# 通用输入输出端口的基本概念

通用输入输出端口(GPIO)是MCU的最基本外设。

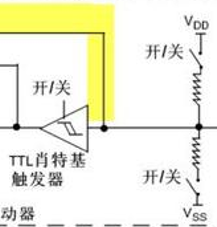
GPIO=General Purpose Input Output，通用输入输出。有时候简称为“IO口”。

**端口（Port）**：I/O端口主要功能是CPU和外设进行通信的媒介。一个端口一般有8个引脚。TM4C1294有15个端口，能控制90个引脚(Port A, Port B, Port C, Port D, Port E, Port F, Port G, Port H, Port J, Port K, Port L, PortM, Port N, Port P, Port Q)

**引脚（管脚,Pin）**：每个引脚都是MCU与外界连接的独立的导线，隶属于某一个端口。对其中一个引脚进行操作的时候不会影响到其他引脚。我们可以对引脚单独设置为数字输入或者数字输出。其中有些引脚还可以用作模拟输入和外设中断。同一时刻只有一个功能映射到同一个引脚。

**输入方式**：

* 浮空输入（上下两个电阻上电开关都打开，输入信号必须有确定的电平，不能悬空）
* 上拉输入（上面的电阻上电开关闭合，悬空时为高电平）
* 下拉输入（下面的电阻上电开关闭合，悬空时为低电平）
* 模拟输入（上下两个电阻上电开关都打开，接模拟信号）



EK-TM4C1294XL开发板开发板的简单按键，必须配合上拉输入才能工作：



**输出方式**：

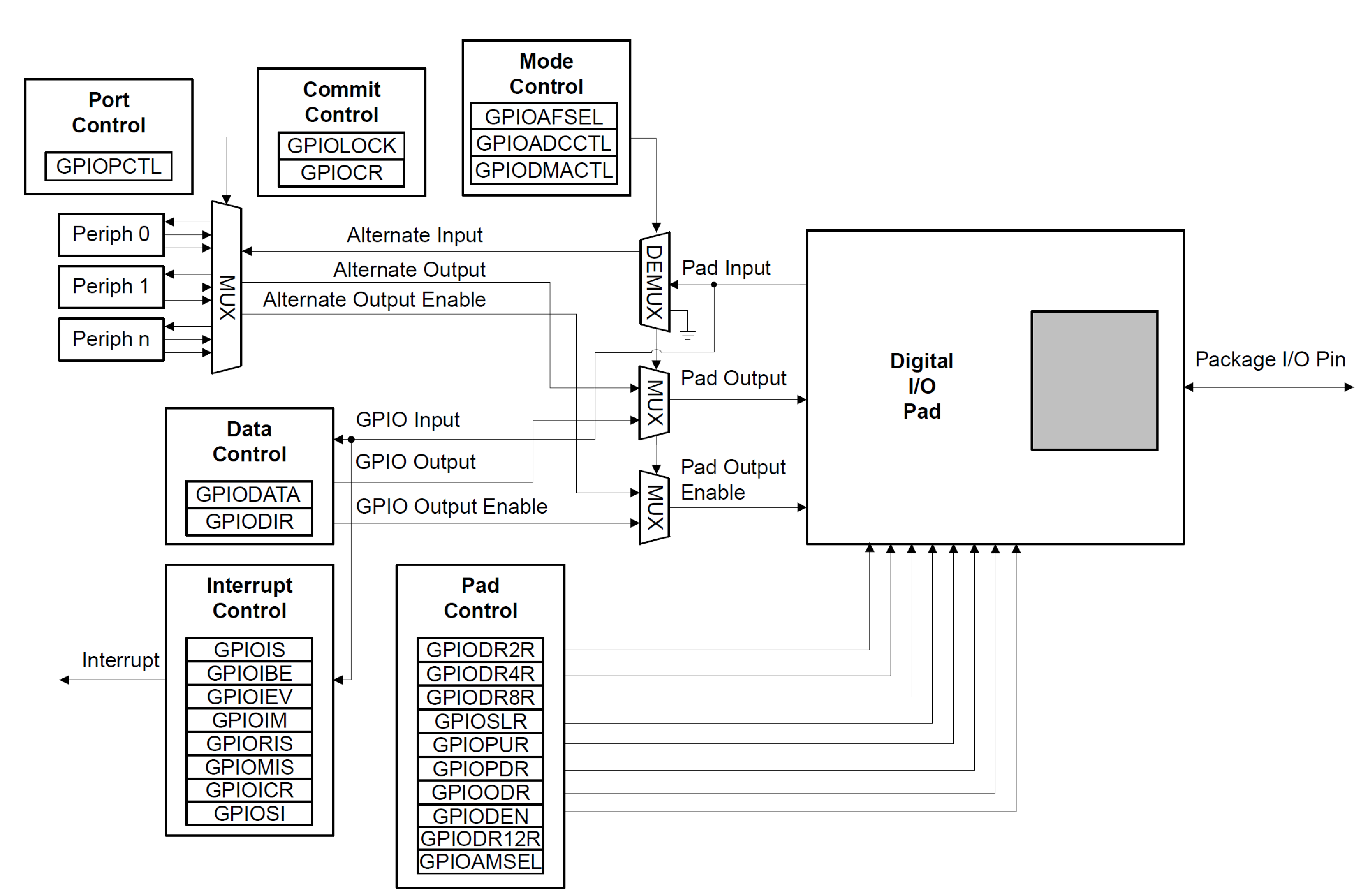
* 推挽输出（高电平VDD，低电平GND）
* 开漏输出（需要上拉电阻才能工作，高电平可以为VDD以下的任意电压，低电平GND，可用于5V系统与3.3V系统连接）

|  |  |
| --- | --- |
|  |  |
| 推挽输出 | 开漏输出 |

# TM4C1294的GPIO

引脚结构：

每个GPIO模块控制8个引脚，其中一个引脚与GPIO模块的逻辑关系为：



有多个控制寄存器来控制引脚的具体功能。这些**外设寄存器**与**CPU的寄存器**不同，**外设寄存器**可以通过系统总线访问，在系统总线上，有确定的地址。

既可以做输入可以做输出，有**GPIODIR**寄存器控制。

**输入**，可以为数字输入，也可以为模拟输入。如果**GPIODIR**寄存器将引脚配置为输入，输入的电平被存储到GPIODATA寄存器中，读取GPIODATA寄存器，就可以知道引脚上的电平状态。

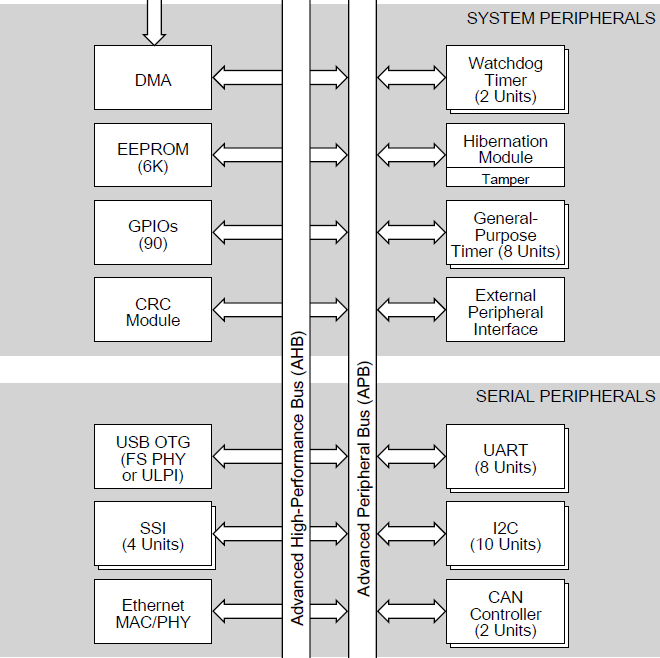
**输出**，带**使能**控制。如果**GPIODIR**寄存器将引脚配置为输出，GPIODATA寄存器可以控制输出的电平。

# 控制寄存器的访问

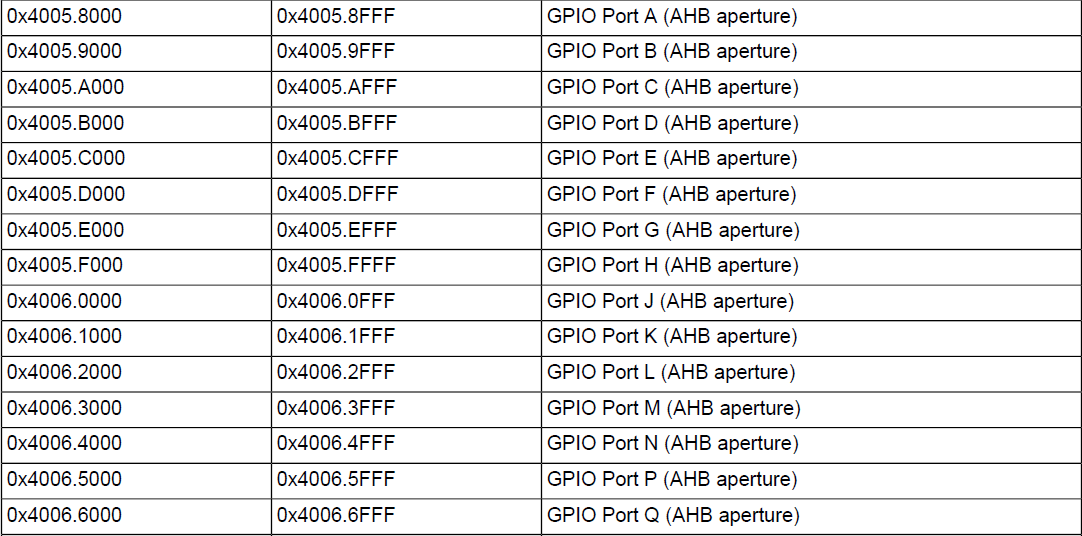
端口的控制寄存器组的地址，由基地址和偏移地址组成。控制寄存器在总线上的实际地址为：基地址+偏移地址。

每个端口（PORT A … PORT Q）寄存器组的基地址不同，但是相同功能的寄存器的偏移地址是相同的。

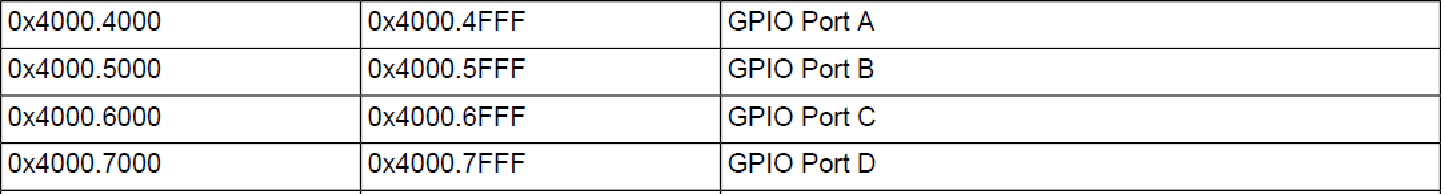
有两条系统总线可以与外设通信，分别是APB和AHB（AHB比APB快）：

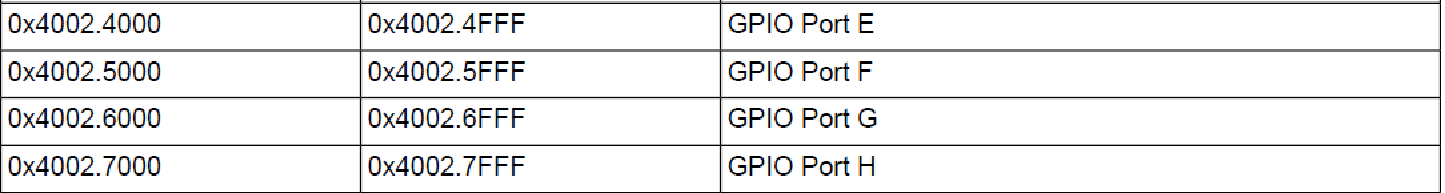


各个端口寄存器组在AHB上的基地址：



各个端口寄存器组在APB上的基地址：







虽然地址有所不同，但功能是一致的。推荐使用AHB的地址，因为访问更快，如果使用了APB的地址，效果也是一样的。

在头文件hw\_memmap.h中，使用宏定义，定义了端口基地址：

AHB地址：

**#define** GPIO\_PORTA\_AHB\_BASE 0x40058000 // GPIO Port A (high speed)

**#define** GPIO\_PORTB\_AHB\_BASE 0x40059000 // GPIO Port B (high speed)

**#define** GPIO\_PORTC\_AHB\_BASE 0x4005A000 // GPIO Port C (high speed)

**#define** GPIO\_PORTD\_AHB\_BASE 0x4005B000 // GPIO Port D (high speed)

**#define** GPIO\_PORTE\_AHB\_BASE 0x4005C000 // GPIO Port E (high speed)

**#define** GPIO\_PORTF\_AHB\_BASE 0x4005D000 // GPIO Port F (high speed)

**#define** GPIO\_PORTG\_AHB\_BASE 0x4005E000 // GPIO Port G (high speed)

**#define** GPIO\_PORTH\_AHB\_BASE 0x4005F000 // GPIO Port H (high speed)

**#define** GPIO\_PORTJ\_AHB\_BASE 0x40060000 // GPIO Port J (high speed)

**#define** GPIO\_PORTK\_BASE 0x40061000 // GPIO Port K

**#define** GPIO\_PORTL\_BASE 0x40062000 // GPIO Port L

**#define** GPIO\_PORTM\_BASE 0x40063000 // GPIO Port M

**#define** GPIO\_PORTN\_BASE 0x40064000 // GPIO Port N

**#define** GPIO\_PORTP\_BASE 0x40065000 // GPIO Port P

**#define** GPIO\_PORTQ\_BASE 0x40066000 // GPIO Port Q

APB地址：

**#define** GPIO\_PORTA\_BASE 0x40004000 // GPIO Port A

**#define** GPIO\_PORTB\_BASE 0x40005000 // GPIO Port B

**#define** GPIO\_PORTC\_BASE 0x40006000 // GPIO Port C

**#define** GPIO\_PORTD\_BASE 0x40007000 // GPIO Port D

**#define** GPIO\_PORTE\_BASE 0x40024000 // GPIO Port E

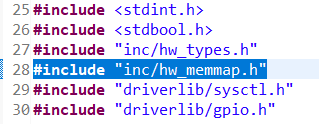
**#define** GPIO\_PORTF\_BASE 0x40025000 // GPIO Port F

