12 shows an example of ΔV and Δt when the output voltage increases from low to high. The slew rate definition applies to both the rising edge and the falling edge of a voltage output.

The higher the slew rate, the shorter time the output voltage takes to rise or fall to desired values. Therefore, a higher slew rate allows faster speed at which the processor can toggle the logic level of a GPIO pin. Figure 14-12 also compares the desired square wave output and the real output when the logic output of a GPIO pin is toggled periodically. A shorter rise and fall time allows a GPIO pin to change its logic value more rapidly.

However, a large slew rate often causes high electromagnetic interference (EMI), also called radio frequency interference (RFI) to neighbor electronic circuits. A fast rising and falling signal has large amplitude and high-frequency harmonics, which can transfer to a victim circuit via radiation, conduction, or induction, and may cause malfunctions. A slower valid slew rate is often preferred to minimize EMI disturbance.

The slew rate of the GPIO circuit is programmable by setting the GPIO output speed. For example, the digital output speed of a GPIO pin can be 400 KHz, 2 MHz, 10 MHz or 40 MHz in the STM32L processors.

14.6 Lighting up an LED

The following shows the basic procedure to light up an LED. The software initialization involves two key steps. First, it changes RCC to enable the clock of the GPIO port B. Second, it configures GPIO registers to set the pin 6 of GPIO port B as a general-purpose output with push-pull. To light up the blue LED, we need to output logic "1" to pin 6.

A GPIO port has ten control registers, such as the mode register, the output type register, and the output speed register. The memory address for each control register is pre-defined by the manufacturer during chip design. A GPIO port has 16 pins, and each pin may take 1, 2, or 4 bits in a control register. Therefore, the size of these control registers can be 2, 4, or 8 bytes.

To access each control register conveniently in C, we can cast the base memory address of a GPIO port to a data structure, as shown in Figure 14-13.

```
typedef struct{
  __IO uint32_t MODER;  // Mode register
  __IO uint16_t OTYPER; // Output type register
       uint16_t rev0;   // Padding two bytes
  __IO uint32_t OSPEEDR;// Output speed register
  __IO uint32_t PUPDR;  // Pull-up/pull-down register
  __IO uint16_t IDR;    // Input data register
       uint16_t rev1;   // Padding two bytes
  __IO uint16_t ODR;    // Output data register
       uint16_t rev2;   // Padding two bytes
  __IO uint16_t BSRRL;  // Bit set/reset register(low)
  __IO uint16_t BSRRH;  // Bit set/reset register(high)
  __IO uint32_t LCKR;   // Configuration lock register
  __IO uint32_t AFR[2]; // Alternate function registers
} GPIO_TypeDef

#define GPIOB ((GPIO_TypeDef *) 0x4002400))
```

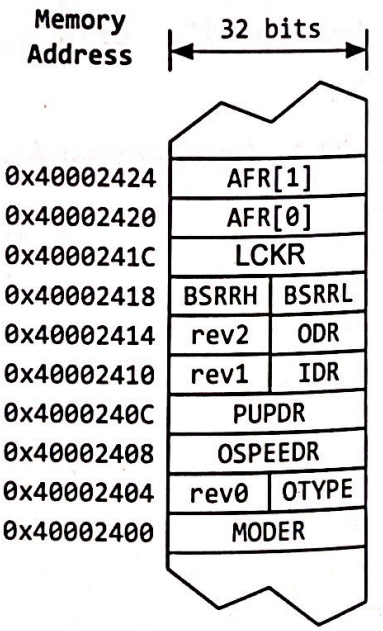

Figure 14-13. Casting a memory address to the GPIO structure.

Six bytes are padded in the GPIO_TypeDef structure, making its structure members to align properly with their pre-defined memory addresses. The following C statement casts the memory address defined by the chip manufacturer to the GPIO control structure.

```
#define GPIOB ((GPIO_TypeDef *) 0x4002400))
```

If we want to set the output the GPIO pin 6 to high, we can use the following C statement:

```
GPIOB->ODR |= 1<<6;     // Set bit[6] to 1
```

Note the pin number of a port starts with 0, instead of 1. In assembly, a load-modify-store sequence is required to change the register value stored in memory. The following gives corresponding implementation in assembly language.

```
LDR r7, =0x40020400          ; Load GPIO port B base address
LDR r1, [r7, #20]            ; Byte offset of ODR is 20
ORR r1, r1, #(1<<6)          ; Set bit 6
STR r1, [r7, #20]            ; Write ODR
```

We can use directive "EQU" to substitute a constant with a symbol, which makes the assembly program more readable and self-documenting. The following is an example.

```
GPIOB_BASE EQU 0x40020400
GPIO_ODR   EQU 20

LDR r7, =GPIOB_BASE          ; Load GPIO port B base address
LDR r1, [r7, #GPIO_ODR]      ; r1 = GPIOB->ODR
ORR r1, r1, #(1<<6)          ; set bit 6
LDR r1, [r7, #GPIO_ODR]      ; Write GPIOB->ODR
```

