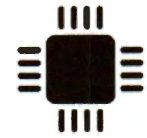
CHAPTER 14

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. Software must 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 ***alternate 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 pin *i* is set as a ***digital input***, the binary data read from this pin of this GPIO group is saved at bit *i* in the input data register (`IDR`). Each bit in `IDR` holds the digital input of the corresponding pin.
- When a GPIO pin *i* is configured as a ***digital output***, bit *i* in the output data register (`ODR`) holds the output of this pin. 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 in `ODR`. Chapter 4.6 introduces how to test, clear, set, and toggle a specific bit of a register in C and assembly.
- All GPIO pins in a GPIO port can be configured as input or output independently.

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.

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 low.

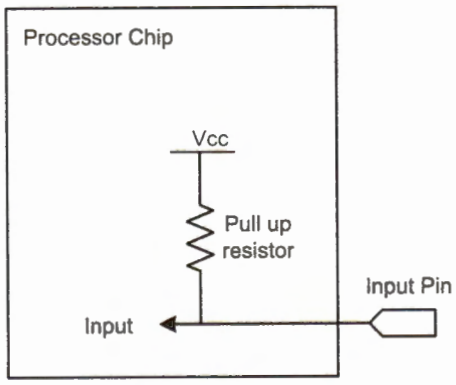


Figure 14-1. The GPIO pin is pulled up internally.

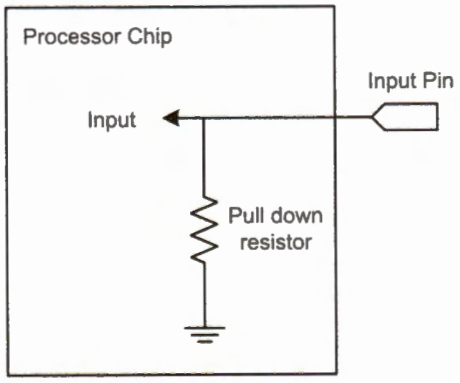


Figure 14-2. The GPIO pin is pulled down internally.

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 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 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 a 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. 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.

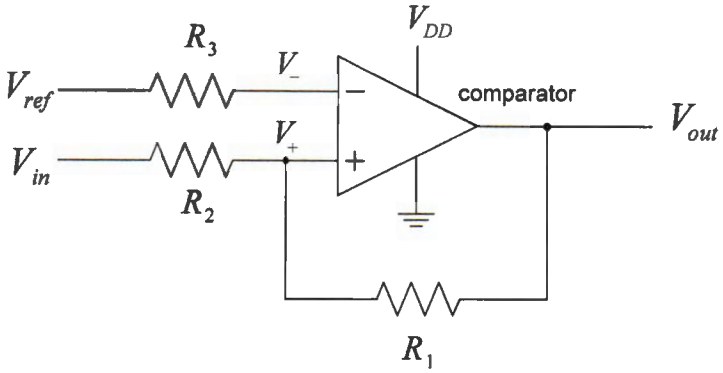


Figure 14-3. Circuit of non-inverting Schmitt trigger with hysteresis. Non-inverting means V_{in} is connected to the non-inverting terminal (*i.e.*, the plus input lead).

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:

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

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.

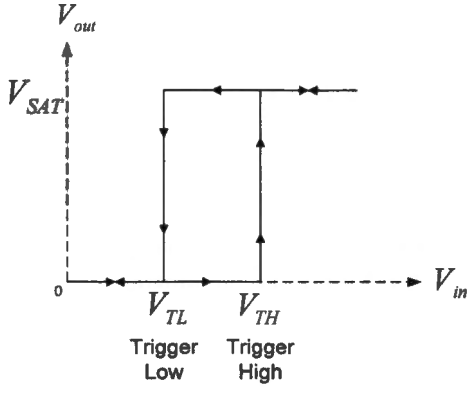


Figure 14-4. Relationship between V_{out} and V_{in} of inverting Schmitt trigger with reference voltage. V_{TL} and V_{TH} are the low and high switching thresholds.

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.

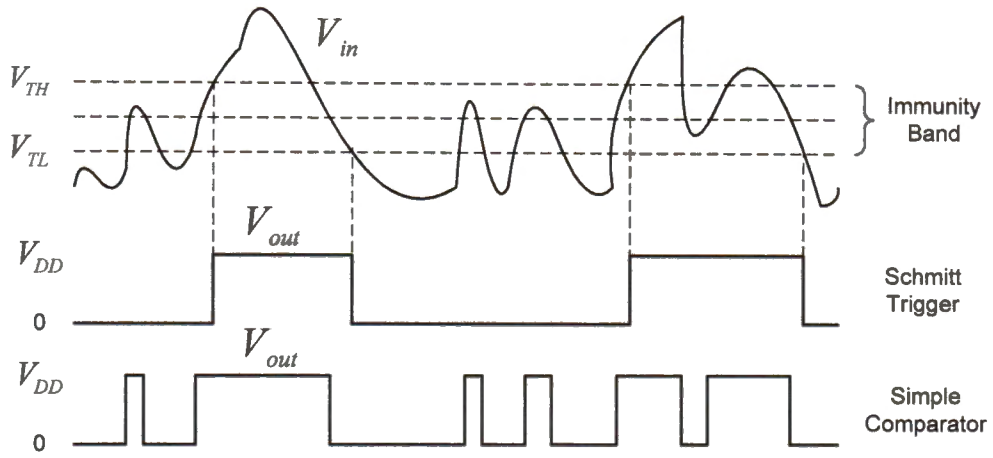


Figure 14-5. Comparing the voltage output of Schmitt trigger with a simple comparator. Schmitt trigger converts an irregular-shaped signal V_{in} into a square wave V_{out} based on two switching thresholds. The simple comparator uses a single threshold (the dotted line between V_{TL} and V_{TH}) to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

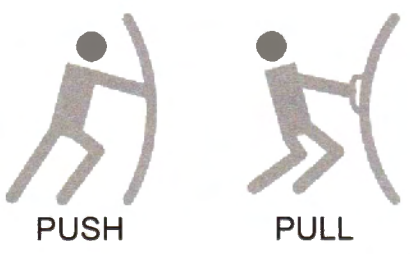
14.4 GPIO Output Modes: Push-Pull and Open-Drain

Software can configure a GPIO output pin as either push-pull or open-drain. Push-pull mode allows the pin to supply and absorb current. However, a GPIO pin in 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.