**#define** GPIO\_PORTG\_BASE 0x40026000 // GPIO Port G

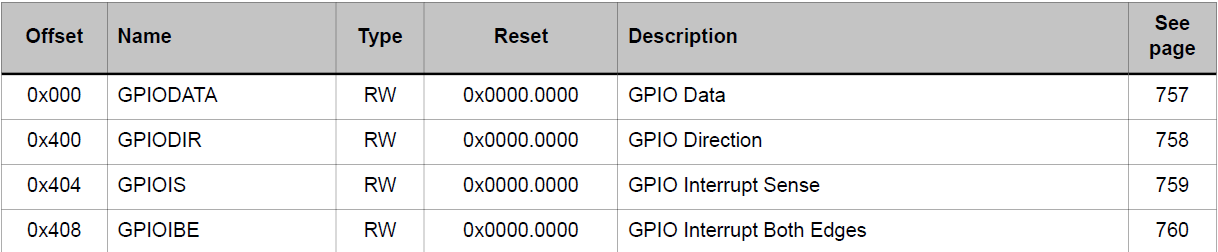
**#define** GPIO\_PORTH\_BASE 0x40027000 // GPIO Port H

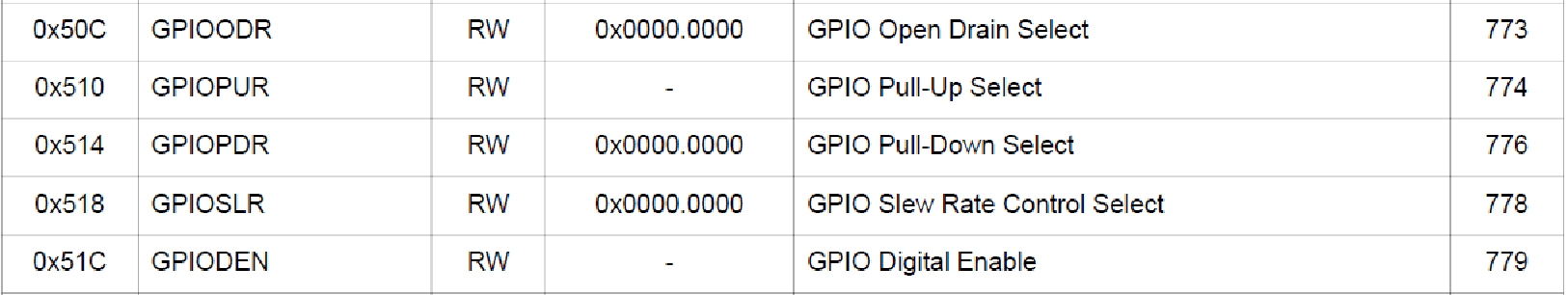
**#define** GPIO\_PORTJ\_BASE 0x4003D000 // GPIO Port J

所以在编程时，需要首先包含头文件**#include** "inc/hw\_memmap.h"



各个端口配置寄存器组的基地址不同，但是负责控制具体功能的偏移地址是相同的：





偏移地址在inc/hw\_gpio.h进行了定义，如果用到的话，需要包含该头文件 **#include** "inc/hw\_gpio.h"：

**#define** GPIO\_O\_DATA 0x00000000 // GPIO Data

**#define** GPIO\_O\_DIR 0x00000400 // GPIO Direction

**#define** GPIO\_O\_IS 0x00000404 // GPIO Interrupt Sense

**#define** GPIO\_O\_IBE 0x00000408 // GPIO Interrupt Both Edges

**#define** GPIO\_O\_IEV 0x0000040C // GPIO Interrupt Event

**#define** GPIO\_O\_IM 0x00000410 // GPIO Interrupt Mask

**#define** GPIO\_O\_PUR 0x00000510 // GPIO Pull-Up Select

**#define** GPIO\_O\_PDR 0x00000514 // GPIO Pull-Down Select

**#define** GPIO\_O\_SLR 0x00000518 // GPIO Slew Rate Control Select

**#define** GPIO\_O\_DEN 0x0000051C // GPIO Digital Enable

某个端口控制寄存器在总线上的实际地址为：基地址+偏移地址。

如，要控制PORTn端口的GPIODIR寄存器，需要调用HWREG(GPIO\_PORTN\_BASE + GPIO\_O\_DIR)

其中HWREG的定义如下：

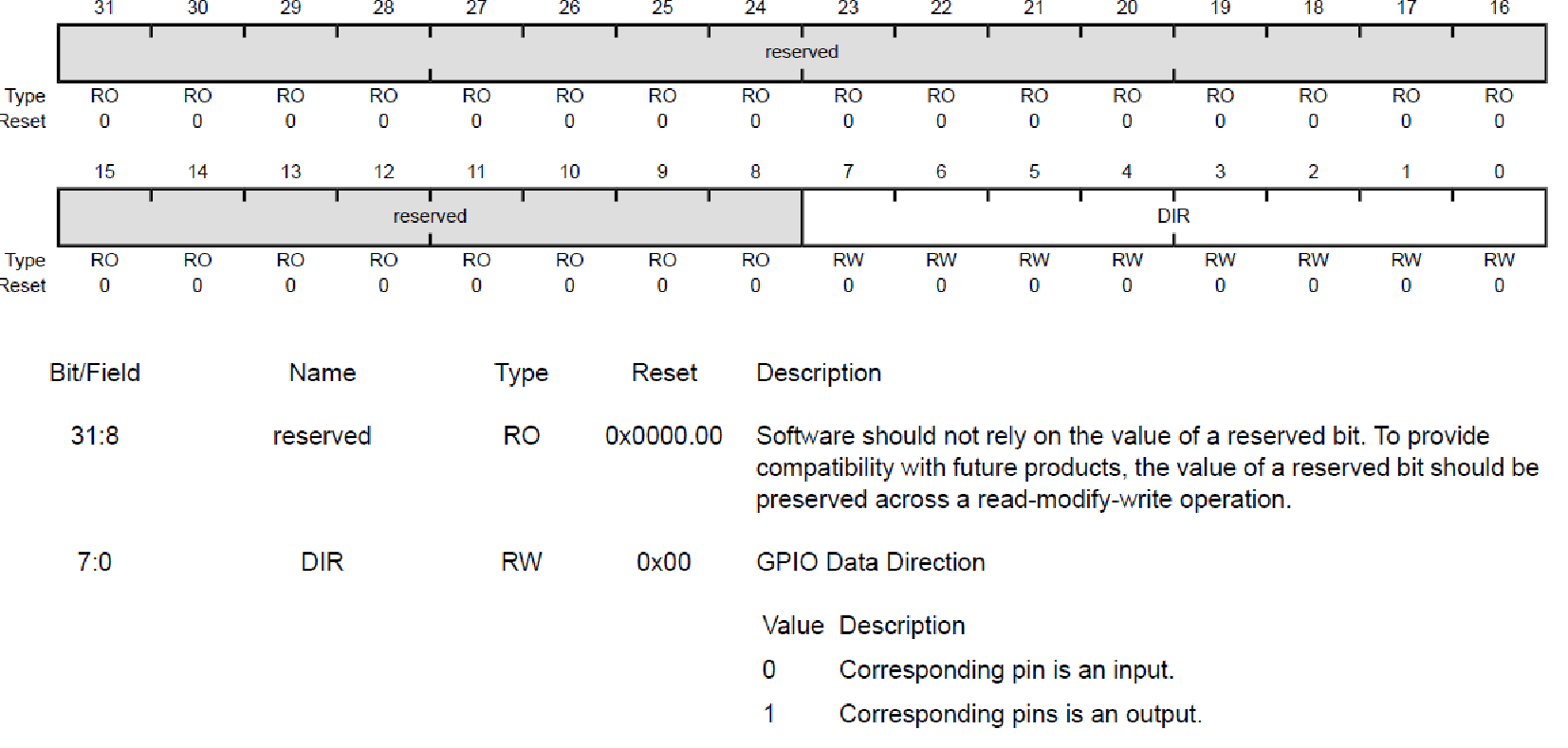
**#define** HWREG(x) \

(\*((**volatile** uint32\_t \*)(x)))

# 控制寄存器的功能

只有低八位有实际作用，用于控制该端口的八个引脚的方向，每一位控制一个引脚。

0为输入，1为输出



如果要将PORTn的Pin0设置为输出：

HWREG(GPIO\_PORTN\_BASE + GPIO\_O\_DIR)=0x01;

如果要将PORTn的Pin1设置为输出：

HWREG(GPIO\_PORTN\_BASE + GPIO\_O\_DIR)=0x02;

如果要将PORTn的Pin2设置为输出：

HWREG(GPIO\_PORTN\_BASE + GPIO\_O\_DIR)=0x04;

如果要将PORTn的Pin3设置为输出：

HWREG(GPIO\_PORTN\_BASE + GPIO\_O\_DIR)=0x08;

如果要将PORTn的Pin0和Pin3都设置为输出：

HWREG(GPIO\_PORTN\_BASE + GPIO\_O\_DIR)=0x09;

由于这种每一位控制一个引脚的情况比较多，为了方便引用各个引脚，gpio.h中对各个引脚做了宏定义：

**#define** GPIO\_PIN\_0 0x00000001 // GPIO pin 0

**#define** GPIO\_PIN\_1 0x00000002 // GPIO pin 1

**#define** GPIO\_PIN\_2 0x00000004 // GPIO pin 2

**#define** GPIO\_PIN\_3 0x00000008 // GPIO pin 3

**#define** GPIO\_PIN\_4 0x00000010 // GPIO pin 4

**#define** GPIO\_PIN\_5 0x00000020 // GPIO pin 5

**#define** GPIO\_PIN\_6 0x00000040 // GPIO pin 6

**#define** GPIO\_PIN\_7 0x00000080 // GPIO pin 7

这样，如果要将PORTn的Pin3设置为输出：

HWREG(GPIO\_PORTN\_BASE + GPIO\_O\_DIR)= GPIO\_PIN\_3;

也可以用或操作，控制设置多个引脚

如果要将PORTn的Pin0和Pin3都设置为输出

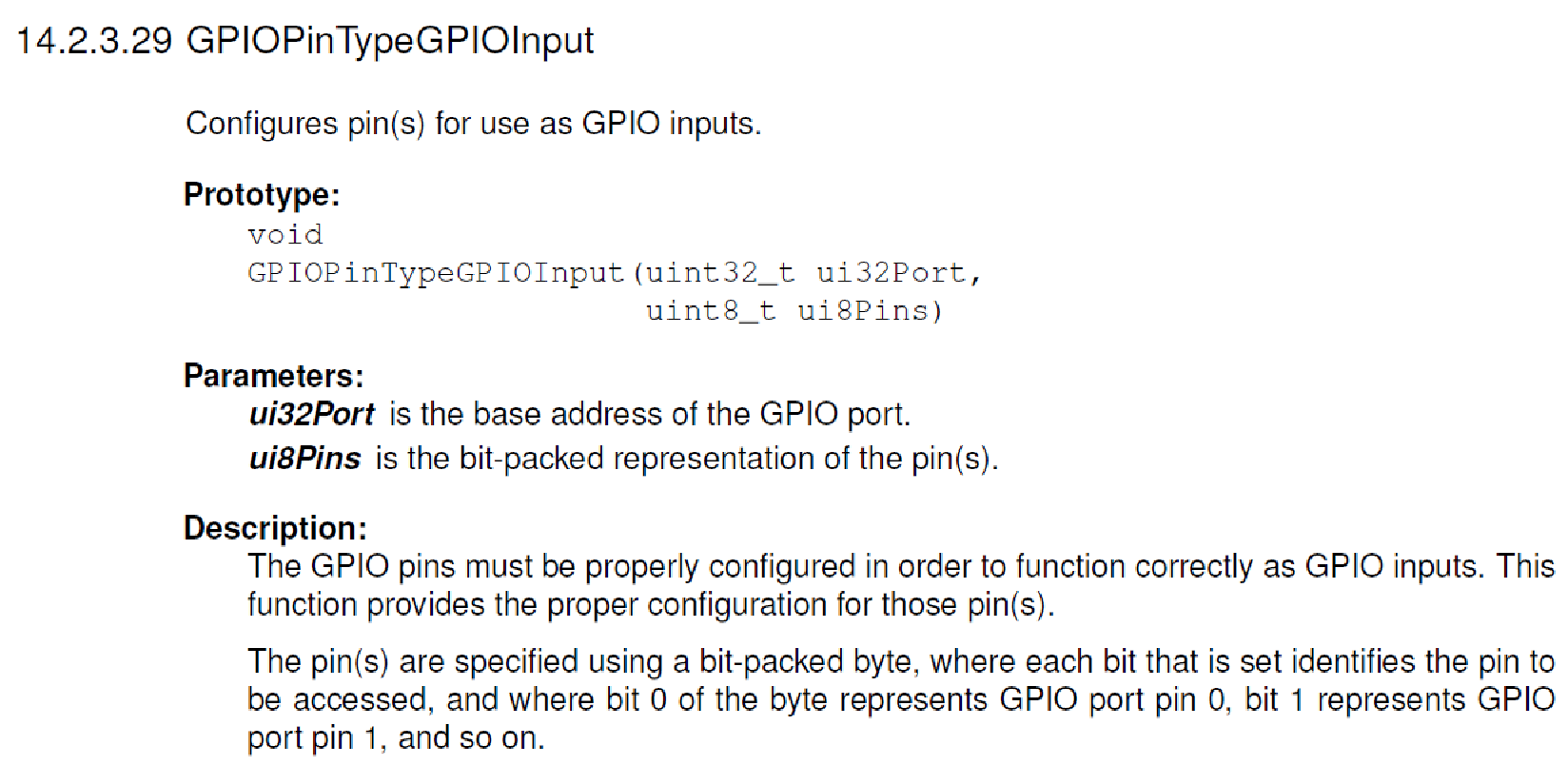
HWREG(GPIO\_PORTN\_BASE + GPIO\_O\_DIR)= GPIO\_PIN\_0 | GPIO\_PIN\_3;

TivaWare提供了库函数，可以直观的设置引脚的输入输出：

* void GPIOPinTypeGPIOInput (uint32\_t ui32Port, uint8\_t ui8Pins)
* void GPIOPinTypeGPIOOutput (uint32\_t ui32Port, uint8\_t ui8Pins)
* void GPIOPinTypeGPIOOutputOD (uint32\_t ui32Port, uint8\_t ui8Pins)