Additionally, we also need to enable the clock of GPIO port B. To save energy, the clock of all peripherals is turned off by default. We can enable the clock of a peripheral by setting the corresponding bit of the clock control register defined in the reset and clock control (RCC) structure, as shown below.

```
// Reset and clock control
typedef struct {
  __IO uint32_t CR;         // Clock control register
  __IO uint32_t ICSCR;      // Internal clock sources calibration register
  __IO uint32_t CFGR;       // Clock configuration register
  __IO uint32_t CIR;        // Clock interrupt register
  __IO uint32_t AHBRSTR;    // AHB peripheral reset register
  __IO uint32_t APB2RSTR;   // APB2 peripheral reset register
  __IO uint32_t APB1RSTR;   // APB1 peripheral reset register
  __IO uint32_t AHBENR;     // AHB peripheral clock enable register
  __IO uint32_t APB2ENR;    // APB2 peripheral clock enable register
  __IO uint32_t APB1ENR;    // APB1 peripheral clock enable register
  __IO uint32_t AHBLPENR;   // AHB peripheral clock enable in low power mode register
  __IO uint32_t APB2LPENR; // APB2 peripheral clock enable in low power mode register
```

```
  __IO uint32_t APB1LPENR; // APB1 peripheral clock enable in low power mode register
  __IO uint32_t CSR;       // Control/status register
} RCC_TypeDef;

#define RCC ((RCC_TypeDef *) 0x40023800))
```

The following C statements enable the clock of GPIO port B.

```
#define RCC_AHBENR_GPIOBEN  (0x00000002)
RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
```

Figure 14-14 shows the flowchart of initializing a GPIO pin as digital output with push-pull.

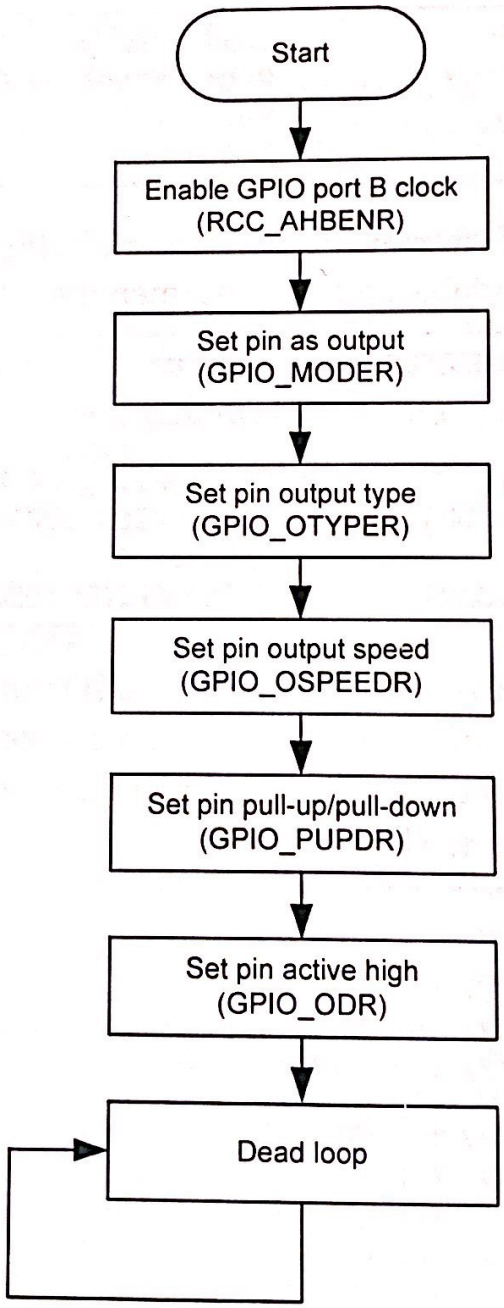

Figure 14-14. Flowchart of GPIO initialization

The following C program demonstrates how to set up a GPIO pin and light up an LED in detail. Suppose we use the GPIO pin PB 6 to drive a green LED.

- When we change the value of specific bits in a register, we need to preserve the value of the other bits in this register to avoid creating unexpected negative impacts. For example, if we want to set the least significant bit in register R, "R = 0x1;" is incorrect because it also clears all the other bits. Instead, we should use a bitwise logical OR operation "R |= 0x1;".
- When we change the value of multiple bits, it is a good practice to reset these bits before updating them. For example, if we want to set the least significant four bits $b_3b_2b_1b_0$ in register R to 1001, we need to clear these four bits first by running "R &= ~0xF; R |= 0x9;" If we do not clear these four bits first, we may fail to set the register correctly if their initial values are not 0. For example, if the value of $b_3b_2b_1b_0$ is 0111 initially, "R |= 0x9;" will lead a binary result of 1111.

Each GPIO port has a data output register (ODR) and a data input register (IDR).

- Each bit in ODR controls the output of a corresponding GPIO pin in this port. In a push-pull setting, if the bit value is 1, the output voltage on its corresponding GPIO pin is high; if the bit value is 0, the output voltage then is low.
- The DIR register records the input of all pins of a GPIO port.

```
// Blue LED is connected PB 6 (GPIO port B pin 6)

void GPIO_Clock_Enable(){
   // Enable the clock to GPIO port B
   RCC->AHBENR |= 0x00000002;
}

void GPIO_Pin_Init(){

   // Set pin 6 I/O mode as general-purpose output
   // 00 = digital input(default),    01 = digital output
   // 10 = alternative function,      11 = analog
   GPIOB->MODER &= ~(0x03<<(2*6));          // Mode mask
   GPIOB->MODER |= 0x01<<(2*6);             // Set pin 6 as digital output

   // Set output type of pin 6 as push-pull
   // 0 = push-pull(0, default), 1 = open-drain
   GPIOB->OTYPER &= ~(1<<6);

   // Set I/O output speed
   // 00 = 400 KHz(00), 01 = 2 MHz, 10 = 10 MHz, 11 = 40 MHz
   GPIOB->OSPEEDR &= ~(0x03<<(2*6));        // Speed mask
   GPIOB->OSPEEDR |=  0x01<<(2*6);          // Set output as 2 MHz
```

```
    // Set I/O as no pull-up pull-down
    // 00 = no pull-up, no pull-down (default),
    // 01 = pull-up, 10 = pull-down, 11 = reserved
    GPIOB->PUPDR &= ~(0x03<<(2*6));          // no pull-up, no pull-down
}

int main(void){
    GPIO_Clock_Enable();
    GPIO_Pin_Init();
    GPIOB->ODR |= 1<<6;  // Set bit 6 of output data register (ODR)
    while(1);            // Dead loop & program hangs here
}
```