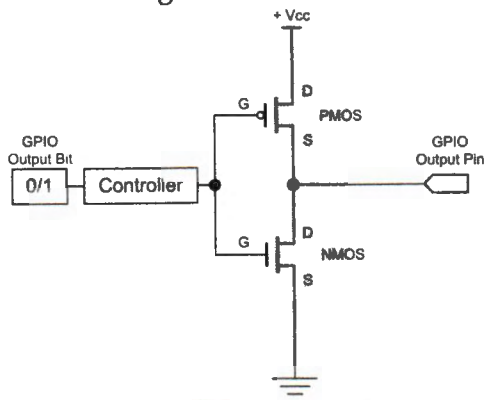


Figure 14-6. A push-pull GPIO digital output

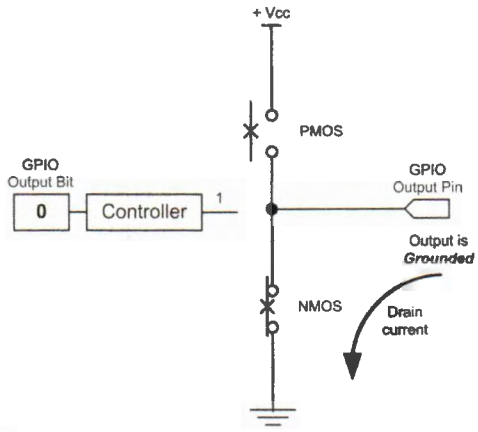


Figure 14-7. If the digital output is 0, then the GPIO output pin is pulled down to the ground in a push-pull setting.

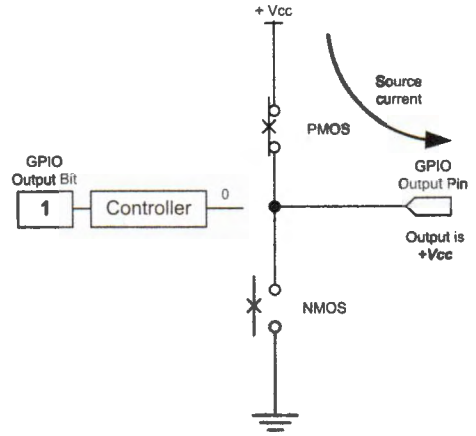
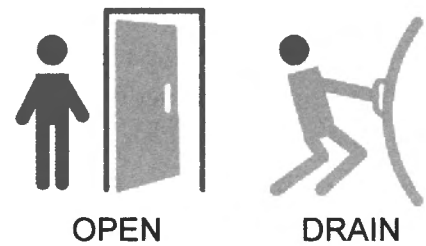


Figure 14-8. If the digital output is 1, then the GPIO output pin is pulled up to the Vcc in a push-pull setting.

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.

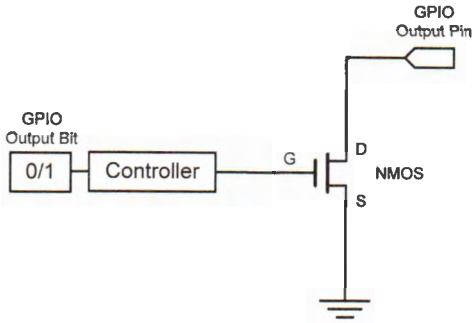


Figure 14-9. An open-drain GPIO digital output

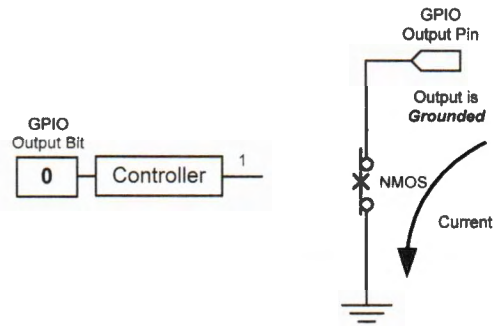


Figure 14-10. If the digital output is 0, then the output pin is pushed to the ground in an open-drain setting (the scenario of *drain*).

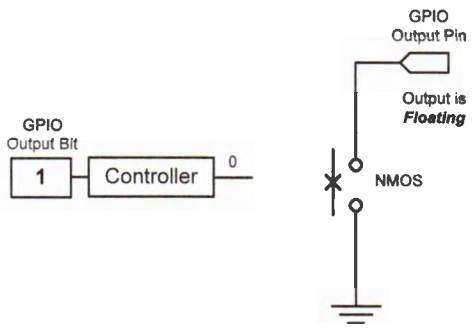


Figure 14-11. If the digital output is 1, then the output pin is floating in an open-drain setting (the scenario of *open*).

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 an easy 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 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.

Figure 14-12 shows the implementation of wired-AND by using open drain and external pull-up when active high logic is used. The output C is determined by the following table.

Inputs				Output
Logic A	Logic B	Circuit A	Circuit B	
0	0	Drain	Drain	0
0	1	Drain	Open	0
1	0	Open	Drain	0
1	1	Open	Open	1

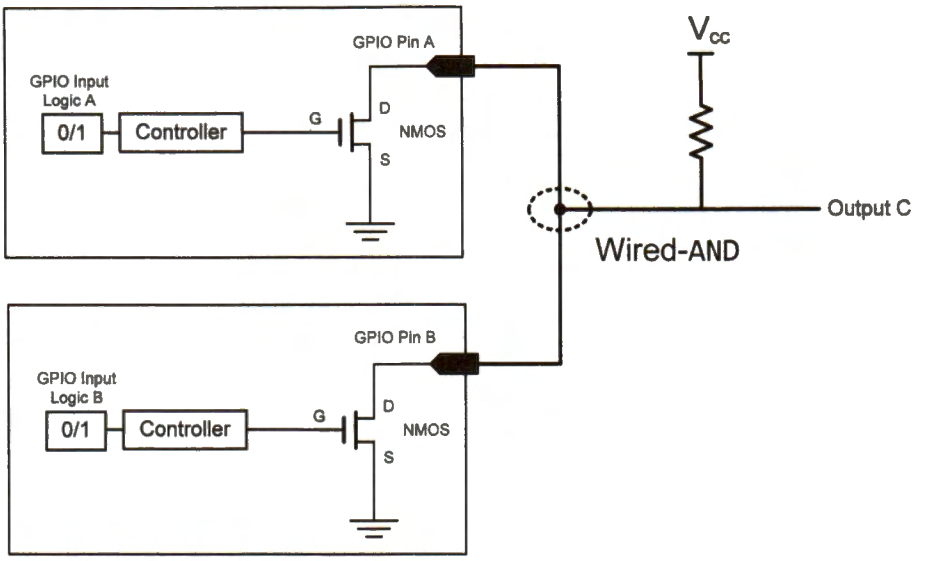


Figure 14-12. Implementation of Wired AND by using open drain and external pull up

Compared to open-drain, 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 speed of change of its output voltage per unit of time, as defined as follows.

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

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 14-13 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.

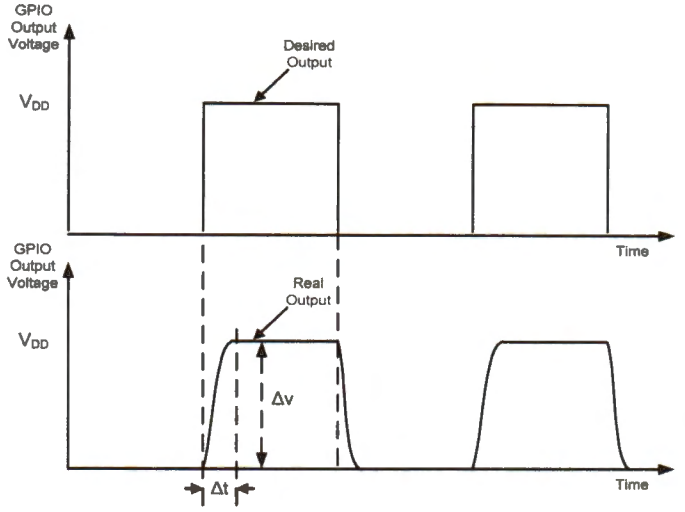


Figure 14-13. Comparing a desired square wave voltage output with the real GPIO output

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 14-13 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 low speed (400 kHz), medium speed (2 MHz), fast speed (10 MHz), or high speed (40 MHz) in the STM32L processors.