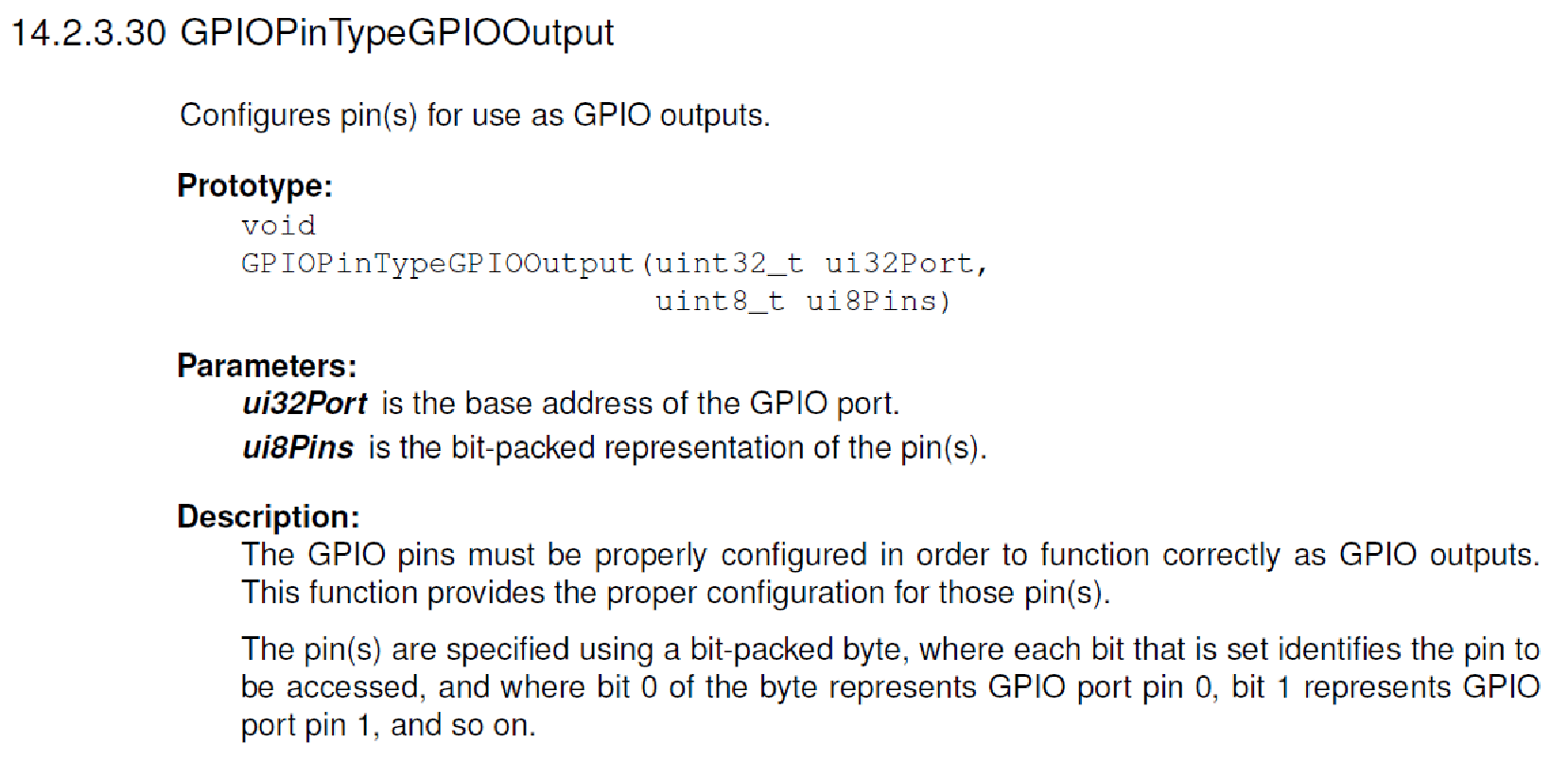
GPIOPinTypeGPIOInput (uint32\_t ui32Port, uint8\_t ui8Pins)：

完成将引脚设置为悬空输入的相关设置，两个参数 ui32Port指定端口的基地址，ui8Pins指定把哪些引脚设置为输入，ui8Pins为8位无符号变量，每一位代表一个引脚。SW-TM4C-DRL-UG-2.1.4.178.pdf文档中的具体说明如下：



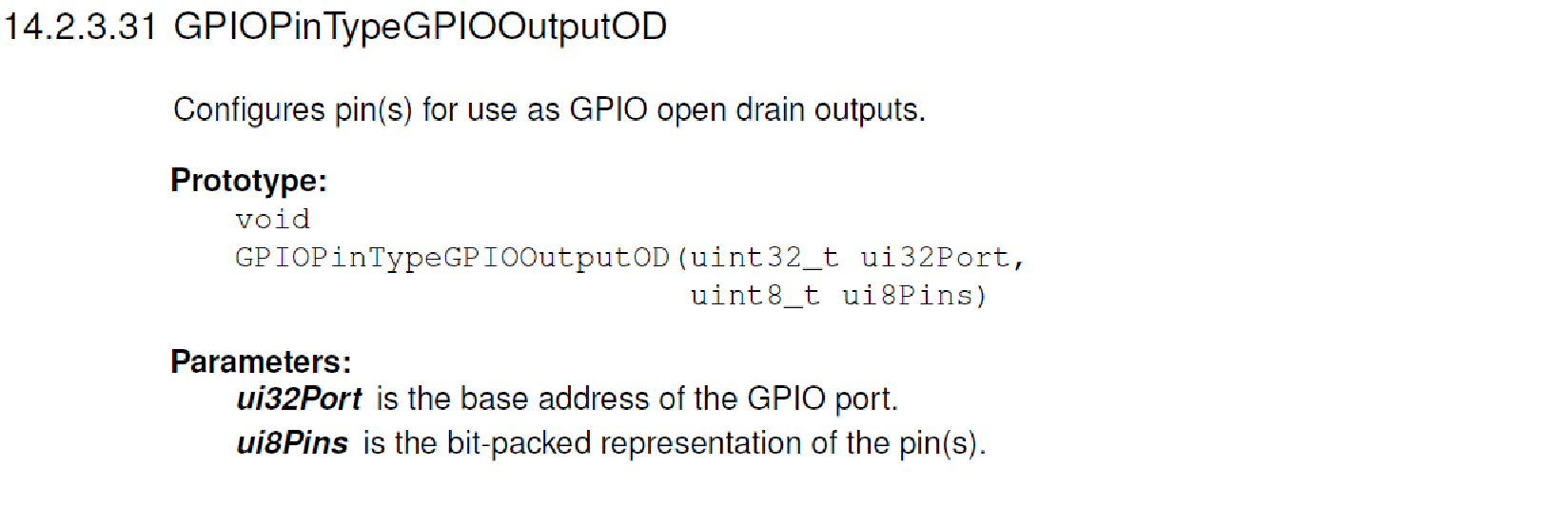
GPIOPinTypeGPIOOutput (uint32\_t ui32Port, uint8\_t ui8Pins)：

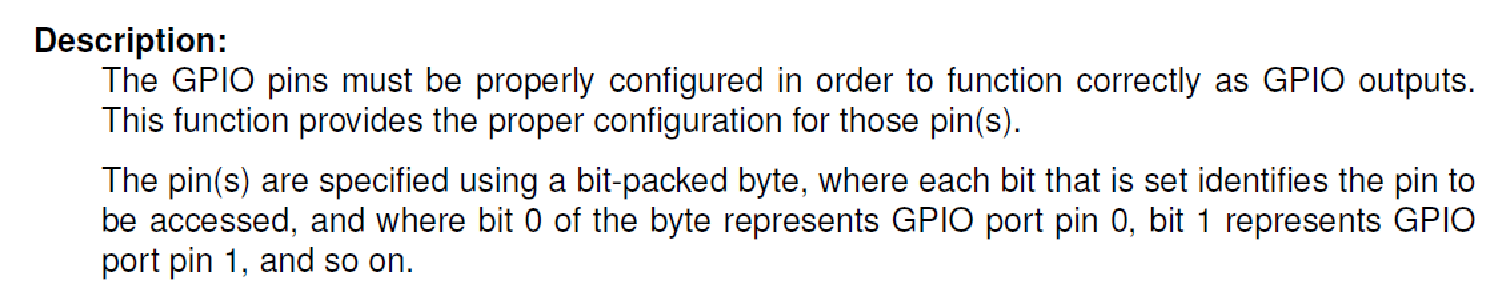
完成将引脚设置为推挽输出的相关设置，两个参数 ui32Port指定端口的基地址，ui8Pins指定把哪些引脚设置为输出，ui8Pins为8位无符号变量，每一位代表一个引脚。SW-TM4C-DRL-UG-2.1.4.178.pdf文档中的具体说明如下：



GPIOPinTypeGPIOOutputOD (uint32\_t ui32Port, uint8\_t ui8Pins)

完成将引脚设置为开漏输出的相关设置，两个参数 ui32Port指定端口的基地址，ui8Pins指定把哪些引脚设置为输出，ui8Pins为8位无符号变量，每一位代表一个引脚。SW-TM4C-DRL-UG-2.1.4.178.pdf文档中的具体说明如下：





这三个函数已经被封装在driverlib库中，无法看到源代码。TivaWare提供了这三个函数的源代码，位于C:\ti\TivaWare\_C\_Series-2.1.4.178\driverlib\gpio.c和同目录下的gpio.h中。

例如，GPIOPinTypeGPIOOutput函数的具体实现如下：

**void**

**GPIOPinTypeGPIOOutput**(uint32\_t ui32Port, uint8\_t ui8Pins)

{

// Check the arguments.

ASSERT(\_GPIOBaseValid(ui32Port));

// Set the pad(s) for standard push-pull operation.

GPIOPadConfigSet(ui32Port, ui8Pins, GPIO\_STRENGTH\_2MA, GPIO\_PIN\_TYPE\_STD);

// Make the pin(s) be outputs.

GPIODirModeSet(ui32Port, ui8Pins, GPIO\_DIR\_MODE\_OUT);

}

例如，GPIOPinTypeGPIOInput函数的具体实现如下：

**void**

**GPIOPinTypeGPIOInput**(uint32\_t ui32Port, uint8\_t ui8Pins)

{

// Check the arguments.

ASSERT(\_GPIOBaseValid(ui32Port));

// Make the pin(s) be inputs.

GPIODirModeSet(ui32Port, ui8Pins, GPIO\_DIR\_MODE\_IN);

// Set the pad(s) for standard push-pull operation.

GPIOPadConfigSet(ui32Port, ui8Pins, GPIO\_STRENGTH\_2MA, GPIO\_PIN\_TYPE\_STD);

}

该函数首先要确保输入的端口基地址是合法的，然后调用了GPIOPadConfigSet和GPIODirModeSet两个函数。GPIOPadConfigSet的具体内容我们稍后再讲，先看GPIODirModeSet函数，传入了三个参数，分别是端口基地址ui32Port、要操作的引脚ui8Pins和GPIO\_DIR\_MODE\_OUT。GPIO\_DIR\_MODE\_OUT在gpio.h中定义：0代表输入，1代表输出，GPIO\_DIR\_MODE\_OUT=1。

**#define** GPIO\_DIR\_MODE\_IN 0x00000000 // Pin is a GPIO input

**#define** GPIO\_DIR\_MODE\_OUT 0x00000001 // Pin is a GPIO output

GPIODirModeSet函数在SW-TM4C-DRL-UG-2.1.4.178.pdf文档中的具体说明如下：

设置指定引脚的方向，GPIO\_DIR\_MODE\_IN代表输入，GPIO\_DIR\_MODE\_OUT代表输出，GPIO\_DIR\_MODE\_HW代表方向由GPIO模块外的其他外设硬件控制。



再看GPIODirModeSet函数的实现（在gpio.c中）：

**void**

**GPIODirModeSet**(uint32\_t ui32Port, uint8\_t ui8Pins, uint32\_t ui32PinIO)

{

// Check the arguments.

ASSERT(\_GPIOBaseValid(ui32Port));

ASSERT((ui32PinIO == GPIO\_DIR\_MODE\_IN) ||

(ui32PinIO == GPIO\_DIR\_MODE\_OUT) ||

(ui32PinIO == GPIO\_DIR\_MODE\_HW));

// Set the pin direction and mode.

HWREG(ui32Port + GPIO\_O\_DIR) = ((ui32PinIO & 1) ?

(HWREG(ui32Port + GPIO\_O\_DIR) | ui8Pins) :

(HWREG(ui32Port + GPIO\_O\_DIR) & ~(ui8Pins)));

HWREG(ui32Port + GPIO\_O\_AFSEL) = ((ui32PinIO & 2) ?

(HWREG(ui32Port + GPIO\_O\_AFSEL) |

ui8Pins) :

(HWREG(ui32Port + GPIO\_O\_AFSEL) &

~(ui8Pins)));

}

首先，先判断端口基地址和输入输出设置是否合法，然后设置GPIO\_O\_DIR寄存器。

如果ui32PinIO为1，即设置为输出那么赋给GPIO\_O\_DIR寄存器的值为:

(HWREG(ui32Port + GPIO\_O\_DIR) | ui8Pins)

先读回GPIO\_O\_DIR寄存器的当前数值，然后通过或运算，将本次要设置的位置1.

如果ui32PinIO为0，即设置为输出那么赋给GPIO\_O\_DIR寄存器的值为:

(HWREG(ui32Port + GPIO\_O\_DIR) & ~(ui8Pins))

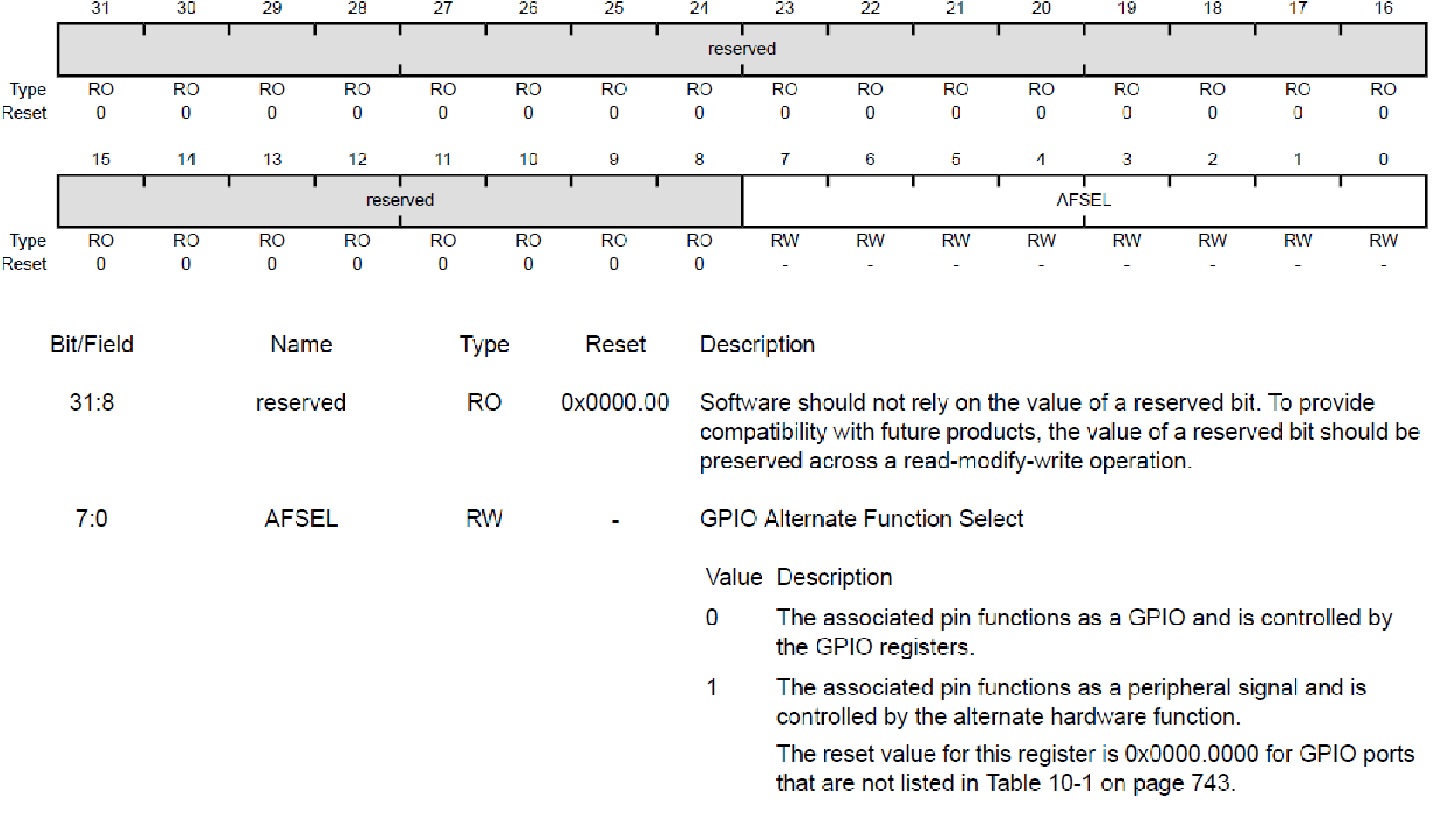
先读回GPIO\_O\_DIR寄存器的当前数值，然后通过与运算和取反运算，将本次要设置的位置0.