Example 14-1. Lighting up an LED in C

The implementation in assembly is similar to the above C program. In the program, the GPIO_BASE and RCC_BASE are pre-defined memory addresses, and GPIO_MODER, GPIO_OTYPER, GPIO_OSPEEDR, GPIO_PUPDR, and GPIO_ODR are byte offset of its corresponding variable in the data structure GPIO_TypeDef defined previously.

It is a good practice to define a frequently used constant as some symbols associated with meaningful semantics. We can use the "EQU" directive to define symbols in assembly. This practice can effectively make a program easier to read and debug.

```
; Memory addresses of GPIO port B and RCC (reset and clock control) data
; structure. These addresses are predefined by the chip manufacturer.
GPIOB_BASE          EQU   0x40020400
RCC_BASE            EQU   0x40023800

; Byte offset of each variable of the GPIO_TypeDef structure
GPIO_MODER          EQU   0x00
GPIO_OTYPER         EQU   0x04
GPIO_RESERVED0      EQU   0x06
GPIO_OSPEEDR        EQU   0x08
GPIO_PUPDR          EQU   0x0C
GPIO_IDR            EQU   0x10
GPIO_RESERVED1      EQU   0x12
GPIO_ODR            EQU   0x14
GPIO_RESERVED2      EQU   0x16
GPIO_BSRRL          EQU   0x18
GPIO_BSRRH          EQU   0x1A
GPIO_LCKR           EQU   0x1C
GPIO_AFR0           EQU   0x20 ;  AFR[0]
GPIO_AFR1           EQU   0x24 ;  AFR[1]
GPIO_AFRL           EQU   0x20
GPIO_AFRH           EQU   0x24

; Byte offset of variable AHBENR in the RCC_TypeDef structure
RCC_AHBENR          EQU   0x1C
```

```
; Enable the clock to GPIO port B
LDR r7, =RCC_BASE    ; Load address of reset and clock control (RCC)
; LDR is a pseudo-instruction and the compiler translates it to
; multiple real instructions
LDR r1, [r7, #RCC_AHBENR]     ; r1 = RCC->AHBENR
ORR r1, r1, #0x00000002       ; Set bit 2 of AHBENR
STR r1, [r7, #RCC_AHBENR]     ; GPIO port B clock enable

; Set pin 6 I/O mode as General-purpose Output
LDR r7, =GPIOB_BASE           ; Load GPIO port B base address
LDR r1, [r7, #GPIO_MODER]     ; r1 = GPIOB->MODER
BIC r1, r1, #(0x03 << 12)     ; Direction mask pin 6, clear bit 13 and 12
ORR r1, r1, #(0x1 << 12)      ; Set mode as digital output (mode = 0b01)
STR r1, [r7, #GPIO_MODER]     ; Save the mode

; Set pin 6 the push-pull mode for the output type
LDR r1, [r7, #GPIO_OTYPER]    ; r1 = GPIOB->OTYPER
BIC r1, r1, #(1<<6)           ; Push-pull(0, default), open-drain(1)
STR r1, [r7, #GPIO_OTYPER]    ; Save output type

; Set I/O output speed value as 2 MHz
LDR r1, [r7, #GPIO_OSPEEDR]   ; r1 = GPIOB->OSPEEDR
BIC r1, r1, #(0x03<<12)       ; Speed mask for pin 6
ORR r1, r1, #(0x03<<12)       ; 400KHz(00), 2MHz(01), 10MHz(01), 40MHz(11)
STR r1, [r7, #GPIO_OSPEEDR]   ; Save output speed

; Set I/O as no pull-up pull-down
LDR r1, [r7, #GPIO_PUPDR]     ; r1 = GPIOB->PUPDR
BIC r1, r1, #(0x03<<12)       ; PUPD mask for pin 6
ORR r1, r1, #(0x00<<12)       ; No PUPD(00, reset), pull-up(01)
STR r1, [r7, #GPIO_PUPDR]     ; Save pull-up and pull-down setting

; Light up the LED
LDR r7, =GPIOB_BASE           ; Load GPIO port B base address
LDR r1, [r7, #GPIO_ODR]       ; r1 = GPIOB->ODR
ORR r1, r1, #(1<<6)           ; Set output of pin 6 to high
STR r1, [r7, #GPIO_ODR]       ; Write the output data register
```

Example 14-2. Lighting up an LED in an assembly program

14.7 Push Button

When a push button is pressed, two metal contacts bang together and immediately rebound a couple of times before setting. These rebounds produce multiple signals within a few milliseconds due to the bounce effects. Figure 14-15 shows the voltage signal

across a push button when it is pressed at the time instant 0. Because the processor runs at a fast speed, the processor can observe these falling and rising transitions and mistakenly thinks the push button has been pressed multiple times.