14.6 Memory-mapped I/O

Typically, an on-chip peripheral device has a few registers, such as control registers, status registers, data input registers, and data output registers. A peripheral may also have data buffers, such as the display memory of the LCD controller. Input/output or I/O refers to data communication between the processor core and a peripheral device.

There are two complementary approaches to performing I/O operations: port-mapped I/O, and memory-mapped I/O.

- ***Port-mapped I/O*** uses special machine instructions, which are designed specifically for I/O operations. The memory address space and the I/O device address space are independent of each other. Each device is assigned one or more unique port numbers. For example, Intel x86 processors use `IN` and `OUT` instructions to read from or write to a port.
- ***Memory-mapped I/O*** does not need any special instructions. The memory and the I/O devices share the same address space. Each peripheral register or data buffer is assigned to a memory address in the memory address space of the microprocessor. Memory-mapped I/O is performed by the native load and store instructions of the processor. Therefore, memory-mapped I/O is a more convenient way to interface I/O devices. The most significant disadvantage is that memory-mapped I/O has a more complex address decoding unit than port-mapped I/O.

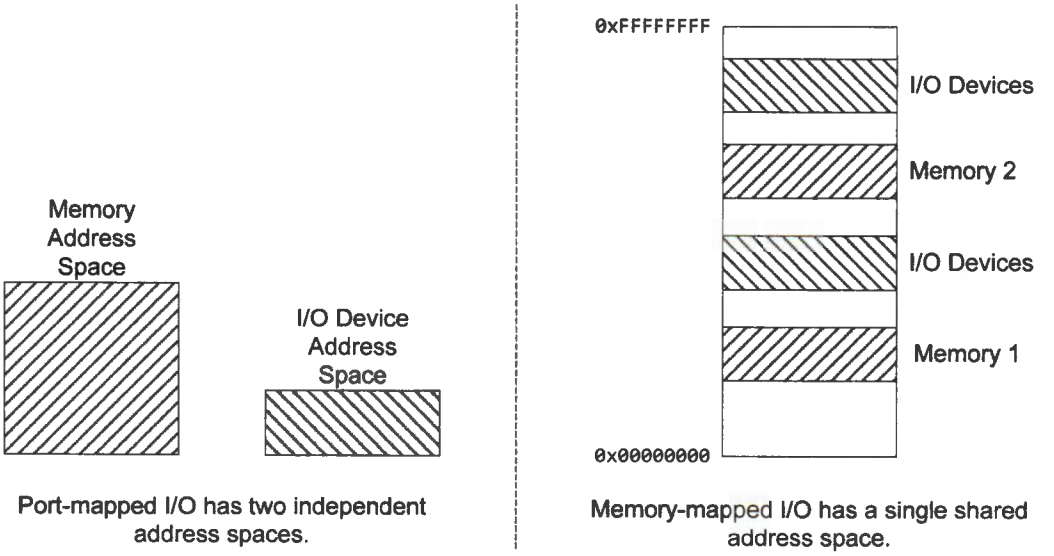


Figure 14-14. Comparison between port-mapped I/O and memory-mapped I/O

ARM Cortex-M processors use memory-mapped I/O to access peripheral registers. All peripheral registers on STM32L4 are mapped to a small memory region starting at `0x40000000`. This region includes the memory addresses of all on-chip peripherals, such as GPIO, timers, UART, SPI, and ADC. The memory address of each peripheral register is determined by chip manufacturers, and usually cannot be changed by software.

A peripheral register usually takes four bytes in memory. For example, the output data register (`ODR`) of Port B on STM32L4 is mapped to memory addresses `0x48000414` to `0x48000417`, with the upper halfword being reserved. Note that values stored in peripheral registers are in the format of little-endian (see Section 5.2).

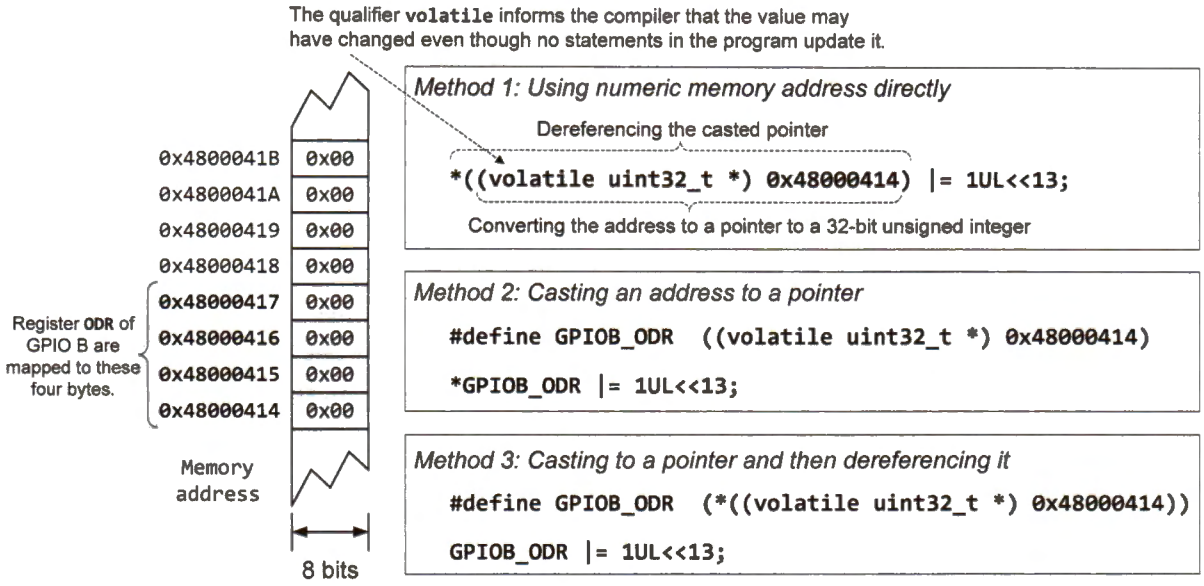


Figure 14-15. Three different approaches to casting a memory address to a pointer

To output high on pin i, $i = 0, 1, 2, \ldots, 15$, software needs to set bit i in `ODR` to 1. For example, to set the output of GPIO pin B 13 to high, software can use the following statement:

```
*((volatile uint32_t *) 0x48000414) |= 1UL<<13;
```

First, this C statement casts the numeric memory address to a pointer, which points to a volatile 32-bit unsigned variable. Note that we put a `volatile` qualifier on each register. When a variable is declared as volatile, the compiler is informed that even though no statements in the program appear to change it, the value might still change. Typically, compilers minimize the number of memory accesses, by temporally storing the memory value in a register, and then repeatedly using it without accessing the memory. The volatile qualifier on a variable prevents the compiler from making such optimization on this variable.