操作完GPIO\_O\_DIR寄存器后，还需要设置GPIO\_O\_AFSEL寄存器，由于配置输入输出时，ui32PinIO只为0或1，所以赋给GPIO\_O\_AFSEL寄存器的值为(HWREG(ui32Port + GPIO\_O\_AFSEL) &~(ui8Pins))，即把本次要设置的位置0

**复用选择寄存器GPIOAFSEL**

只有低八位有实际作用，用于控制该端口的八个引脚是否启用复用功能，每一位控制一个引脚。

0为不复用，1为复用。复用的功能，由GPIOPCTL决定



因此，(HWREG(ui32Port + GPIO\_O\_AFSEL) &~(ui8Pins))代表关闭要设置的位的复用功能，使该引脚作为普通GPIO。

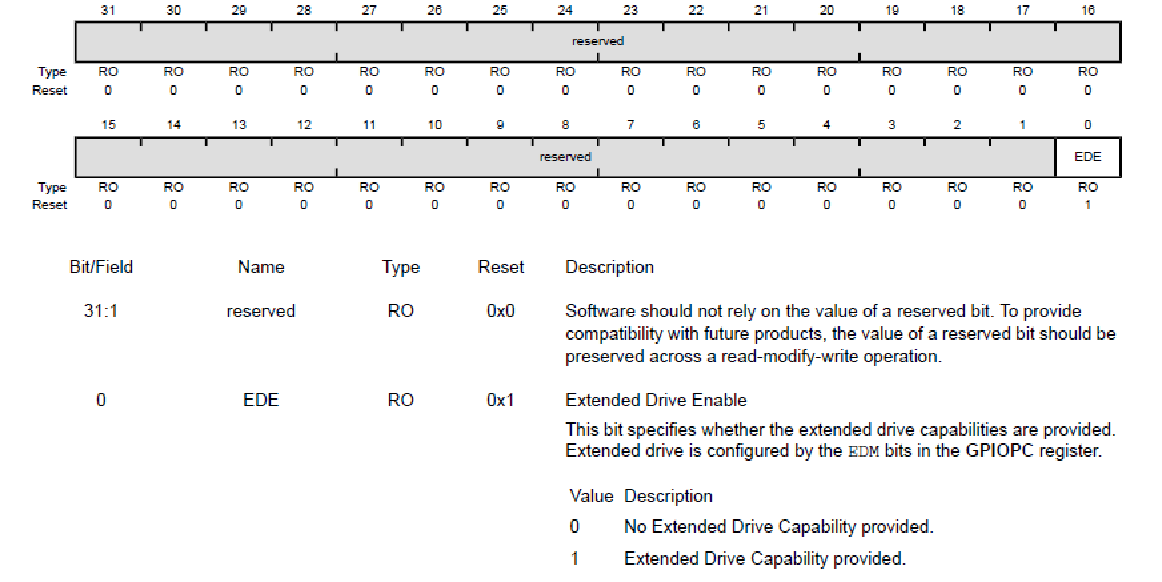
然后我们需要了解GPIOPadConfigSet函数，在此之前，我们需要了解另外几个寄存器：

**输出驱动能力配置寄存器GPIODR2R，GPIODR4R，GPIODR8R，GPIODR12R，GPIOPP，GPIOPC**

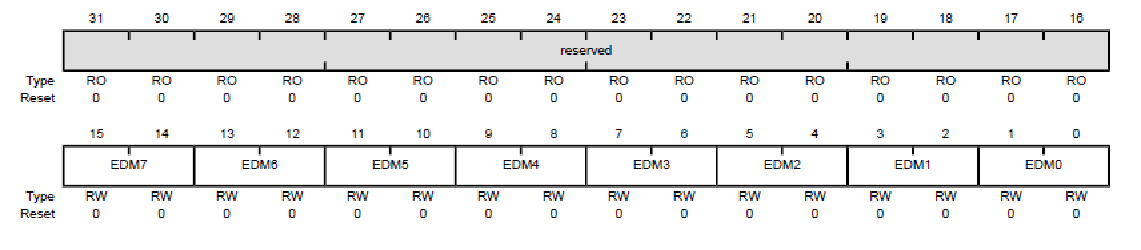
这几个寄存器用于控制引脚的最大电流输出能力，可以是2mA、4mA、8mA、10mA或者12mA。

GPIODR2R，GPIODR4R，GPIODR8R，GPIODR12R这几个寄存器都是只有低八位有实际作用，用于控制该端口的对应的八个引脚的输出能力，每一位控制一个引脚。

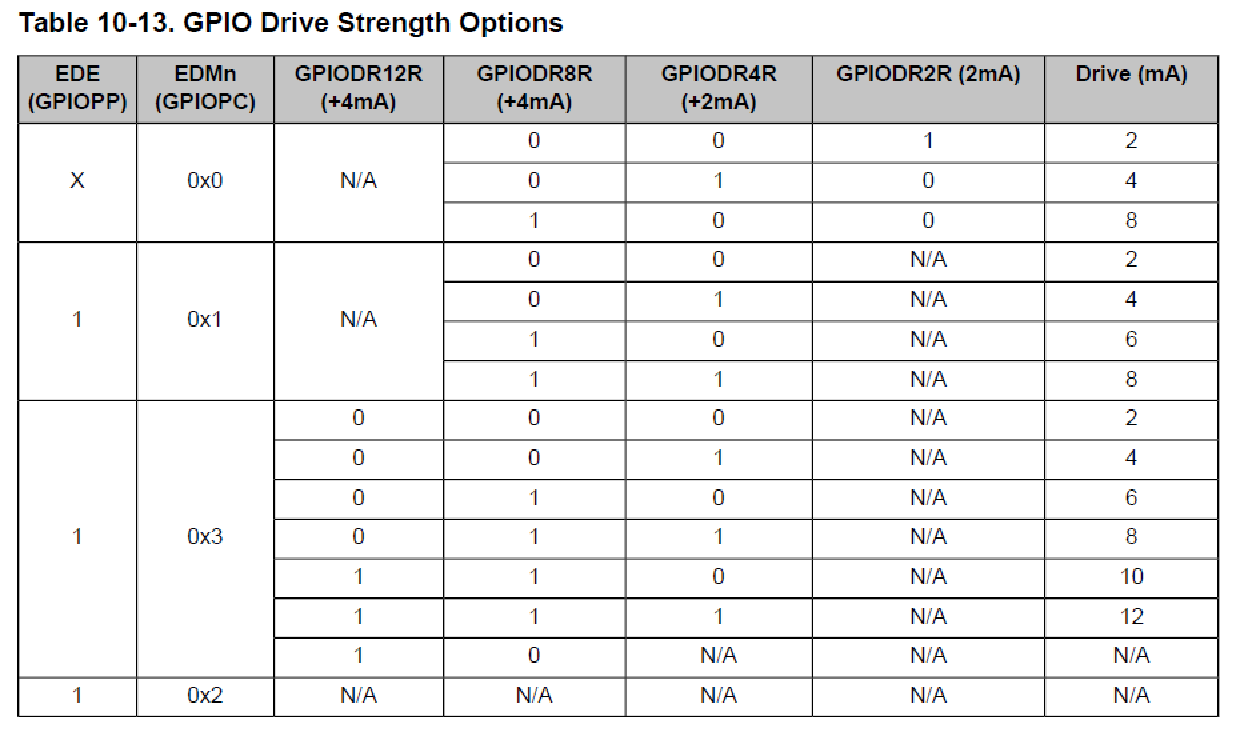
GPIOPP寄存器只有一位有作用，控制是否开启强化输出模式



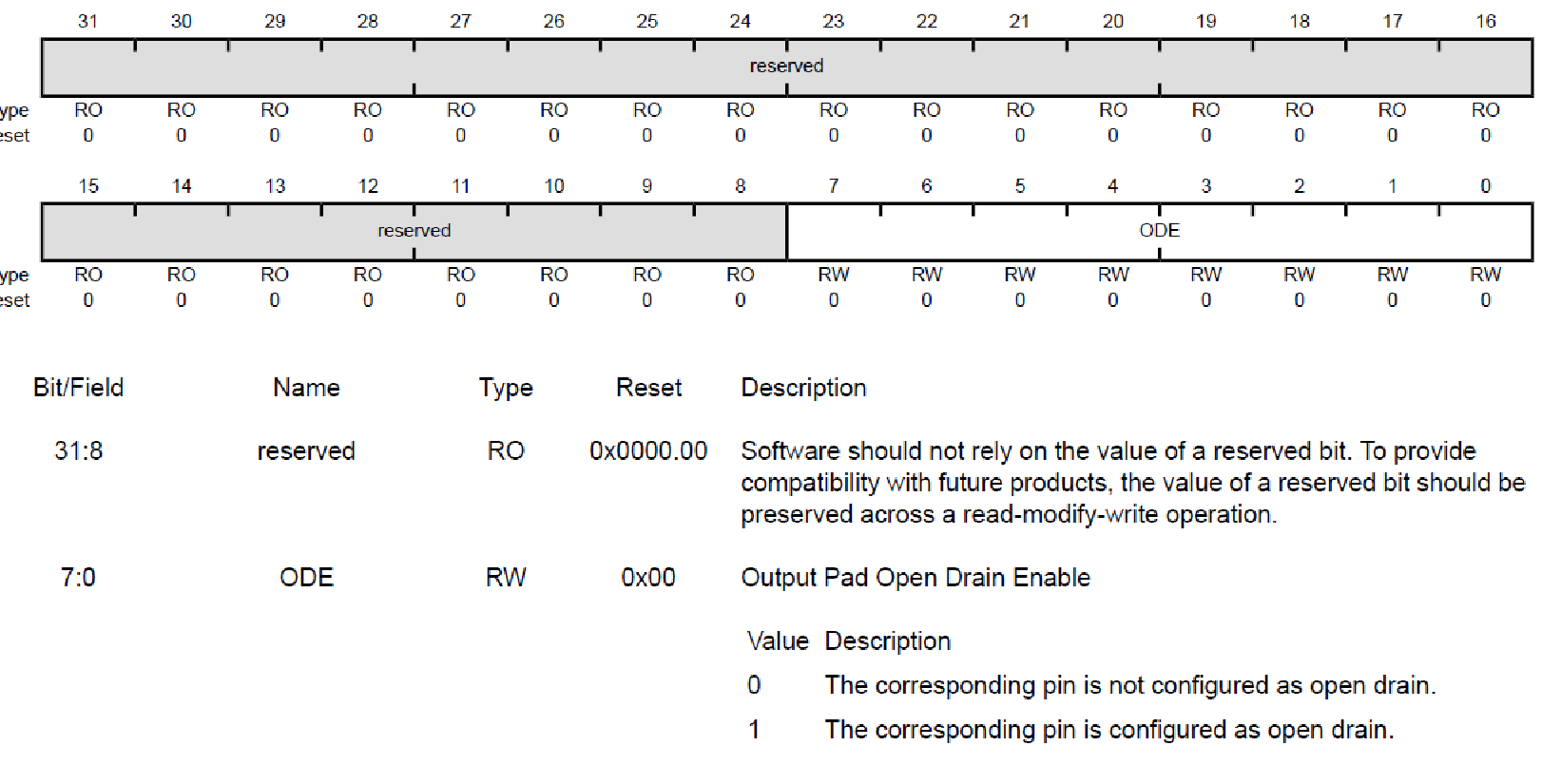
GPIOPC寄存器的低16位有意义，每两位控制一个引脚：



GPIODR2R，GPIODR4R，GPIODR8R，GPIODR12R，GPIOPP，GPIOPC这几个寄存器对应的位的组合，决定了对应引脚的最大电流输出能力，如下表所示。