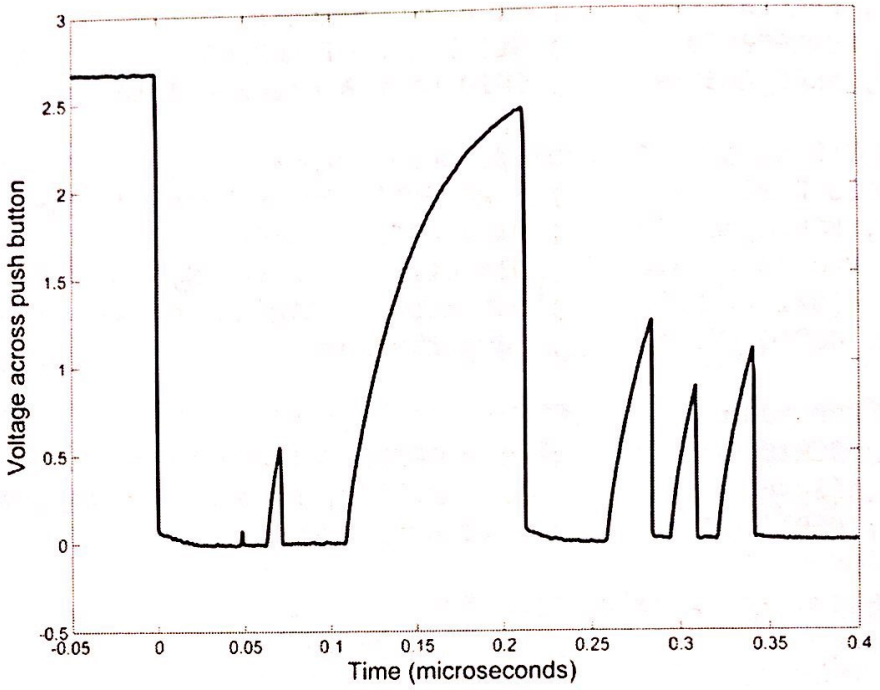

Figure 14-15. The voltage across a push button when there is no hardware debouncing

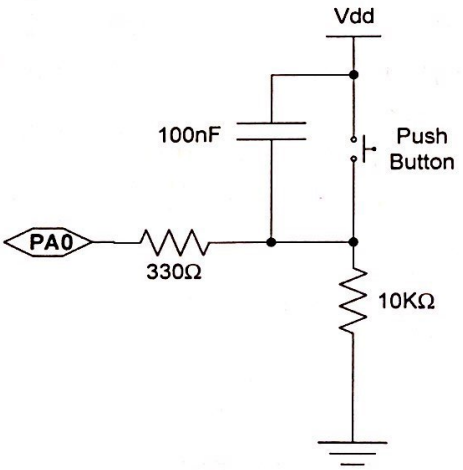

Figure 14-16. A push button with hardware debouncing

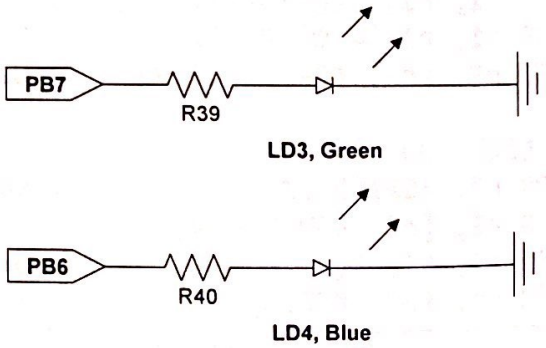

Figure 14-17. Blue and green LED

There are both hardware and software solutions to eliminate the bouncing effects. These solutions are called *debouncing*.

The hardware debouncing usually uses a simple RC circuit, which includes a capacitor connected in parallel with the pushbutton to filter out any high-frequency signals, as shown in Figure 14-16.

Figure 14-18 shows the voltage signals when the LED is lit up after the push button is pressed. It shows that the voltage on the pin (PA 0) connected to the push button rises smoothly without generating any bouncing signals. In this example, the processor constantly checks whether the button is pressed. The time span between successfully detecting a push event and lighting up the LED is approximate 12 μs in this example.

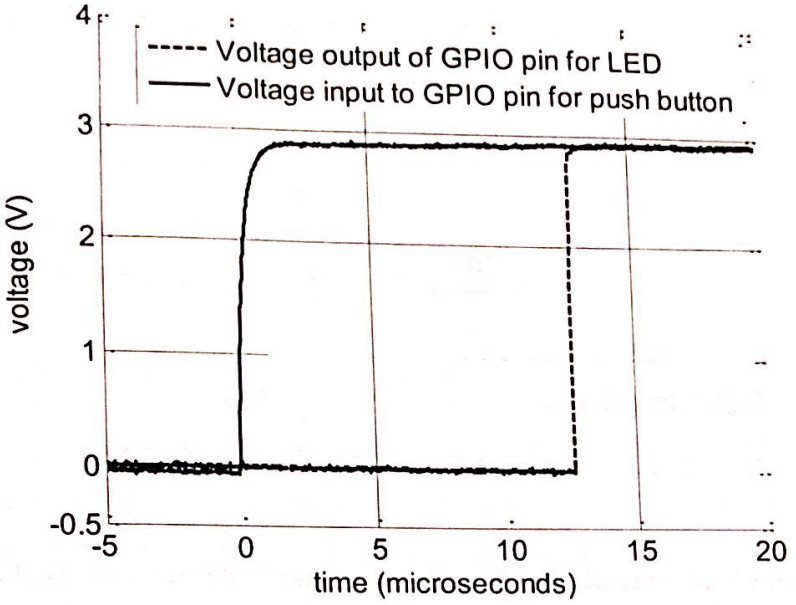

Figure 14-18. Voltage signals on the LED pin when the push button is pressed

Software debouncing re-examines the signal after a short delay. When a voltage transition from high to low is detected, the software checks the signal input after several milliseconds. If the signal is still low after the delay, the button is pushed.