Then, it uses dereferencing to access the value stored in the memory location pointed to by the pointer. However, this C statement is difficult to read and maintain.

Figure 14-15 shows two better approaches (Methods 2 and 3), which use a macro to improve the code's readability. The macro in the second method represents type-casting. The macro in the third method represents both type-casting and dereferencing.

Nevertheless, these approaches are still inconvenient for two reasons.

- First, we must define many macros, one for each peripheral register, even though some peripherals share the same register layout.
- Second, if a function takes a peripheral as input, it is cumbersome to pass all registers of this peripheral as function arguments.

A better approach is to use structures and pointers. Typically, all registers of a peripheral, such as those of GPIO port B (shown in Figure 14-16), are mapped to a contiguous block of physical memory. Therefore, we can define a data structure whose memory layout matches the address assignment given by the chip manufacturer. In C, a `struct` encapsulates related variables into a single structure. All variables in a `struct` are stored contiguously in memory.

```
#define __IO  volatile  // allows read and write

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
  __IO uint32_t BRR;     // Bit reset register
  __IO uint32_t ASCR;    // Analog switch control register
} GPIO_TypeDef

#define GPIOB ((GPIO_TypeDef *) 0x48000400))
```

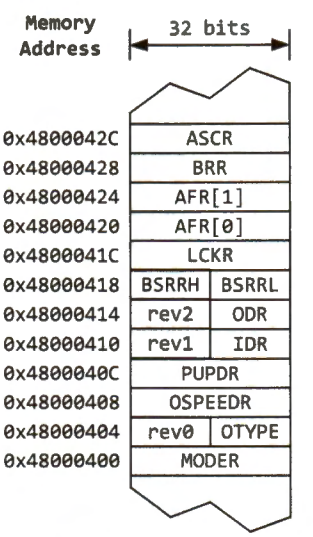


Figure 14-16. Casting the memory address of GPIO B to a GPIO structure pointer.

Let's take GPIO port B as an example. Each GPIO port has a set of registers, such as the mode register (`MODER`), the output data register (`ODR`), and the output speed register (`OSPEEDR`).

- The memory addresses of the registers are defined during the chip design stage. Software cannot change them.

- Each GPIO port has up to 16 pins, and each pin may take 1, 2, or 4 bits in a control register. For example, two bits are required to specify the mode of a GPIO pin. Therefore, the size of these control registers can be either 2, 4, or 8 bytes.
- Because the memory address of each register is word-aligned (*i.e.*, is a multiple of four, see Chapter 10.1.2), dummy bytes are padded in the data structure to correctly map the fixed physical memory layout to the data structure.

To conveniently access a set of registers which are contiguous in memory, we can cast the base memory address of a GPIO port to a pointer to a data structure, as shown below.

```
#define GPIOB ((GPIO_TypeDef *) 0x48000400))
```

In Figure 14-16, six bytes are padded in the `GPIO_TypeDef` structure, making its structure members align properly with their pre-defined memory addresses. Also, in the `GPIO_TypeDef` struct, we put a `volatile` qualifier on each register. This informs the compiler that the variable might change spontaneously by another task thread or by hardware.

If we want to set the output of GPIO port B pin 6 to high, we can use the following C statement. `1UL` is an unsigned long integer with a value of 1. Note the pins are numbered 0 – 15, instead of 1 – 16.

```
  GPIOB->ODR |= 1UL<<6;     // Set bit 6
```

Encapsulating all registers of a peripheral inside a structure provides several advantages.

1. It allows software to access registers in a very convenient way.
2. We can reuse the structure for all peripherals with the same sets of registers. For example, we can reuse the `GPIO_TypeDef` struct for all GPIO ports, as shown below.

```
#define GPIOA ((GPIO_TypeDef *) 0x48000000))
#define GPIOB ((GPIO_TypeDef *) 0x48000400))
#define GPIOC ((GPIO_TypeDef *) 0x40000800))
```

3. We can pass all registers of a peripheral to a function via a single `struct` pointer.

```
void GPIO_Init (GPIO_TypeDef * GPIO);
void main(void){
  GPIO_Init(GPIOA);
  GPIO_Init(GPIOB);
  ...
}
```

14.7 Lighting up an LED

The following shows the basic procedure for lighting up an LED. The software initialization involves two key steps. First, it enables the clock of the GPIO port B via the RCC module. Second, it configures pin 2 of GPIO port B as a general-purpose output pin, with the output type as push-pull. To light up the red LED, we need to output logic "1" to pin 2.

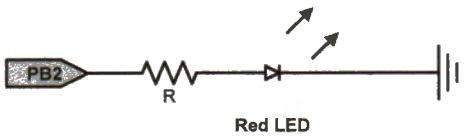


Figure 14-17. Connection diagram between a processor pin and LED

In assembly, a load-modify-store sequence is required to change the register value stored in memory. Also, we can use "EQU" directive to create symbols for the GPIO B base address and ODR register offset, which make the assembly program more readable and self-documenting. The following is an example.

```
GPIOB_BASE EQU 0x48000400     ; Base memory address
GPIO_ODR   EQU 20             ; Byte offset of ODR from the base

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

Additionally, we also need to enable the clock of GPIO port B. To save energy, every peripheral's clock 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 AHB1ENR;    // AHB1 peripheral clocks enable register
  __IO uint32_t AHB2ENR;    // AHB2 peripheral clocks enable register
  __IO uint32_t AHB3ENR;    // AHB3 peripheral clocks enable register
  ...
} RCC_TypeDef;

#define RCC ((RCC_TypeDef *) 0x40021000))
```

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

```
#define RCC_AHB2ENR_GPIOBEN  (0x00000002)
RCC->AHB2ENR |= RCC_AHB2ENR_GPIOBEN;
```

Figure 14-18 shows the flowchart of initializing a GPIO pin as digital output with push-pull.