**开漏输出配置寄存器GPIOODR**

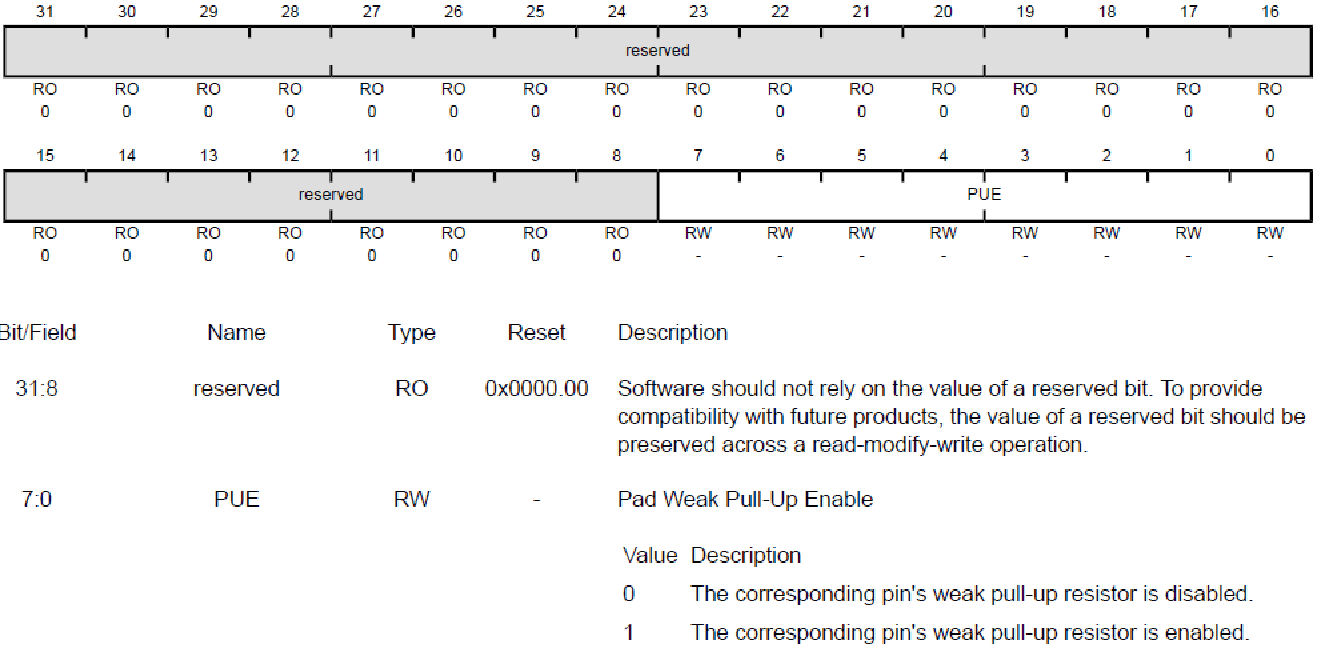


**上拉电阻与下拉电阻配置寄存器： GPIOPUR ，GPIOPDR**

这两个寄存器都是只有低八位有实际作用，用于控制该端口的八个引脚是否连接上拉电阻、下拉电阻，每一位控制一个引脚。

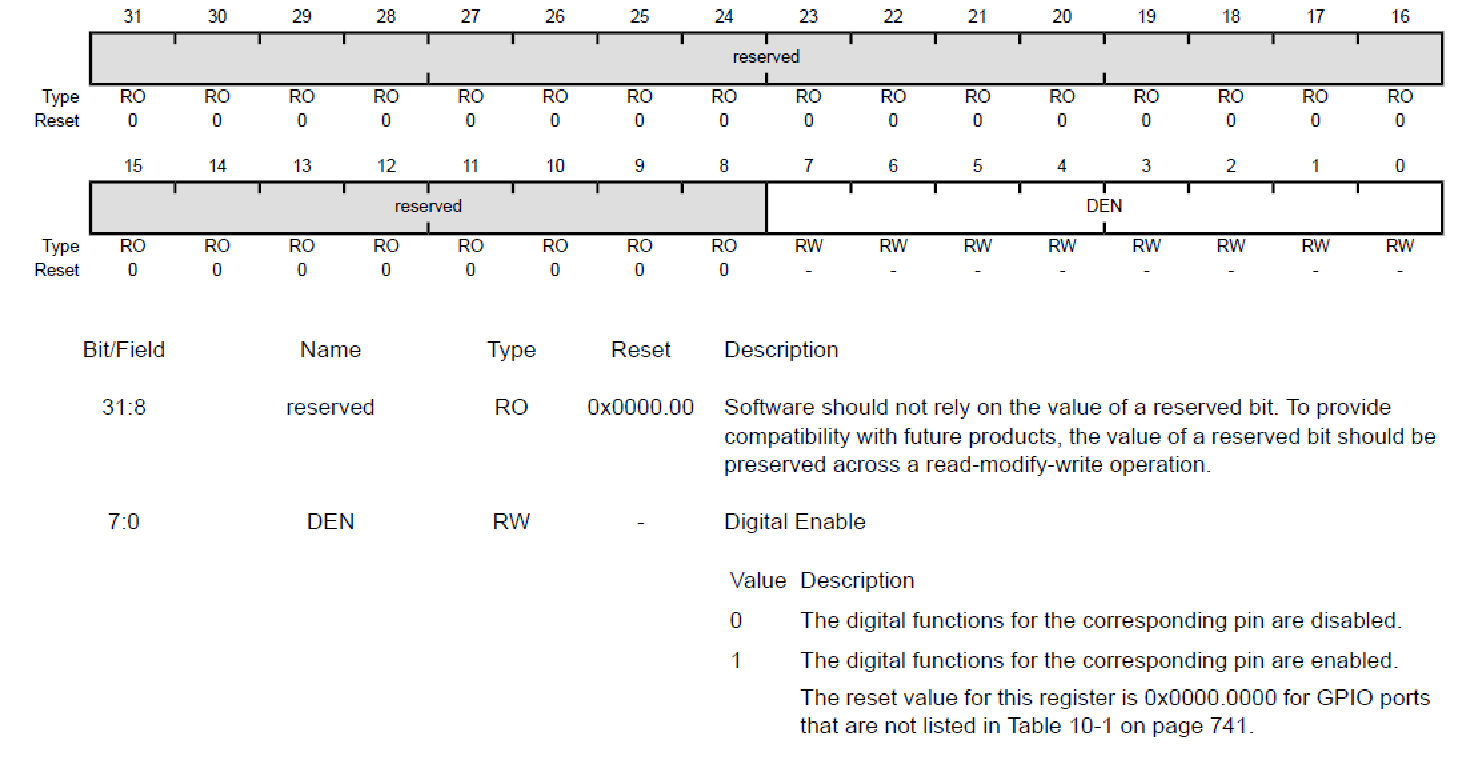
GPIOPUR 对应的位为0表示该引脚不连接上拉电阻，GPIOPUR 对应的位为1表示该引脚连接上拉电阻

GPIOPDR对应的位为0表示该引脚不连接上拉电阻，GPIOPDR 对应的位为1表示该引脚连接上拉电阻



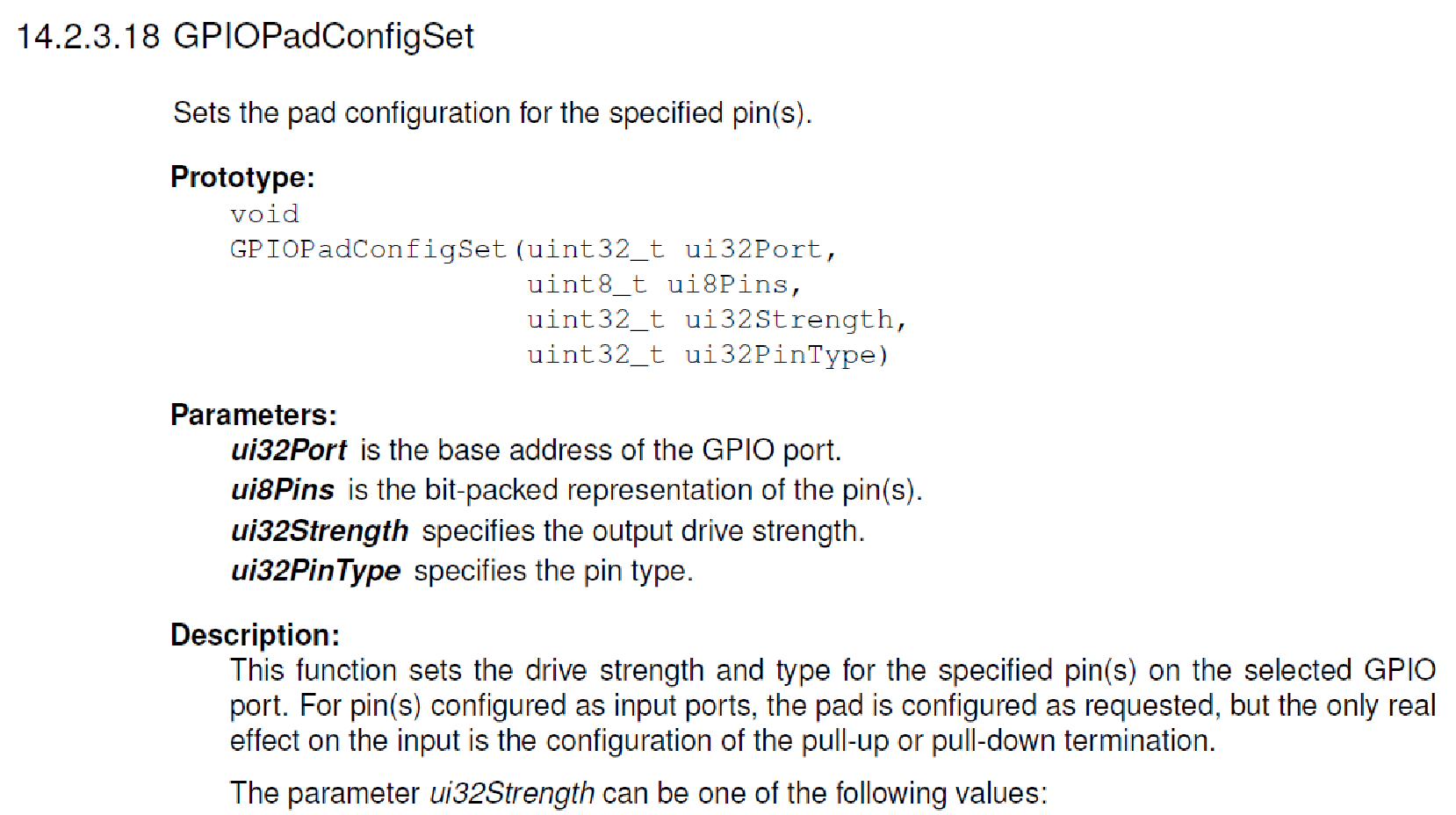
**数字逻辑功能使能寄存器GPIODEN：**

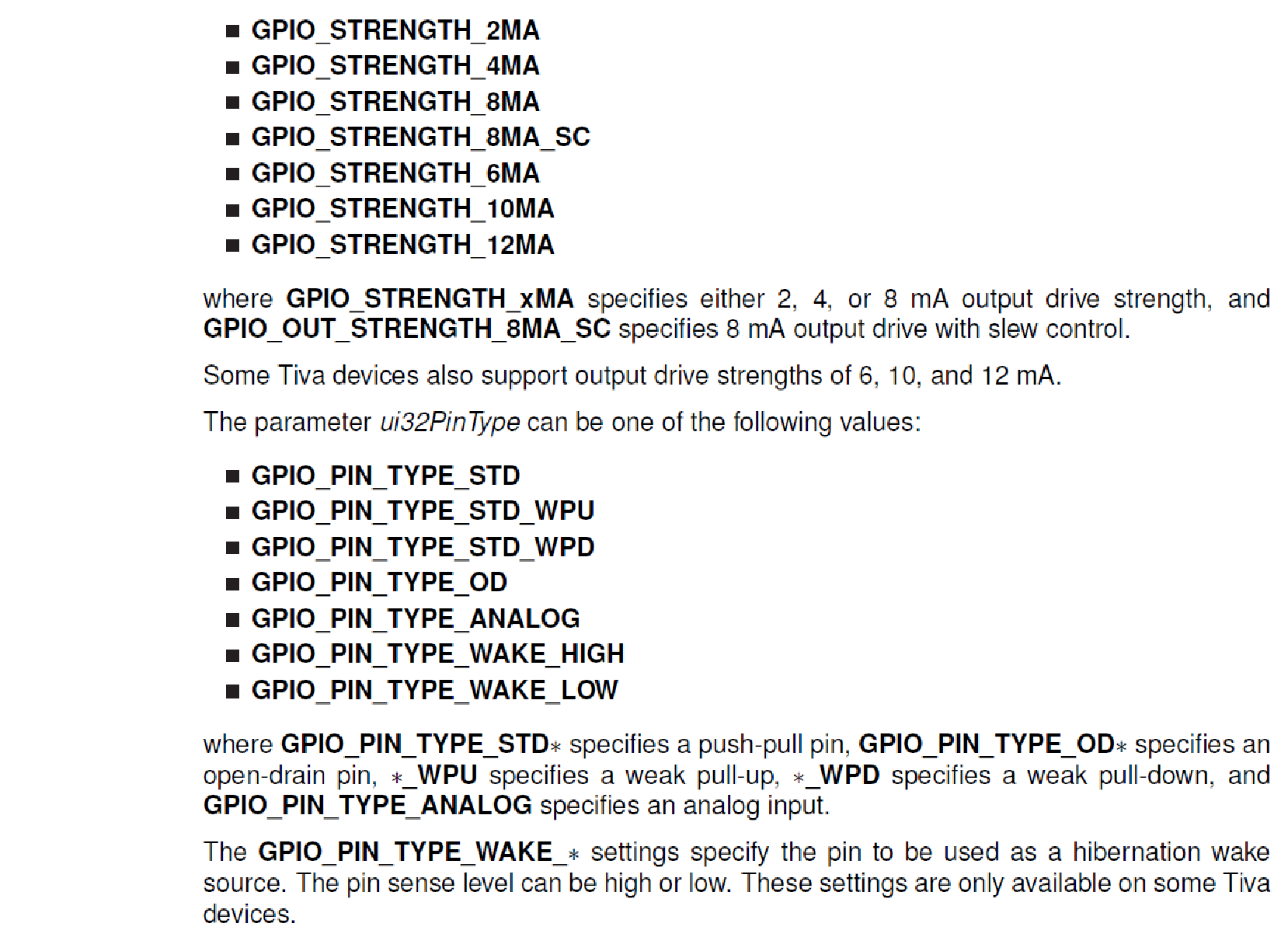
重置后，引脚自动配置为三态，断开引脚与GPIO模块的连接，如果要使用引脚的数字逻辑功能，需要使能GPIODEN寄存器中相应的位：

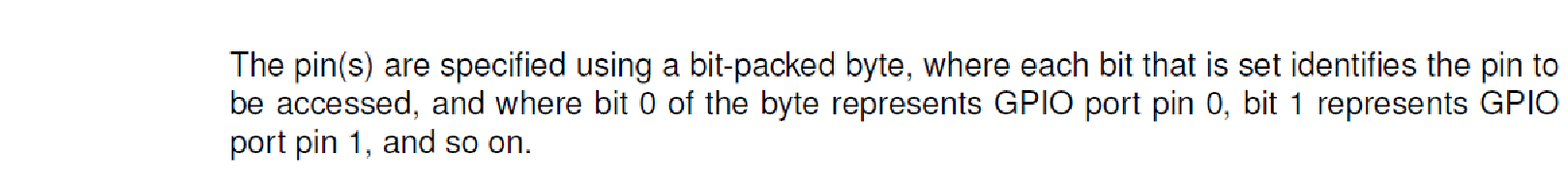


了解了驱动能力、开漏输出、上拉电阻、下拉电阻、数字逻辑使能等相关配置后，我们再看GPIOPadConfigSet的功能描述：

GPIOPadConfigSet函数用于设置引脚的电流输出能力和引脚类型，如上拉、下拉、开漏等。







那么在GPIOPinTypeGPIOOutput函数中，调用的

GPIOPadConfigSet(ui32Port, ui8Pins, GPIO\_STRENGTH\_2MA, GPIO\_PIN\_TYPE\_STD),

作用是将指定的端口的引脚，设定为推挽输出，最大输出电流为2mA。

GPIOPadConfigSet的具体实现方式在gpio.c中，首先，设置GPIOPC寄存器：

**for**(ui8Bit = 0; ui8Bit < 8; ui8Bit++)

{

**if**(ui8Pins & (1 << ui8Bit))

{

HWREG(ui32Port + GPIO\_O\_PC) = (HWREG(ui32Port + GPIO\_O\_PC) &

~(0x3 << (2 \* ui8Bit)));

HWREG(ui32Port + GPIO\_O\_PC) |= (((ui32Strength >> 5) & 0x3) <<

(2 \* ui8Bit));

}

}

驱动能力ui32Strength的相关定义在gpio.h中：

**#define** GPIO\_STRENGTH\_2MA 0x00000001 // 2mA drive strength

**#define** GPIO\_STRENGTH\_4MA 0x00000002 // 4mA drive strength

**#define** GPIO\_STRENGTH\_6MA 0x00000065 // 6mA drive strength

**#define** GPIO\_STRENGTH\_8MA 0x00000066 // 8mA drive strength

**#define** GPIO\_STRENGTH\_8MA\_SC 0x0000006E // 8mA drive with slew rate control

**#define** GPIO\_STRENGTH\_10MA 0x00000075 // 10mA drive strength

**#define** GPIO\_STRENGTH\_12MA 0x00000077 // 12mA drive strength

首先把要操作的引脚，对应的GPIOPC寄存器里的位清零。然后，如果驱动能力大于等于6mA，则设置该引脚GPIOPC中对应的位为0x03，如果驱动能力设置为2mA或者4mA，那么设置该引脚GPIOPC中对应的位为0x00。

然后设置GPIODR2R，GPIODR4R，GPIODR8R，GPIODR12R这几个寄存器

HWREG(ui32Port + GPIO\_O\_DR2R) = ((ui32Strength & 1) ?