The STM32L discovery kit has two push buttons on board, one for users, and the other for reset. The USER push button is connected to the GPIO Port A Pin 0 (PA 0) of the STM32L processor chip. The RESET push button is used to reset the STM32L processor chip to restart the processor. We use the USER push button to control the blue LED. When the button is pressed, the blue LED is toggled.

The program uses a polling I/O method (*busy waiting*) that constantly queries the input of external devices. Software repeatedly checks whether the push button is pressed or not. Although the polling method is simple, it is inherently inefficient because the CPU wastes many cycles on querying or waiting for input.

If there are many input devices, this polling method is not recommended because the time required to poll them can exceed the time available to service a given I/O input. A method based on interrupts is more efficient than polling. See Chapter 11.5 for details of external interrupt.

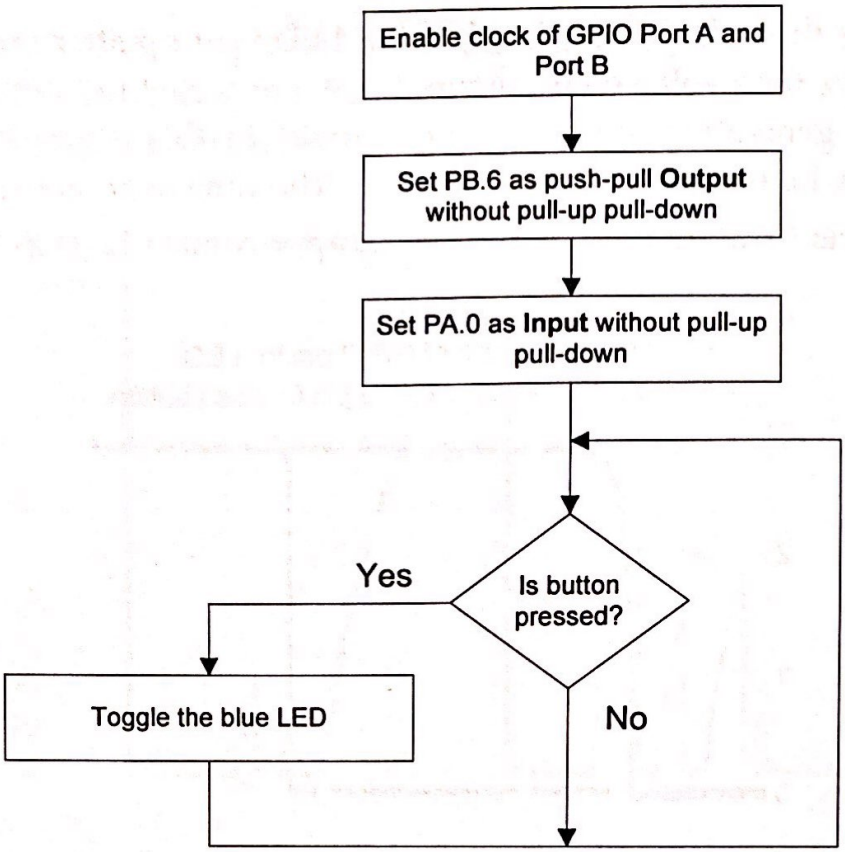

Figure 14-19. Flowchart of programming a pushbutton to control an LED

Suppose a push button is connected to the GPIO pin PA 0. We can set up the clock and the pin configuration as follows. The input of pin 0 is saved at bit 0 in the input data register (IDR) of GPIO port A. A low voltage input yields to a value of 0, and a high voltage generates a value of 1.

```
// Enable the clock to GPIO port A
RCC->AHBENR  |= RCC_AHBENR_GPIOAEN;

// Set pin 0 I/O mode as general-purpose input
// input(00, default), output(01), alternative function(10), analog(11)
GPIOA->MODER &= ~(0x03);          // Set mode as input (00)

// Set pin 0 the output type as push-pull
GPIOA->OTYPER &= ~(0x1);          // push-pull(0, reset), open-drain(1)

// Set I/O output speed value as 2 MHz
// 400 KHz(00), 2 MHz(01), 10 MHz(10), 40 MHz(11)
GPIOA->OSPEEDR &= ~(0x03);        // Speed mask
GPIOA->OSPEEDR |=  0x01;

// Set I/O as no pull-up pull-down
// No pull-up/pull-down (00, default),
// pull-up(01), pull-down(10), reserved(11)
GPIOA->PUPDR &= ~(0x03);          // Pull-up pull-down mask
```

14.8 Keypad Scan

Suppose we have a keypad that has 12 keys, as shown in Figure 14-20. One simple way is to interface each key in the same approach as a push button, with each key having a dedicated pin to detect whether it is pressed or not. However, this would require 12 I/O pins, which is not desirable for many applications because the total number of pins available for use on a microcontroller is limited. To reduce the number of pins required, a keypad usually organizes its keys in a matrix, as shown in Figure 14-21. This matrix scheme decreases the number of I/O pins from 12 to 7 in this example.