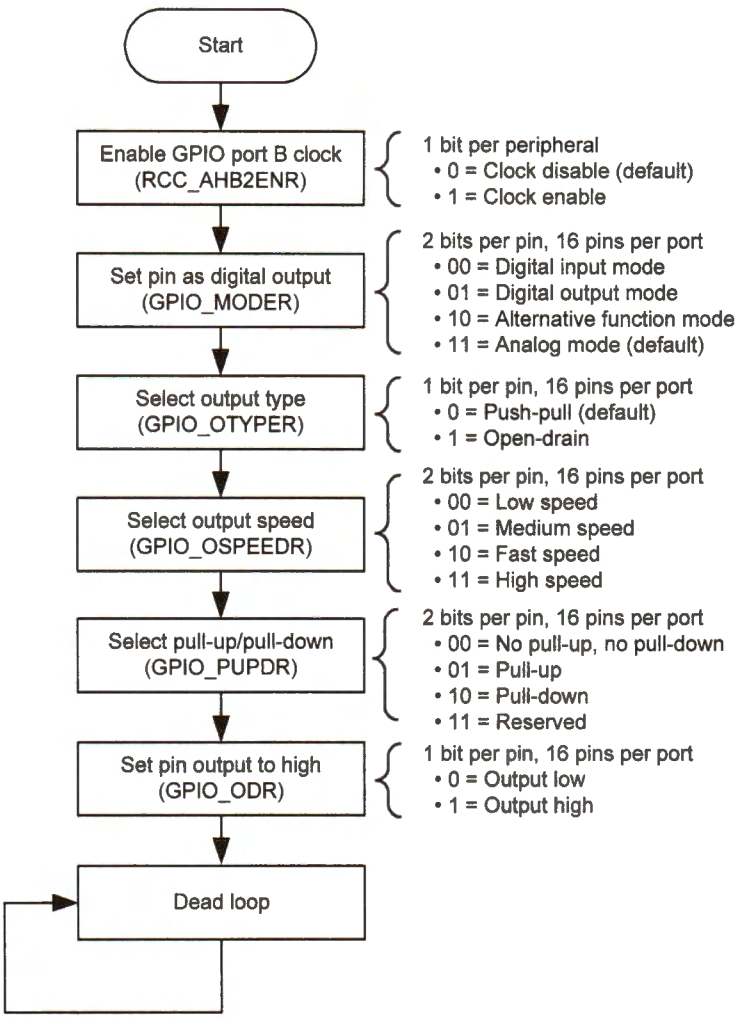


Figure 14-18. Flowchart of GPIO initialization

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 2 to drive a red 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 IDR register records the input of all pins of a GPIO port.

```
// Red LED is connected PB 2 (GPIO port B pin 2)
void GPIO_Clock_Enable(){
   // Enable the clock to GPIO port B
   RCC->AHB2ENR |= RCC_AHB2ENR_GPIOBEN;
}

void GPIO_Pin_Init(){
   // Set mode of pin 2 as digital output
   // 00 = digital input,          01 = digital output
   // 10 = alternate function,     11 = analog (default)
   GPIOB->MODER &= ~(3UL<<4);    // Clear mode bits
   GPIOB->MODER |= 1UL<<4;       // mode = 01, digital output

   // Set output type of pin 2 as push-pull
   // 0 = push-pull (default)
   // 1 = open-drain
   GPIOB->OTYPER &= ~(1<<2);

   // Set output speed of pin 2 as low
   // 00 = Low speed,      01 = Medium speed
   // 10 = Fast speed,     11 = High speed
   GPIOB->OSPEEDR &= ~(3UL<<4);       // Clear speed bits

   // Set pin 2 as no pull-up, no pull-down
   // 00 = no pull-up, no pull-down  01 = pull-up
   // 10 = pull-down,                11 = reserved
   GPIOB->PUPDR &= ~(3UL<<4);         // no pull-up, no pull-down
}
```

```
int main(void){
   GPIO_Clock_Enable();
   GPIO_Pin_Init();
   GPIOB->ODR |= 1UL<<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 like the above C program. In the program,

- `GPIOB_BASE` and `RCC_BASE` are pre-defined memory addresses
- `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.

```
; Constants defined in file stm32l476xx_constants.s
;
; 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   0x48000400
RCC_BASE            EQU   0x40021000

; Byte offset of each variable in 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 AHB2ENR in the RCC_TypeDef structure
RCC_AHB2ENR         EQU   0x4C
```

The following shows the assembly program that sets pin B.2 output to high.

```
  INCLUDE stm32l476xx_constants.s

  AREA    main, CODE, READONLY
  EXPORT      __main              ; make __main visible to linker
  ENTRY

__main PROC
  ; Enable the clock to GPIO port B
  ; Load address of reset and clock control (RCC)
  LDR r2, =RCC_BASE               ; Pseduo instruction
  LDR r1, [r2, #RCC_AHB2ENR]      ; r1 = RCC->AHB2ENR
  ORR r1, r1, #2                  ; Set bit 2 of AHB2ENR
  STR r1, [r2, #RCC_AHB2ENR]      ; GPIO port B clock enable

  ; Load GPIO port B base address
  LDR r3, =GPIOB_BASE             ; Pseudo instruction

  ; Set pin 2 I/O mode as general-purpose output
  LDR r1, [r3, #GPIO_MODER]       ; Read the mode register
  BIC r1, r1, #(3 << 4)           ; Direction mask pin 6, clear bits 5 and 4
  ORR r1, r1, #(1 << 4)           ; Set mode as digital output (mode = 01)
  STR r1, [r3, #GPIO_MODER]       ; Save to the mode register

  ; Set pin 2 the push-pull mode for the output type
  LDR r1, [r3, #GPIO_OTYPER]      ; Read the output type register
  BIC r1, r1, #(1<<2)             ; Push-pull(0), open-drain (1)
  STR r1, [r3, #GPIO_OTYPER]      ; Save to the output type register

  ; Set I/O output speed value as low
  LDR r1, [r3, #GPIO_OSPEEDR]     ; Read the output speed register
  BIC r1, r1, #(3<<4)             ; Low(00), Medium(01), Fast(01), High(11)
  STR r1, [r3, #GPIO_OSPEEDR]     ; Save to the output speed register

  ; Set I/O as no pull-up, no pull-down
  LDR r1, [r3, #GPIO_PUPDR]       ; r1 = GPIOB->PUPDR
  BIC r1, r1, #(3<<4)             ; No PUPD(00), PU(01), PD(10), Reserved(11)
  STR r1, [r3, #GPIO_PUPDR]       ; Save pull-up and pull-down setting

  ; Light up LED
  LDR r1, [r3, #GPIO_ODR]         ; Read the output data register
  ORR r1, r1, #(1<<2)             ; Set bit 2
  STR r1, [r3, #GPIO_ODR]         ; Save to the output data register