(HWREG(ui32Port + GPIO\_O\_DR2R) |

ui8Pins) :

(HWREG(ui32Port + GPIO\_O\_DR2R) &

~(ui8Pins)));

HWREG(ui32Port + GPIO\_O\_DR4R) = ((ui32Strength & 2) ?

(HWREG(ui32Port + GPIO\_O\_DR4R) |

ui8Pins) :

(HWREG(ui32Port + GPIO\_O\_DR4R) &

~(ui8Pins)));

HWREG(ui32Port + GPIO\_O\_DR8R) = ((ui32Strength & 4) ?

(HWREG(ui32Port + GPIO\_O\_DR8R) |

ui8Pins) :

(HWREG(ui32Port + GPIO\_O\_DR8R) &

~(ui8Pins)));

HWREG(ui32Port + GPIO\_O\_DR12R) = ((ui32Strength & 0x10) ?

(HWREG(ui32Port + GPIO\_O\_DR12R) |

ui8Pins) :

(HWREG(ui32Port + GPIO\_O\_DR12R) &

~(ui8Pins)));

最后设置GPIOOD，GPIOPUR，GPIOPDR，GPIODEN这几个寄存器，设置要操作的引脚对应的位：

HWREG(ui32Port + GPIO\_O\_ODR) = ((ui32PinType & 1) ?

(HWREG(ui32Port + GPIO\_O\_ODR) | ui8Pins) :

(HWREG(ui32Port + GPIO\_O\_ODR) & ~(ui8Pins)));

HWREG(ui32Port + GPIO\_O\_PUR) = ((ui32PinType & 2) ?

(HWREG(ui32Port + GPIO\_O\_PUR) | ui8Pins) :

(HWREG(ui32Port + GPIO\_O\_PUR) & ~(ui8Pins)));

HWREG(ui32Port + GPIO\_O\_PDR) = ((ui32PinType & 4) ?

(HWREG(ui32Port + GPIO\_O\_PDR) | ui8Pins) :

(HWREG(ui32Port + GPIO\_O\_PDR) & ~(ui8Pins)));

HWREG(ui32Port + GPIO\_O\_DEN) = ((ui32PinType & 8) ?

(HWREG(ui32Port + GPIO\_O\_DEN) | ui8Pins) :

(HWREG(ui32Port + GPIO\_O\_DEN) & ~(ui8Pins)));

其中ui32PinType的枚举类型如下：

**#define** GPIO\_PIN\_TYPE\_STD 0x00000008 // Push-pull

**#define** GPIO\_PIN\_TYPE\_STD\_WPU 0x0000000A // Push-pull with weak pull-up

**#define** GPIO\_PIN\_TYPE\_STD\_WPD 0x0000000C // Push-pull with weak pull-down

**#define** GPIO\_PIN\_TYPE\_OD 0x00000009 // Open-drain

**#define** GPIO\_PIN\_TYPE\_ANALOG 0x00000000 // Analog comparator

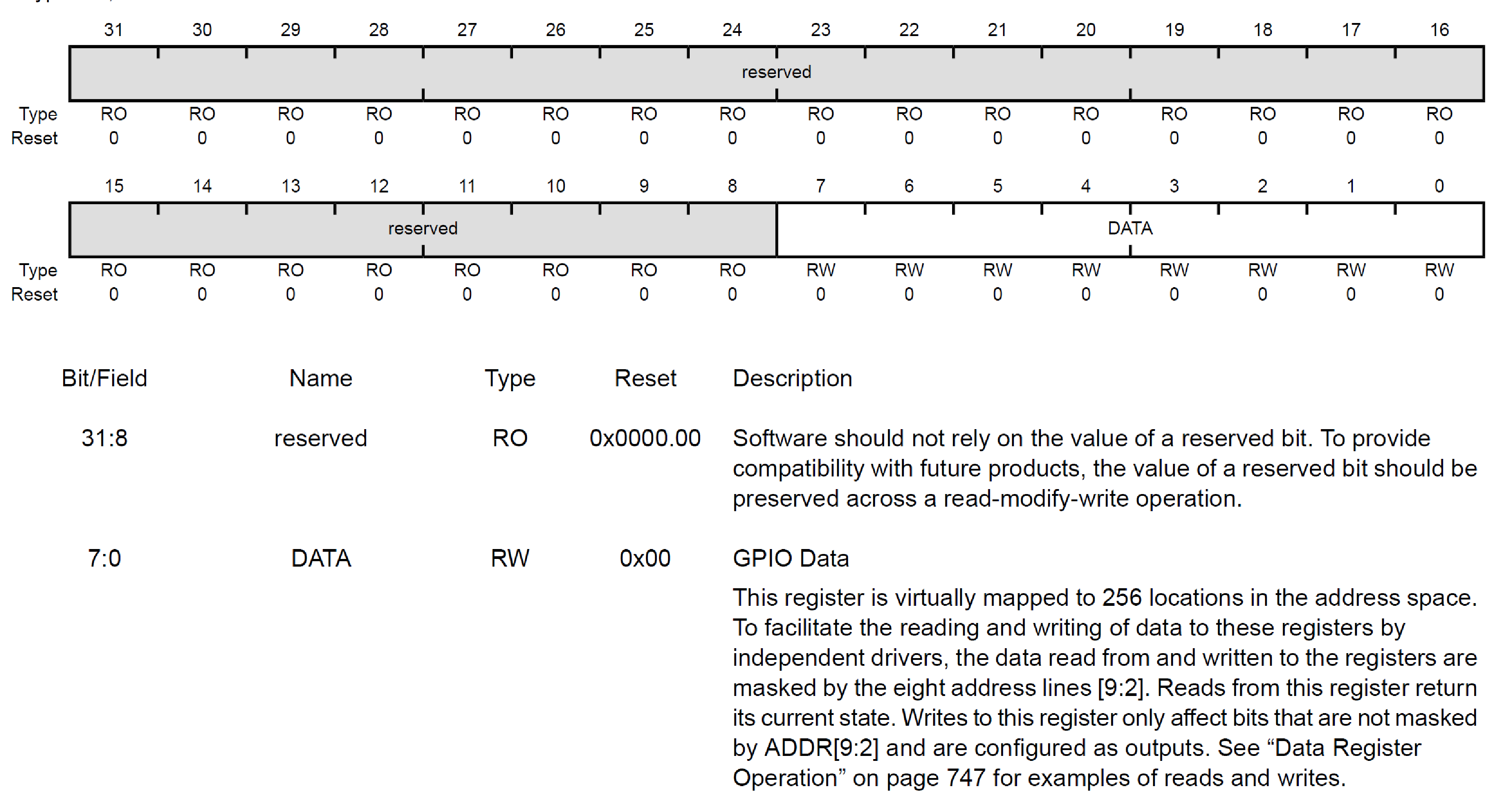
**#define** GPIO\_PIN\_TYPE\_WAKE\_HIGH 0x00000208 // Hibernate wake, high

**#define** GPIO\_PIN\_TYPE\_WAKE\_LOW 0x00000108 // Hibernate wake, low

# 数据寄存器的功能与操作

**数据寄存器GPIODATA：**

低八位有意义，代表了每个引脚的电平：



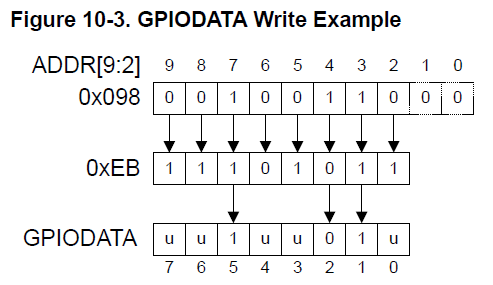
如果GPIODIR寄存器配置为输出，那么这个寄存器相应的位的电平，会直接传递到对应的引脚上。

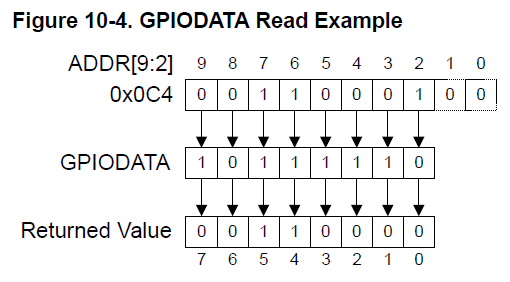
如果GPIODIR寄存器配置为输入，那么这个寄存器相应的位的电平，代表对应的引脚上的电平。

读取和写入GPIODATA：

为了方便写入和读出特定的位，而不影响其他位，采用了数据结合屏蔽位的读写方式，屏蔽位是偏移地址的低[9:2]位：

只有屏蔽位对应的位为1时，才能对GPIODATA寄存器进行读写。





TivaWare在gpio.h中提供了GPIOPinWrite和GPIOPinRead函数，对GPIODATA寄存器进行读写：

在gpio.c中，提供了这两个函数的具体实现方式：

**void**

**GPIOPinWrite**(uint32\_t ui32Port, uint8\_t ui8Pins, uint8\_t ui8Val)

{

// Check the arguments.

ASSERT(\_GPIOBaseValid(ui32Port));

// Write the pins.

HWREG(ui32Port + (GPIO\_O\_DATA + (ui8Pins << 2))) = ui8Val;

}

int32\_t

**GPIOPinRead**(uint32\_t ui32Port, uint8\_t ui8Pins)

{

// Check the arguments.

ASSERT(\_GPIOBaseValid(ui32Port));

// Return the pin value(s).

**return**(HWREG(ui32Port + (GPIO\_O\_DATA + (ui8Pins << 2))));

}

ui32Port为端口基地址，ui8Pins是要操作的位，是GPIO\_PIN\_0-7的组合（或逻辑），ui8Val是要操作的位对应的值，也可以用GPIO\_PIN\_0-7的组合（或逻辑）表示。

**#define** GPIO\_PIN\_0 0x00000001 // GPIO pin 0

**#define** GPIO\_PIN\_1 0x00000002 // GPIO pin 1

**#define** GPIO\_PIN\_2 0x00000004 // GPIO pin 2

**#define** GPIO\_PIN\_3 0x00000008 // GPIO pin 3

**#define** GPIO\_PIN\_4 0x00000010 // GPIO pin 4

**#define** GPIO\_PIN\_5 0x00000020 // GPIO pin 5

**#define** GPIO\_PIN\_6 0x00000040 // GPIO pin 6

**#define** GPIO\_PIN\_7 0x00000080 // GPIO pin 7

比如，将PORTn的0号引脚设置为1，将PORTn的1号引脚设置为0：

**GPIOPinWrite**(GPIO\_PORTN\_BASE, (GPIO\_PIN\_0| GPIO\_PIN\_1), GPIO\_PIN\_0);

将PORTn的0号引脚设置为0，将PORTn的1号引脚设置为1：

**GPIOPinWrite**(GPIO\_PORTN\_BASE, (GPIO\_PIN\_0| GPIO\_PIN\_1), GPIO\_PIN\_1);

将PORTn的0号引脚设置为1，将PORTn的1号引脚设置为1：

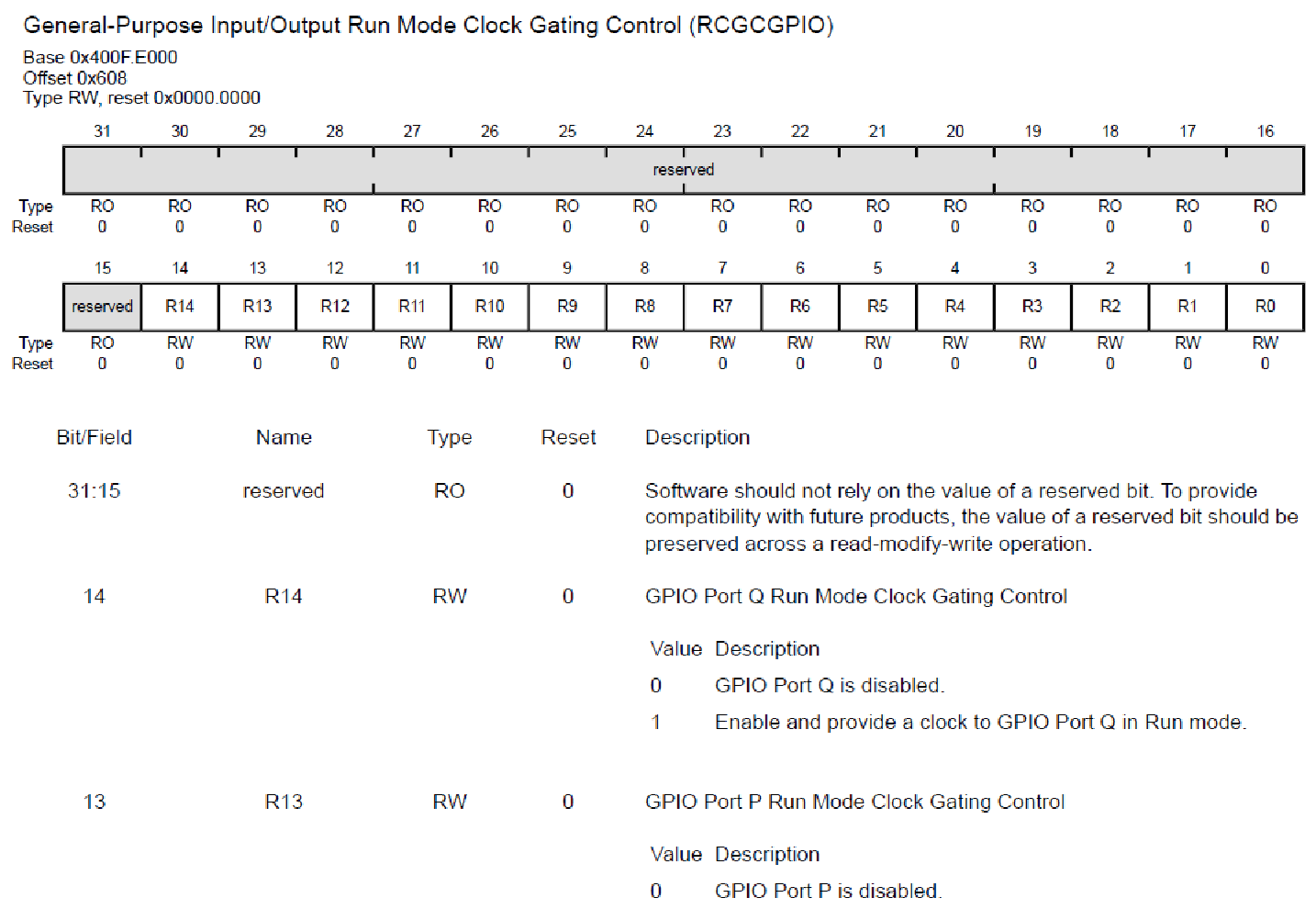
**GPIOPinWrite**(GPIO\_PORTN\_BASE, (GPIO\_PIN\_0| GPIO\_PIN\_1), GPIO\_PIN\_0 | GPIO\_PIN\_1);