On most processors, a GPIO pin provides only weak pull-up and weak pull-down internally. The internal pull-up and pull-down circuit consists of a 60KΩ resistor in series with a switchable PMOS/NMOS. The pull-up and pull-down configuration bits of a GPIO pin turn on or turn off the PMOS and NMOS.

When the load has a fair amount of capacitance, applications often require a strong pull-up or pull-down to shorten the rising or falling time of the voltage signal on a pin. In a strong pull-up or strong pull-down setting, the pin should be externally connected to the ground or high voltage via a resistor with a much lower resistance than the internal pull-up and pull-down resistors. In Figure 14-21, each pin connected to the input port (`C1`, `C2`, and `C3`) is pulled up to 3.3V via a 2.2KΩ resistor. Because these pins are externally pulled up, they should be configured as no pull-up and no pull-down internally.

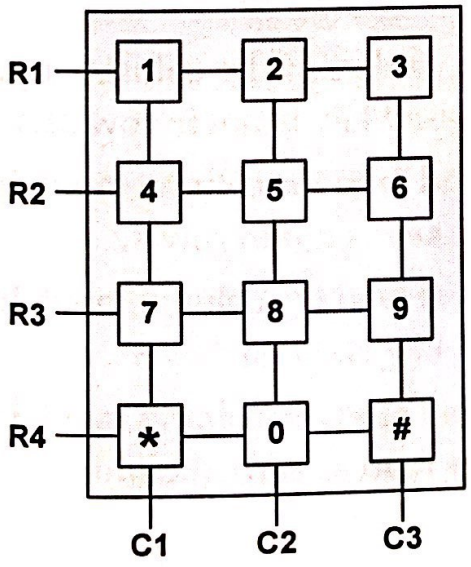

Figure 14-20. 3×4 keypad

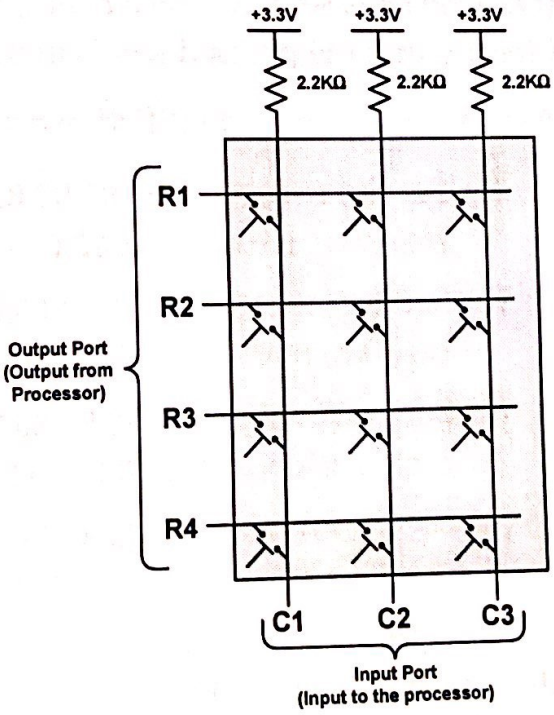

Figure 14-21. Input and output setting

Scanning algorithm is widely used to detect which key is pressed. The algorithm has two iterations of loops: looping over the row pins and then looping over the column pins. Suppose all row pins are set as output and all column pins are set as input. Each column pin is pulled up to a high-level voltage via a small resistor. The algorithm involves two steps.

1. Identify the column number of the pressed key. Set the output of all row pins as zero and read all column pins. If all columns are read as `1`, then no key has been pressed. If one of them is zero, then at least one of the keys in that corresponding column is pressed down.
2. Identify the row number of the pressed key. Drive the output of the first row to low (zero) while keeping the other rows at high (one). For example, suppose the input of column `C2` is read as zero. If the input of `C2` is still zero when the output of row `R1` is high, then the pressed key is not located in row `R1`. Otherwise, row `R1` is the row in which the pressed key is located. We repeat the process for all the other rows until the row is identified successfully.

The following gives a simple example how the scanning algorithm works when the key "`0`" is pressed.

1. Before key "`0`" is pressed, the row output port is set as low, *i.e.* `R1,R2,R3,R4 = 0000`. If the input port is read now, `C1`, `C2`, and `C3` are read as one, *i.e.* `C1,C2,C3 = 111`.
2. When key "`0`" is pressed, the column `C2` is connected to the ground via the `R4` pin (because `R4` is set to `0`). As a result, we have `C1,C2,C3 = 101`. Thus, we successfully identify that the pressed key is in the `C2` column.
3. After the column is identified, we scan the output row by row.
   (1) Set the row output (`R1,R2,R3,R4`) as `0111`, and read the column input. In this case, we have `C1,C2,C3 = 111`. The pressed key is not in row `R1`.
   (2) Set the row output (`R1,R2,R3,R4`) as `1011`, and read the column input. In this case, we have `C1,C2,C3 = 111`. The pressed key is not in row `R2`.
   (3) Set the row output (`R1,R2,R3,R4`) as `1101`, and read the column input. In this case, we have `C1,C2,C3 = 111`. The pressed key is not in row `R3`.
   (4) Set the row output (`R1,R2,R3,R4`) as `1110`, and read the column input. In this case, we have `C1,C2,C3 = 101`. Because `C2` is read as zero, the pressed key is in row `R4`.
4. After identifying that the pressed key is located in column `C2` and row `R4`, we can look up the pre-defined mapping table of the matrix keypad to find that key "`0`" has been pressed.