stop
  B  stop  ; dead loop & program hangs here
  ENDP
  END
```

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

14.8 Push Button

When a mechanical 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 14-19 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.

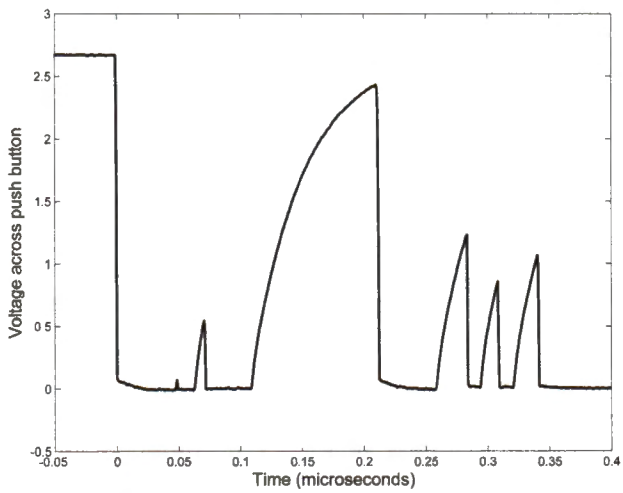


Figure 14-19. The voltage across a push button when there is no hardware debouncing

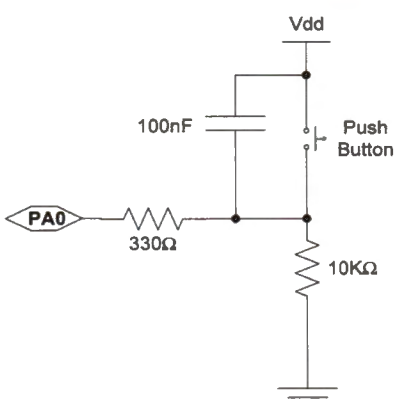


Figure 14-20. A push button with RC debouncer

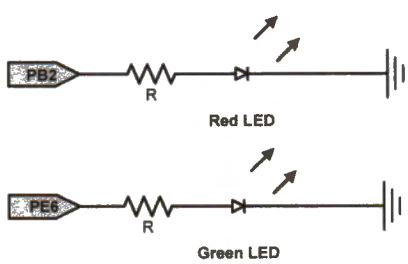


Figure 14-21. Red and green LED

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 14-20. When the switch is open, this capacitor is fully charged. Therefore, there is no current on these resistors, and the voltage on the processor pin is zero. As soon as the button is pressed, the capacitor is quickly discharged. If the button rebounds and the switch is open briefly, the capacitor cannot be recharged fast enough to pull the processor pin low. Figure 14-22 shows the voltage signals when the LED is lit up after the push button is pressed. It indicates that the voltage on the pin (PA `0`) connected to the push button rises smoothly without generating any bouncing signals.

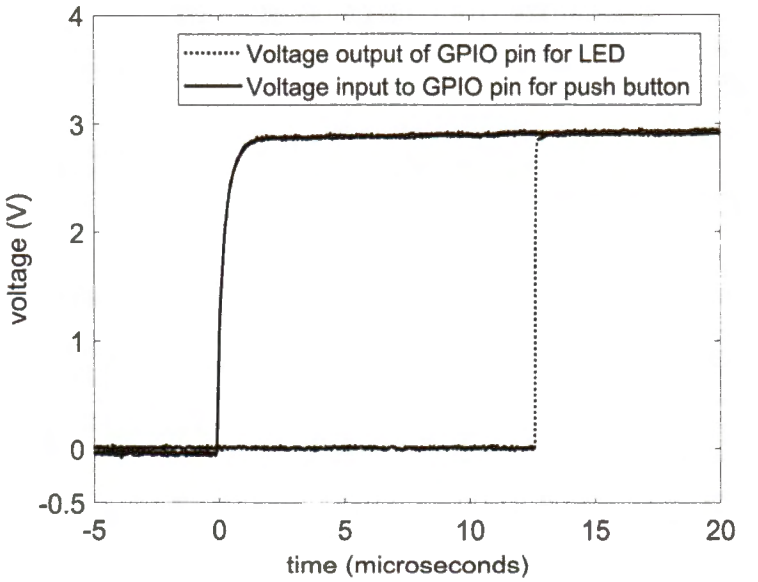


Figure 14-22. Voltage signals on the LED pin when the push button is pressed

The easiest ***software debouncing*** technique is *wait-and-see*, as shown in Example 14-3. When the program detects that a button is pressed, it re-examines the input signal after a short delay, typically between 20 and 50 *ms*. If the input signal still shows the button is pressed, the program then reports that the button has been pressed indeed.

```
bool is_button_pressed(){
   read button input;
   if (button is not pressed) return false;
   wait 50 ms;
   read button input again;
   if (button is not pressed) return false;
   return true;
}
```

Example 14-3. Pseudocode of *wait-and-see* software debouncing

However, the response time of the wait-and-see technique is significant and is not acceptable in many applications, such as gaming or mission-critical systems. A better software debouncing technique is ***counter debouncer***. It polls the button input at regular intervals and requires a few consecutive positive readings to confirm the button has been pressed. In Example 14-4, the pin is polled every 5 *ms* during the debounce period 50 *ms*, and it requires 4 consecutive positive readings. If a button is pressed, this approach has less response time than the wait-and-see method.

```
bool is_button_pressed(){

  read button input;
  if (button is not pressed)
     return false;

  counter = 0;

  for(i = 0; i < 10; i++){
     wait 5 ms;
     read button input;
     if (button is not pressed) {
        counter = 0;            // bounce, reset counter
     } else {
        counter = counter + 1; // stable, increase counter
        if (counter >= 4)      // require 4 consecutive positive readings
           return true;
     }
  }
  return false;
}
```

Example 14-4. Pseudocode of *counter-debouncer* software debouncing

One issue in software debouncing methods is how to implement the time delay. Many new programmers use a large `for` or `while` loop to achieve the delay, as shown below.

```
void wait_ms(uint32_t ms){
   uint32_t i, j;
   for(i = 0; i < ms; i++)
      for(j = 0; j < 255; j++); // adjust 255 to achieve 1 ms delay
}
```

Example 14-5. Loop-based time delay

The loop-based time delay is not recommended for real-time embedded systems. First, these busy loops tie up the processor and prevent other tasks from running, wasting processor time and energy. Second, the timing may change dramatically based on compiler versions, compiler optimization levels, and processor speed.

A better implementation is to use timer interrupts, as shown in Example 14-6.

```
volatile uint8_t counter = 0;
volatile uint8_t pressed = 0;

// Set up timer 4 to generate an interrupt every 5 ms
...

void TIM4_IRQHandler(void) {
   ...
   if((GPIOA->IDR & 0x1) == 0x1){  // check input on pin PA.0
      counter++;                   // button is pressed
      if (counter >= 4) {
         pressed = 1;              // set the flag
         counter = 0;              // reset counter
      } else {                     // button is not pressed
         counter = 0;              // reset counter
      }
   }
}
```

Example 14-6. Using timer interrupts to implement counter debouncer

The following shows a polling I/O method (***busy waiting***) to constantly query 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. A method based on interrupts is more efficient than polling. See Chapter 11.8 external interrupts.

Suppose a push button is connected to the GPIO pin PA `0`, and an LED is attached to the GPIO in PB 2, as shown in Figure 14-20 and Figure 14-21. Because hardware debouncing is used for the push button, software debouncing is not deployed in this example. All inputs of a GPIO port are stored in its input data register (`IDR`). Specifically, the input of pin `0` is saved at bit `0` in `IDR`. A low voltage input yields to a value of `0`, and a high voltage generates a value of `1`. Figure 14-23 gives the program flowchart.