将PORTn的0号引脚设置为0，将PORTn的1号引脚设置为0：

**GPIOPinWrite**(GPIO\_PORTN\_BASE, (GPIO\_PIN\_0| GPIO\_PIN\_1), 0x00);

# GPIO模块的使能

系统控制模块中的RCGCGPIO寄存器控制了14个GPIO的时钟，只有打开GPIO模块的时钟，GPIO模块才能工作：



R0到R14控制了GPIOA到GPIOQ。

RCGCGPIO的地址为基地址+偏移地址：

0x400FE000+0x608=0x400FE608

TivaWare库中，提供了SysCtlPeripheralEnable函数，用于开启外设，如开启GPION模块时钟：

**SysCtlPeripheralEnable**(SYSCTL\_PERIPH\_GPION);

SysCtlPeripheralEnable函数包含在driverlib.lib库中，其源代码在C:\ti\TivaWare\_C\_Series-2.1.4.178\driverlib\sysctl.c文件中进行了实现：

**void**

**SysCtlPeripheralEnable**(uint32\_t ui32Peripheral)

{

// Check the arguments.

ASSERT(\_SysCtlPeripheralValid(ui32Peripheral));

// Enable this peripheral.

HWREGBITW(SYSCTL\_RCGCBASE + ((ui32Peripheral & 0xff00) >> 8),

ui32Peripheral & 0xff) = 1;

}

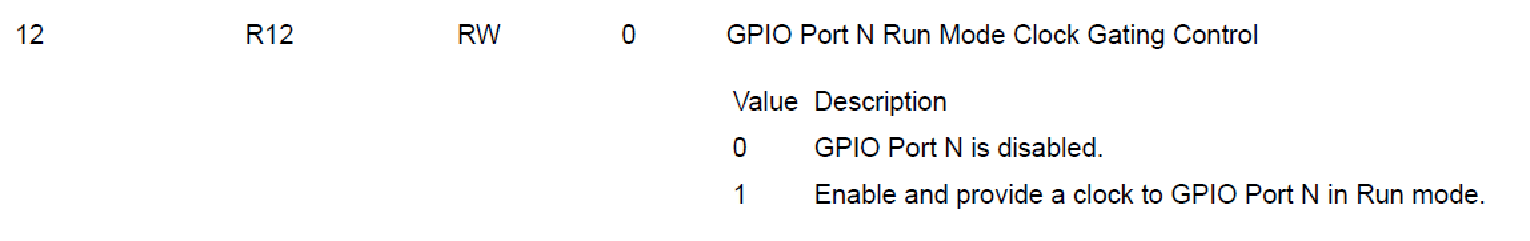
其中SYSCTL\_RCGCBASE的定义为：

**#define** SYSCTL\_RCGCBASE 0x400fe600

传入的参数SYSCTL\_PERIPH\_GPION的定义为：

**#define** SYSCTL\_PERIPH\_GPION 0xf000080c // GPIO N

这样，SYSCTL\_RCGCBASE + ((ui32Peripheral & 0xff00) >> 8)的结果就是0x400fe608，即RCGCGPIO寄存器，然后将RCGCGPIO的第12位设置为1，即开启GPION的时钟。

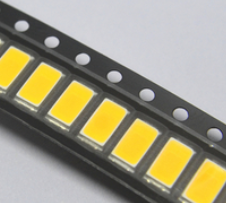
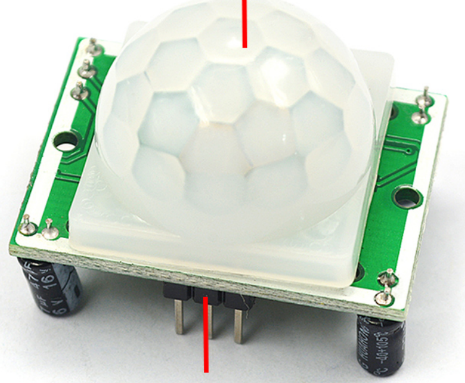


# GPIO模块的应用1 矩阵键盘

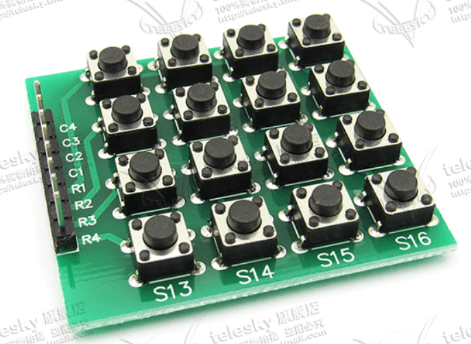
常用的IO驱动电路

* 简单按键开关
* 矩阵键盘
* 继电器
* LED灯
* 数码管
* 传感器开关输入

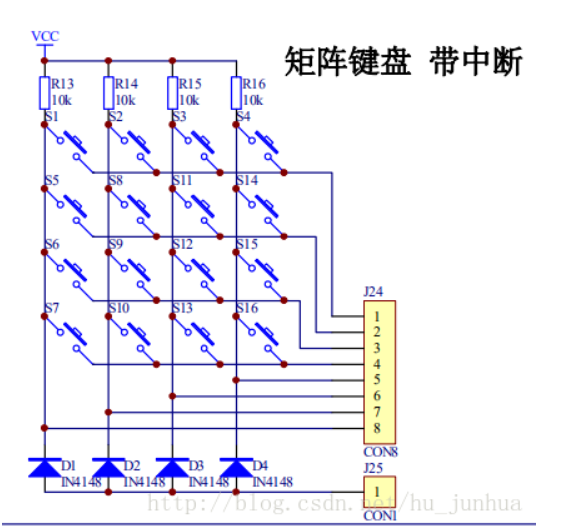
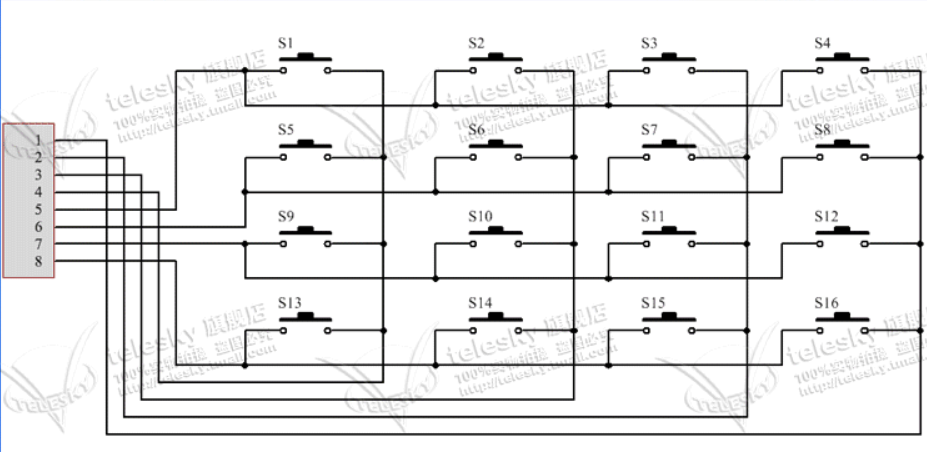
   

**矩阵键盘：**



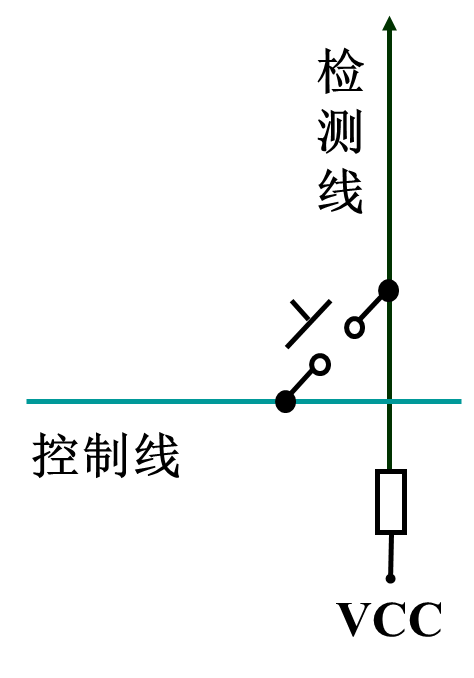
电路连接：



这种键盘有4根行线和4根列线，连接方式像一个4×4的矩阵，因此叫做矩阵键盘。

考虑只有一个按键按下的情况：

每一个按键按下，都可以把他所在的行线和列线短接。



把行线作为控制线，列线作为检测线。

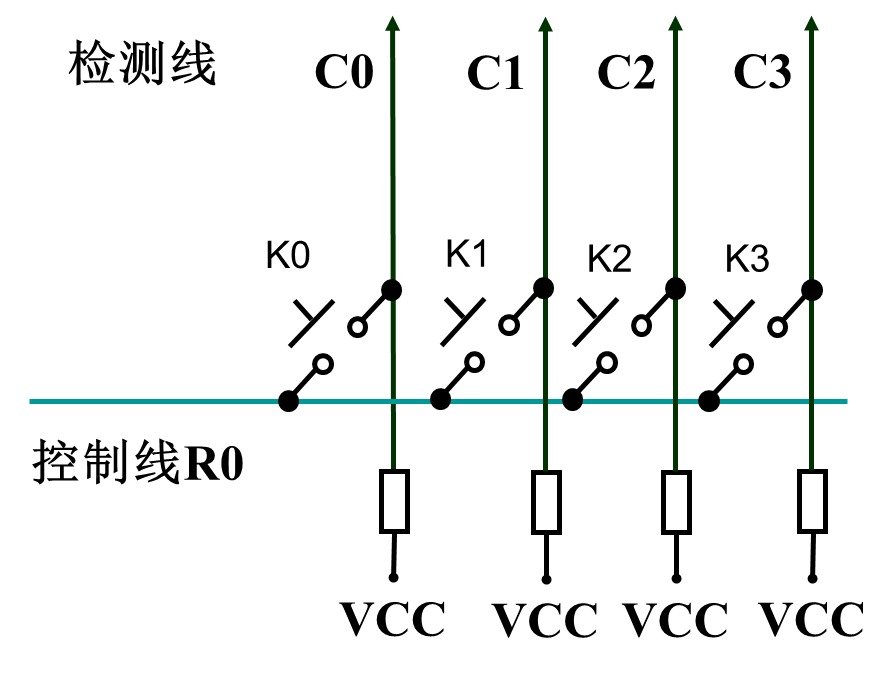
按键按下时，他所在的控制线与检测线相短接。

用上拉电阻把检测线上拉至VCC，

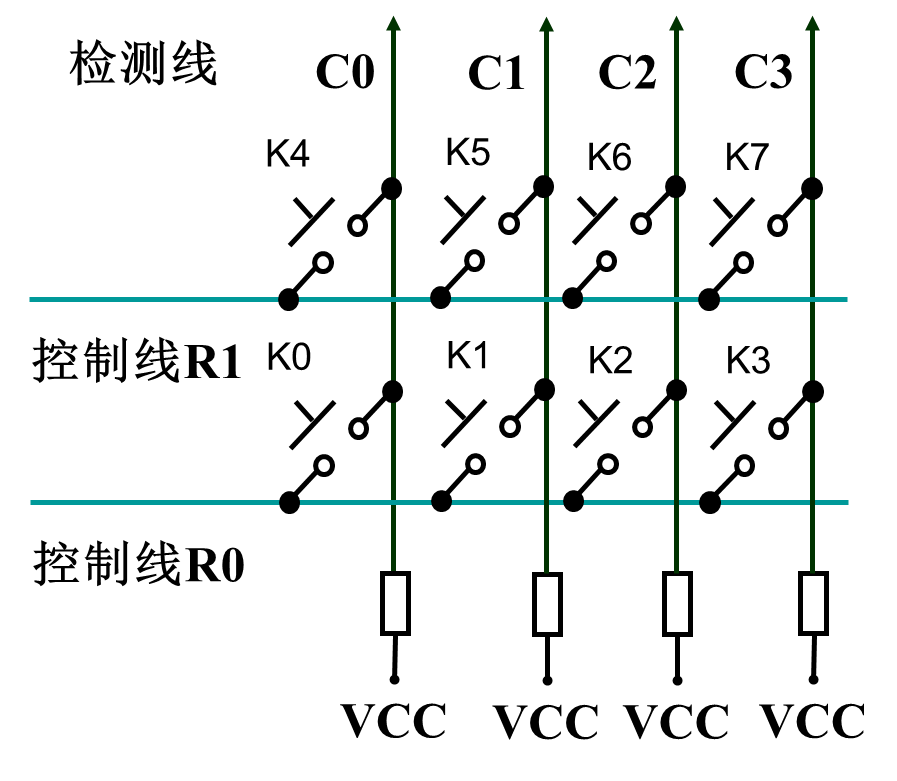
如果在控制线上输出低电平。按键没有按下时，检测线上的电压为高电平。按键按下时，检测线上的电压为低电平。

如果在控制线上输出高电平。那么无论有没有按键按下，检测线都是高电平

如果一行上有不同的按键按下，那么四个检测线上的电平也会有所不同。因此我们可以分辨按键在哪一个列。



但是矩阵按键有多个行：如果控制线R0和R1都输出电平，那么如果检测到C1为低电平，还是无法区分是K1按下还是K5按下。



**扫描法：**

先把R0设置输出低电平，R1设置输出高电平。如果C0-C3上检测到了低电平，说明按键在R0行。通过C0-C3的电平，可以判断是K0-K3中的哪个按键被按下。

然后把R0设置输出高电平，R1设置输出低电平。如果C0-C3上检测到了低电平，说明按键在R1行。通过C0-C3的电平，可以判断是K4-K7中的哪个按键被按下。

如果把R2、R3行画出来，也是以此类推。

扫描法是最常用的方法。

**反转法：**

先把R0、R1、R2、R3都设置为输出、然后输出低电平。

把C0、C1、C2、C3 都设置为输入，上拉。

如果C0-C3上检测到低电平，可以通过C0-C3的电平，判断按键在哪一列 M;