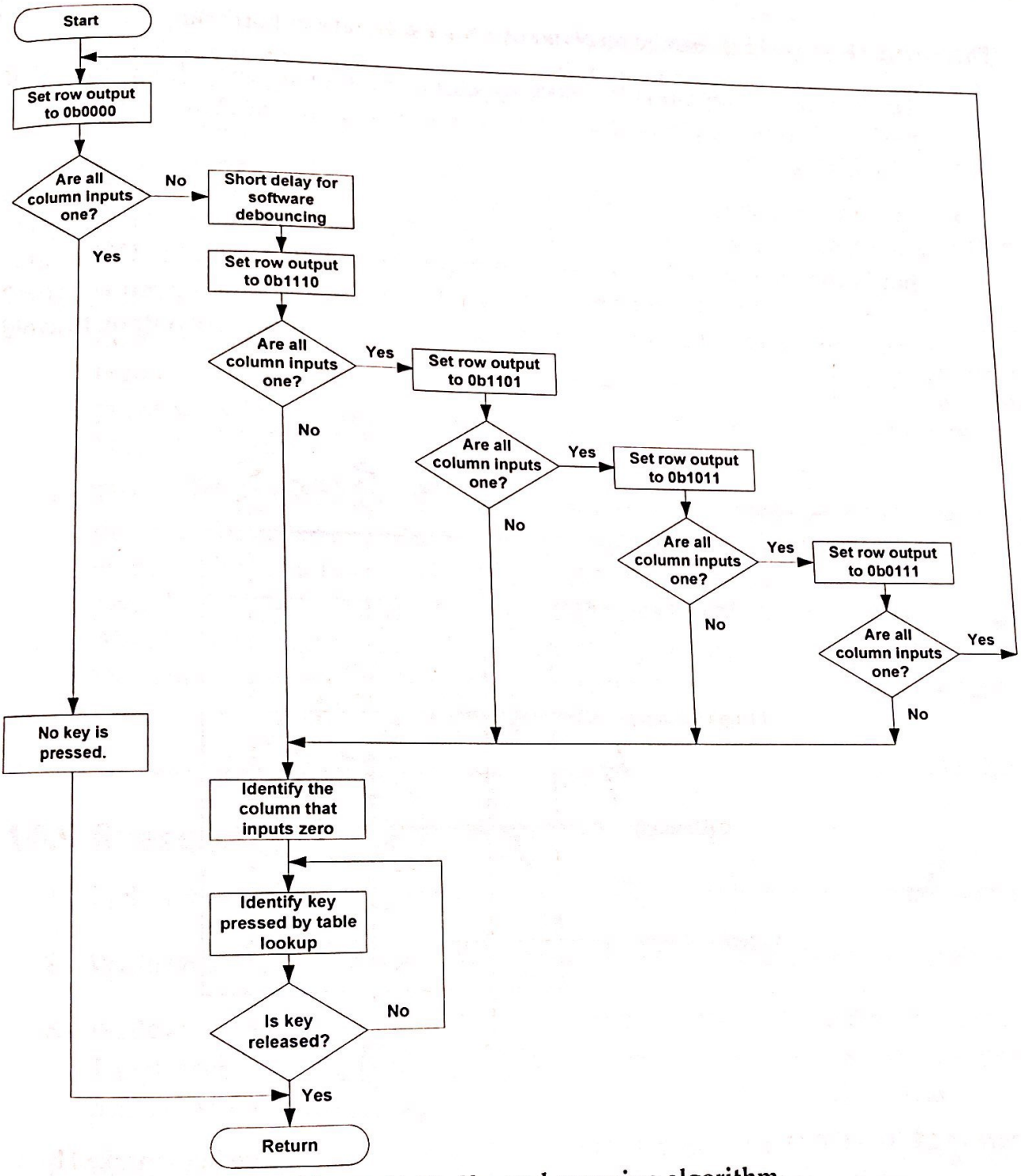

Figure 14-22. Keypad scanning algorithm.
All rows are set as output, and all columns are set as inputs.

This approach is simple, but not safe. During the second step, two row-pins are shorted if multiple keys in the same column are pressed simultaneously. Specifically, the row pins that outputs `1` (*i.e.* 3 V) is directly connected to the row pins that output `0` (*i.e.* `0` V), thus potentially damaging the microcontroller. Figure 14-23 gives one example in which R2 and R3 cause a short circuit.

This circuit shortage issue can be resolved by either software or hardware.

- The hardware solution is to configure all output pins as open-drain, instead of push-pull. When a pin outputs one, the pin is then in HiZ state, and no circuit short can occur.
- The software solution is to switch the row pin from output to input when the rows are scanned. Specifically, when the output of a row pin is set to zero, software should change the mode of the other row pins from GPIO output to GPIO input. For example, when the third row is tested during the row locating process, row 3 is set to output zero, but row 1, 2, and 4 are set as input, instead of output, to avoid a circuit short.