```
// Enable the clock to GPIO port A and B
RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN | RCC_AHB2ENR_GPIOBEN;

// Set mode of pin 0 as general-purpose input
// 00 = Digital input,        01 = Digital output
// 10 = Alternate function,   11 = Analog
GPIOA->MODER &= ~3UL;          // Set mode as input (00)

// Set I/O as no pull-up, no pull-down
// 00 = No pull-up/pull down, 01 = Pull-up
// 10 = Pull-down,            11 = Reserved
GPIOA->PUPDR &= ~3UL;          // Pull-up pull-down mask
```

```
// Set PB.2 as digital output with push-pull, no pull-up/pull-down
// See Example 14-1
...

while (1) {
  // Toggle red LED when button PA.0 is pushed
  if((GPIOA->IDR & 0x1) == 0x1){
     GPIOB->ODR ^= GPIO_ODR_ODR_2;       // Toggle pin PB.2
     while((GPIOA->IDR & 0x1) != 0x00); // Wait until button is released
  }
}
```

Example 14-7. Read pin PA.0, and toggle pin PB.2 if the input on PA.0 is 1. No software debouncing is used because pin PA.0 has been debounced by hardware.

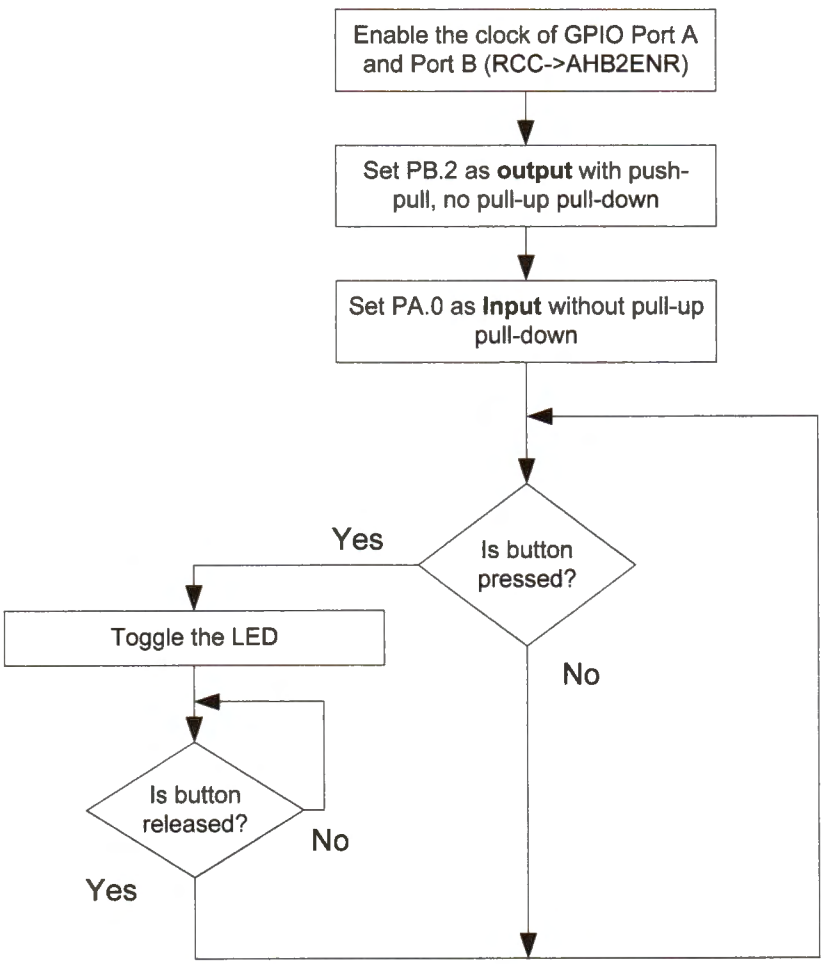


Figure 14-23. Flowchart of programming a pushbutton to control an LED

14.9 Keypad Scan

Suppose we have a keypad that has 12 keys, as shown in Figure 14-24. 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 14-25. 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 14-25, 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.

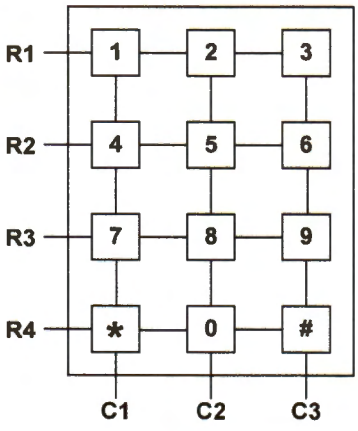


Figure 14-24. 3×4 keypad

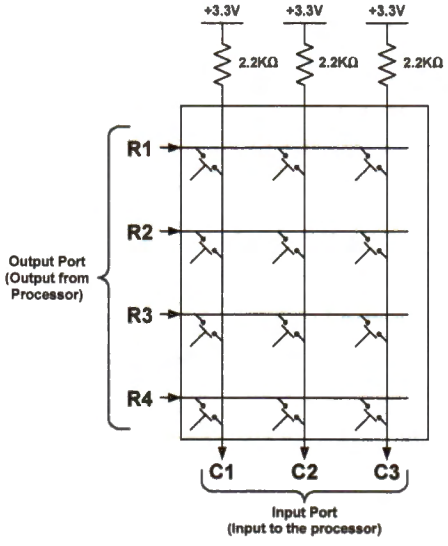


Figure 14-25. Input and output setting

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 pushed down.
2. Identify the row number of the pressed key. Drive the output of the first row 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`). Thus, we have `C1,C2,C3 = 101` and 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 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.

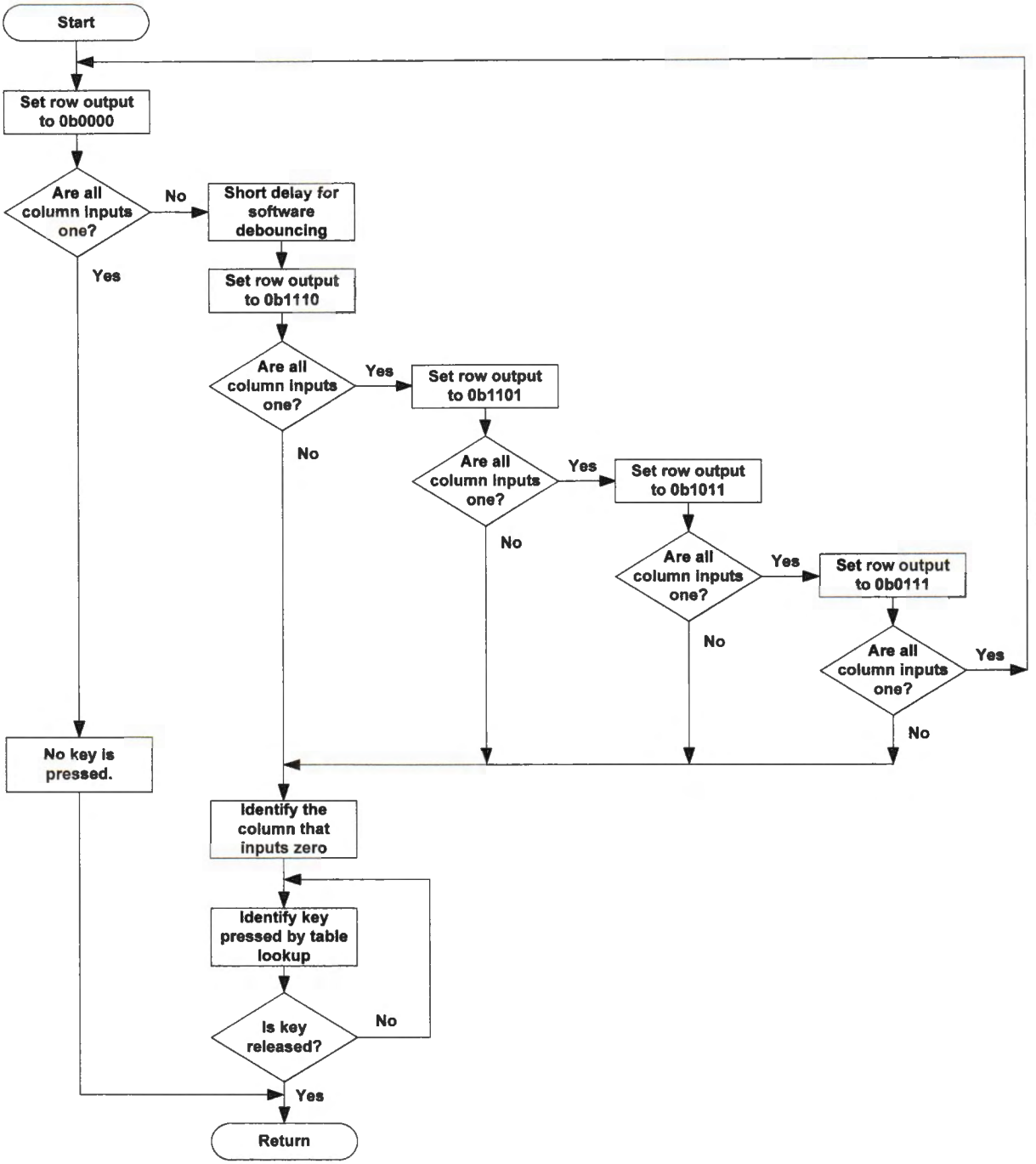


Figure 14-26. Keypad scanning algorithm.
All rows are set as output, and all columns are set as inputs.

Figure 14-26 gives the flowchart of the keypad scanning algorithm introduced previously. In the following, we will show partial implementation scanning algorithm in C.

To facilitate the key lookup, we define a key-map array, which is used to convert the key position (row, column) to its corresponding logic number or letter.

```
unsigned char key_map [4][3] = {
  {'1','2','3'},  // 1st row
  {'4','5','6'},  // 2nd row
  {'7','8','9'},  // 3rd row
  {'*','0','#'},  // 4th row
};
```

The sketch of the scanning subroutine is given below. It identifies the row and column of the key pressed, and returns the ASCII value of the key pressed. When no key has been pressed, it returns `0xFF`.

```
unsigned char keypad_scan(void) {

  unsigned char row, col, ColumnPressed;
  unsigned char key = 0xFF;

  // Check whether any key has been pressed
  //   1. Output zeros on all row pins
  //   2. Delay shortly, and read inputs of column pins
  //   3. If inputs are 1 for all columns, then no key has been pressed
  ...
  if ( ... )      // If no key pressed, return 0xFF
    return 0xFF;

  // Identify the column of the key pressed
  for(col = 0; col < 3; col++) { // Column scan
    if (...)
       ColumnPressed = col;
  }

  // Identify the row of the column pressed
  for(row = 0; row < 4; row++) { // Row scan
    // Set up the row outputs
    ...
    // Read the column inputs after a short delay
    ...
    // Check the column inputs
    if ( ... )  // If the input from the column pin ColumnPressed is zero
       key = key_map[row][ColumnPressed];
    }
  }

  return key;
}
```

The main function repeatedly performs the scan operations, stores keys that have been pressed in a string array, and displays the string array on LCD. The keys '*' and '#' can be used to implement special functions. For example, we can use '*' to delete the last key pressed.

```
int main(void){

   unsigned char key;
   char str[50];
   unsigned char len = 0;

   // GPIO and clock configurations
   // LCD initializations
   // Configure row pins as open-drain to prevent potential
   // circuit shortage (see detailed explanation in the text below)
   ...

   while(1){

      key = keypad_scan();

      switch (key) {

          case '*':         // If * pressed
              ...
              break;
          case '#':         // If # pressed
              ...
              break;
          case 0xFF:        // No key pressed
              ...
              break;
          Default:          // Add key pressed to string
              str[len] = key;
              str[len + 1] = 0;  // NULL string terminator
              len++;
              if (len >= 48) len = 0;  // Avoid buffer overflow
      }

      LCD_DisplayString((uint8_t *)str);
   }
}
```

The algorithm presented in Figure 14-26 has one serious problem: it may cause a short circuit. 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 14-27 gives one example in which row pins R2 and R3 are

connected if two keys marked are pressed simultaneously. When software scans row 2 or row 3, a short circuit is generated, causing potential hardware damage.

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.

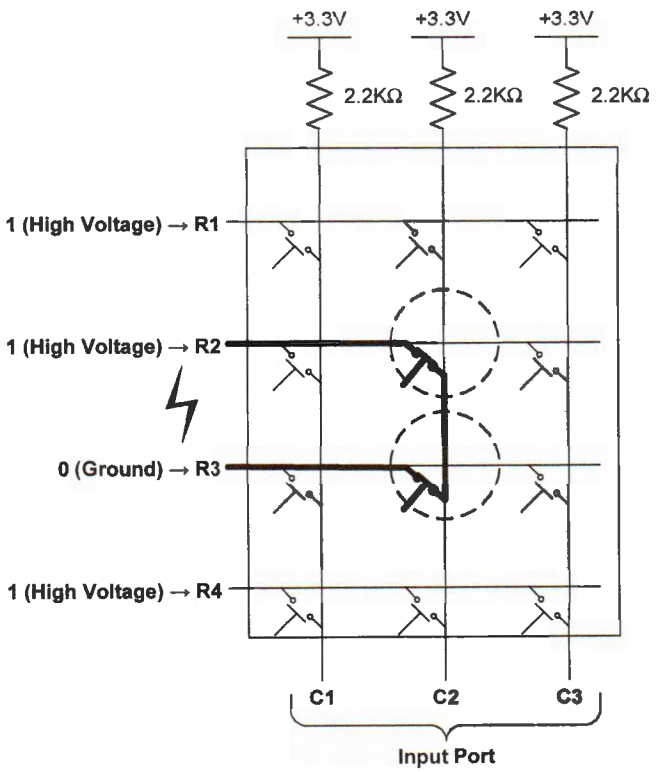


Figure 14-27. The GPIO pins connected to R2 and R3 are shorted if two keys marked by a circle are pressed simultaneously and the GPIO output is 1101.

Another method to avoid damage when multiple keys are pressed is to use reverse scanning algorithm. This approach 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. It involves two steps described below.

- During the first step, like 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.

14.10 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 red 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 14-16. When an assembly program sends a digital output to a GPIO pin, the program should 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), bit `ODR`(i) is automatically set. Writing `0` to any bit of `BSRRH` has no effect on the corresponding `ODR` bit.
   - When writing `1` to bit `BSRRL`(i), bit `ODR`(i) is automatically cleared. 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.