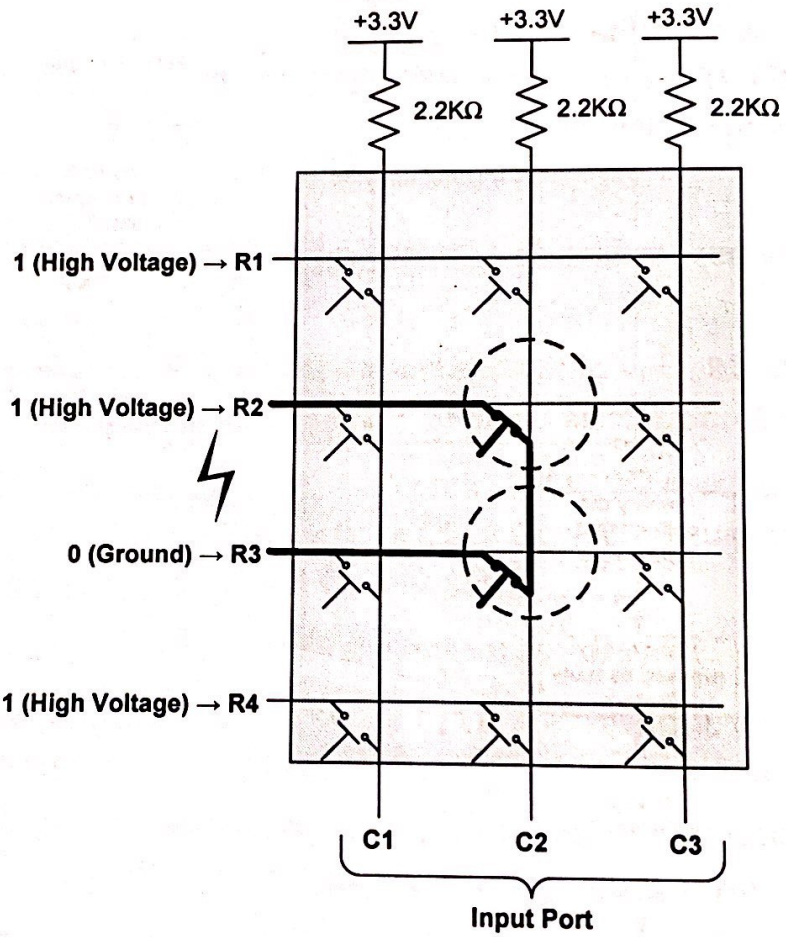

Figure 14-23. The GPIO pins connected to R2 and R3 are shorted if two keys marked by a circle are pressed simultaneously and the GPIO output is 1101.

Another method to avoid damage when multiple keys are pressed is to use reverse scanning algorithm. This method changes the mode of the row port and the column port alternatively to GPIO input and GPIO output to detect the row and the column of a pressed key. This method requires both the row pins and the column pins to be pulled up by some small resistors. This method involves two steps described below.

- During the first step, similar to the scanning algorithm described previously, it sets the row port as output and the column port as input and then reads the column input to identify the column.
- During the second step, it reverses the direction, sets the row port as input and the column port as output, and reads the row input to identify the row.

How does the microcontroller know when a keypad is pressed? There are two methods: polling or interrupt.

- The polling method scans the keypad periodically with a small time interval. This method is simple, but causes a waste of time of microcontrollers. Additionally, because the microcontroller usually has multiple tasks, other tasks may potentially delay the scanning process, so the system is not responsive when a keypad is pressed.
- The interrupt method generates a signal to the processor when the keypad is pressed. This interrupt informs the processor to stop the current tasks and start to execute the scanning code. This method saves the microcontroller from periodically executing the scanning algorithm, thus saving the processor time. Also, the interrupt reduces the latency in responding when a keypad is pressed. However, the interrupt program is more complex to write and debug than polling.

14.9 Exercises

1. Write an assembly program that toggles an LED when the push button is pressed.
2. Write an assembly program that blinks an LED with a time interval of one second.
3. Write an assembly program that scans the keypad to verify a four-digit password. The password is set as 1234. If the user enters the correct password, the program turns the blue LED on. Otherwise, the program turns the red LED on.
4. Write an assembly program to blink an LED to send out an SOS Morse code.
   - Blinking Morse code SOS (· · · — — — · · ·) DOT, DOT, DOT, DASH, DASH, DASH, DOT, DOT, DOT.
   - DOT is on for ¼ second and DASH is on for ½ second, with ¼ second between them.
   - At the end of SOS, the program has a delay of 2 seconds before repeating.

5. Write an assembly program to implement software debouncing for push buttons.

6. Use the logic analyzer to measure the time latency between pressing a button and lighting up an LED.

7. In STM Cortex processors, each GPIO port has one 32-bit set/reset register (`GPIO_BSRR`). We also view it as two 16-bit fields (`GPIO_BSRRL` and `GPIO_BSRRH`) as shown in Figure 14-13. When an assembly program sends a digital output to a GPIO pin, the program has to perform a load-modify-store sequence to modify the output data register (`GPIO_ODR`). The `BSRR` register aims to speed up the GPIO output by removing the load and modify operations.

   - When writing `1` to bit `BSRRH`(i), `ODR`(i) is automatically set to `1`. Writing `0` to any bit of `BSRRH` has no effect on the corresponding `ODR` bit.
   - When writing `1` to bit `BSRRL`(i), `ODR`(i) is automatically set to `0`. Writing `0` to any bit of `BSRRL` has no effect on the corresponding `ODR` bit.

   Therefore, we can change `ODR`(i) by directly writing `1` to `BSRRH`(i) or `BSRRL`(i) without reading the `ODR` and `BSRR` registers. This set and clear mechanism not only improves the performance, but also provides atomic updates to GPIO outputs.

   Write an assembly program that uses the `BSRR` register to toggle the LED.