然后把0、C1、C2、C3都设置为输出、然后输出低电平。

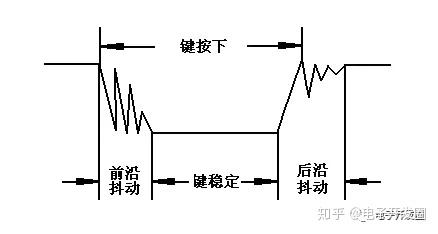
把R0、R1、R2、R3都设置为输入，上拉。

如果R0-R3上检测到低电平，可以通过R0-R3的电平，判断按键在哪一行 N;

这样就可以检测出按键位于N行M列。

**按键的抖动：**

机械按键存在抖动现象，当按下或释放一个键时，往往会出现按键在闭合位置和断开位置之间跳几下才稳定到闭合状态，抖动的持续时间通常不大于10ms。



采用**硬件消抖电路**或**软件延时**方法解决

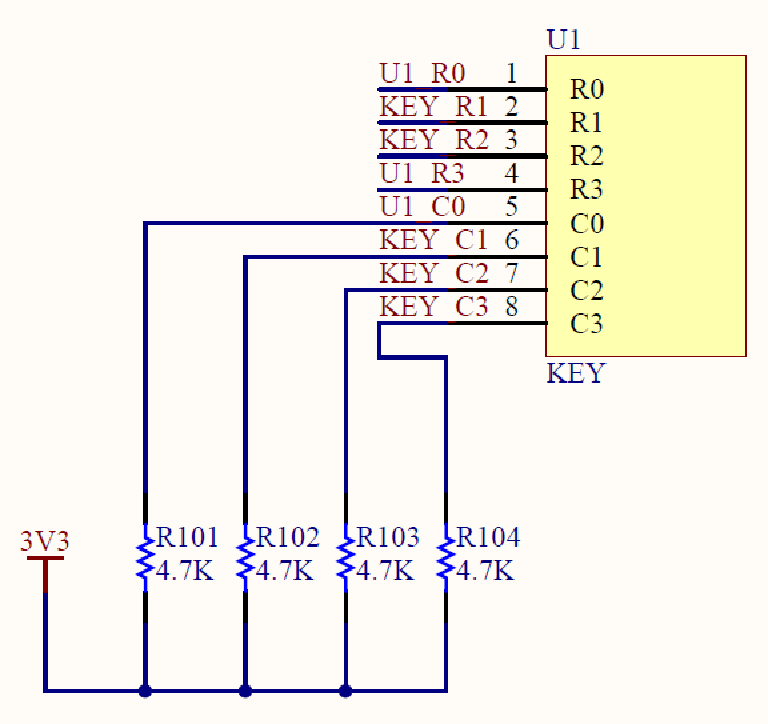
**重键：**两个或多个键同时闭合。出现重键时，读取的键值必然出现一个以上的0，需要根据应用选择合适的办法应对重键。

简单情况：不予识别，认为是错误的按键

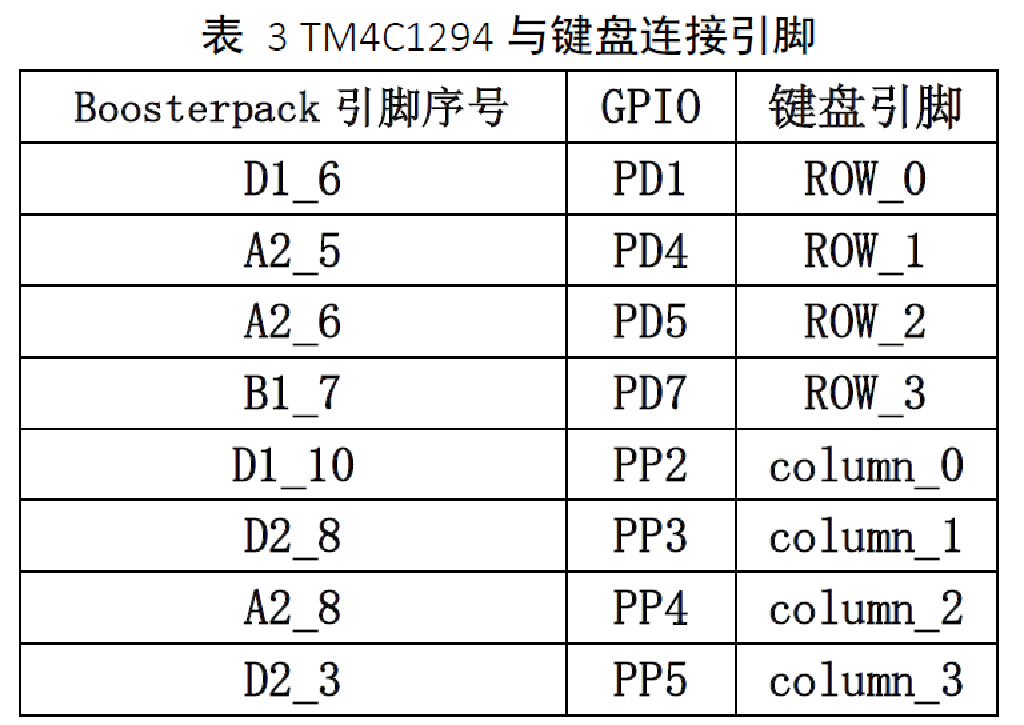
通常情况：只承认先识别出来的键

正常的组合键：都识别出来

实验套件底板上的矩阵键盘：



行悬空，列被上拉，所以行是控制线，列是检测线。行线、列线与开发板引脚的对应关系如下：



所需，需要将PD1、PD4、PD5、PD7设置为输出，将PP2、PP3、PP4、PP5设置为输入。PD7的设置比较特殊，需要先解锁。

**SysCtlPeripheralEnable**(SYSCTL\_PERIPH\_GPIOD);

**SysCtlPeripheralEnable**(SYSCTL\_PERIPH\_GPIOP);

///////解锁PD7//////

HWREG(GPIO\_PORTD\_BASE + GPIO\_O\_LOCK) = GPIO\_LOCK\_KEY;

HWREG(GPIO\_PORTD\_BASE + GPIO\_O\_CR) = 0x80;

**GPIOPinTypeGPIOOutput**(GPIO\_PORTD\_BASE, GPIO\_PIN\_7);

HWREG(GPIO\_PORTD\_BASE + GPIO\_O\_LOCK) = GPIO\_LOCK\_KEY;

HWREG(GPIO\_PORTD\_BASE + GPIO\_O\_CR) = 0x00;

HWREG(GPIO\_PORTD\_BASE + GPIO\_O\_LOCK) = 0;

///////解锁PD7//////

**GPIOPinTypeGPIOOutput**(GPIO\_PORTD\_BASE, GPIO\_PIN\_1|GPIO\_PIN\_4|GPIO\_PIN\_5|GPIO\_PIN\_7);

**GPIOPinTypeGPIOInput**(GPIO\_PORTP\_BASE, GPIO\_PIN\_2|GPIO\_PIN\_3|GPIO\_PIN\_4|GPIO\_PIN\_5); //初始化键盘行、列引脚

底板提供了键盘扫描的示例函数：

为了方便给行线(控制线)赋值，定义一个函数KeyWrite，将输入数据ui8Val的八个位，由低到高，依次赋值给PD1、PD4、PD5、PD7、PP2、PP3、PP4、PP5。

**void**

**KeyWrite**(uint8\_t ui8Val)

{

**GPIOPinWrite**(GPIO\_PORTD\_BASE, GPIO\_PIN\_1, (ui8Val & 0x01)\*2);//将ui8Val最低位写入PD1

**GPIOPinWrite**(GPIO\_PORTD\_BASE, GPIO\_PIN\_4, (ui8Val & 0x02)\*8);//将ui8Val倒数第二位写入PD4

**GPIOPinWrite**(GPIO\_PORTD\_BASE, GPIO\_PIN\_5, (ui8Val & 0x04)\*8);//将ui8Val倒数第三位写入PD5

**GPIOPinWrite**(GPIO\_PORTD\_BASE, GPIO\_PIN\_7, (ui8Val & 0x08)\*16);//将ui8Val倒数第四位写入PD7

**GPIOPinWrite**(GPIO\_PORTP\_BASE, GPIO\_PIN\_2, (ui8Val & 0x10)/4);//将ui8Val倒数低第五位写入PP2

**GPIOPinWrite**(GPIO\_PORTP\_BASE, GPIO\_PIN\_3, (ui8Val & 0x20)/4);//将ui8Val倒数低第六位写入PP3

**GPIOPinWrite**(GPIO\_PORTP\_BASE, GPIO\_PIN\_4, (ui8Val & 0x40)/4);//将ui8Val倒数低第七位写入PP4

**GPIOPinWrite**(GPIO\_PORTP\_BASE, GPIO\_PIN\_5, (ui8Val & 0x80)/4);//将ui8Val最高位写入PP5

}

再定义一个函数KeyRead，将键盘的引脚的数据读出来，返回值的从低位到高位依次是PP5,PP4,PP3,PP2,PD7,PD5,PD4,PD1的电平状态：

uint8\_t **KeyRead**(**void**)

{

uint8\_t temp;

int32\_t temp1,temp2;

temp1=**GPIOPinRead**(GPIO\_PORTD\_BASE, GPIO\_PIN\_1|GPIO\_PIN\_4|GPIO\_PIN\_5|GPIO\_PIN\_7);

temp2=**GPIOPinRead**(GPIO\_PORTP\_BASE, GPIO\_PIN\_2|GPIO\_PIN\_3|GPIO\_PIN\_4|GPIO\_PIN\_5);

temp=(temp1&0x02)/2+(temp1&0x30)/8+(temp1&0x80)/16+(temp2&0x3C)\*4;//temp从最高位到最低位的值分别等于PP5,PP4,PP3,PP2,PD7,PD5,PD4,PD1

**return** temp;

}

定义一个函数check\_key，检测是哪个按键按下：

**void** **check\_key**(**void**)

{

**unsigned** **char** row ,col,tempout,tempin,tmp1,tmp2;

// tmp1用来设置键盘引脚输出值，使列引脚电平低四位中有一个为0

tmp1 = 0x08;

**for**(row=0;row<4;row++)

{

KeyWrite(0x0F);//输出引脚电平低四位为全1

tempout=0x0F-tmp1;

KeyWrite(tempout);//输出引脚电平低四位有一个为0

tmp1 /= 2; // tmp1 右移一位

tempin=KeyRead();

**if** ((tempin & 0xF0) < 0xF0)// 是否键盘引脚高四位有一个为0

{

tmp2 = 0x10; // tmp2用于检测出哪一位为0

**for**(col =0;col<4;col++)// 列检测

{

**if**((tempin & tmp2)==0x00)// 是否是该列

{

key\_val = row\*4 + col; // 获取键值 退出循环

**break**;

}

tmp2 \*= 2; // tmp2左移一位

}

}

}

}

首先，tmp1 = 0x08;，给键盘赋值0x07，即第3行输出低电平，第0、1、2行输出高电平。

KeyWrite(0x0F);//输出引脚电平低四位为全1

tempout=0x0F-tmp1;

KeyWrite(tempout);//输出引脚电平低四位有一个为0，

然后读出键盘的状态tempin=KeyRead();如果有按键按下，那么高四位不是F，(tempin & 0xF0)的值小于0xF0: **if** ((tempin & 0xF0) < 0xF0),如果没有按键，tempin的高四位全是1，(tempin & 0xF0)的值为0xF0。tmp1 /= 2;使tmp1变成了0x04，扫描下一行。

如果有按键按下，进入if程序块，先将tmp2的第4位赋值1，其他位为零。用tempin和tmp2相与，可以检测出PP2是否是电平，如果是，那么按键就位于第row行第col列。计算出key\_val后返回。如果PP2不是低电平，则tmp2左移一位，检测PP3是否是低电平，依次检测PP4和PP5。

第0行的列线都检测完，且没有发现低电平后，tmp1已经变成了0x04，tempout=0x0F-tmp1；KeyWrite(tempout);使第3行输出高电平，第2行输出低电平，第0、1行输出高电平。，检测第2行的四个列线是否有按键按下。

为了消抖，不能直接使用check\_key函数读取按键，还需要加入消抖的环节。

定义key\_event函数：

**void** **key\_event**(**void**)

{

**unsigned** **char** tmp;

KeyWrite(0xF0);// 设置键盘引脚输出值 使得如果有键按下 键盘引脚电平将不为全高

tmp = KeyRead();

**if** ((key\_pressed == 0x00)&&((tmp & 0xF0) < 0xF0))//是否有键按下

{

key\_pressed = 1; // 如果有按键按下，设置key\_pressed标识

**SysCtlDelay**(1000000); //消除抖动

check\_key(); // 调用check\_key(),获取键值

}

**else** **if** ((key\_pressed ==1)&&((tmp & 0xF0) == 0xF0))//是否按键已经释放

{

key\_pressed = 0; // 清除key\_pressed标识

key\_flag = 1;

}

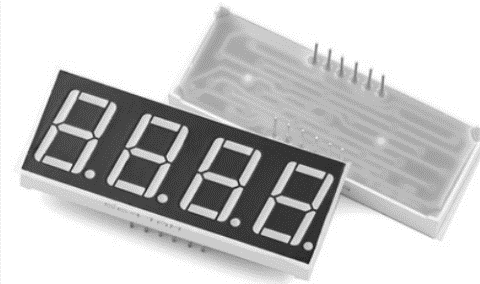
}

先将所有的行线拉低，读取按键状态，如果检测到有按键按下，延迟一段时间再获取按键值。按键释放后，key\_flag=1，表示有按键按下。读取按键后，需要将key\_flag设置为0；需要周期性的调用key\_event函数，检测按键的情况。

示例程序提供的函数，完成了键盘扫描和按键读取的功能，但是程序逻辑杂乱且低效，试图改进键盘扫描程序。

# GPIO模块的应用2 数码管

**数码管：**



发光二极管LED是最简单的显示设备，由7段LED就可以组成的LED数码管，一般还有一个LED代表小数点。

LED数码管广泛用于微机系统，作为显示设备。

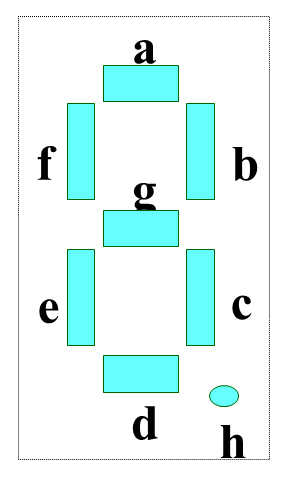
 

**LED数码管的结构:**

7段数码管，主要部分是7段发光LED管，顺时针分别称为a、b、c、d、e、f、g，有的产品还附带有一个小数点h

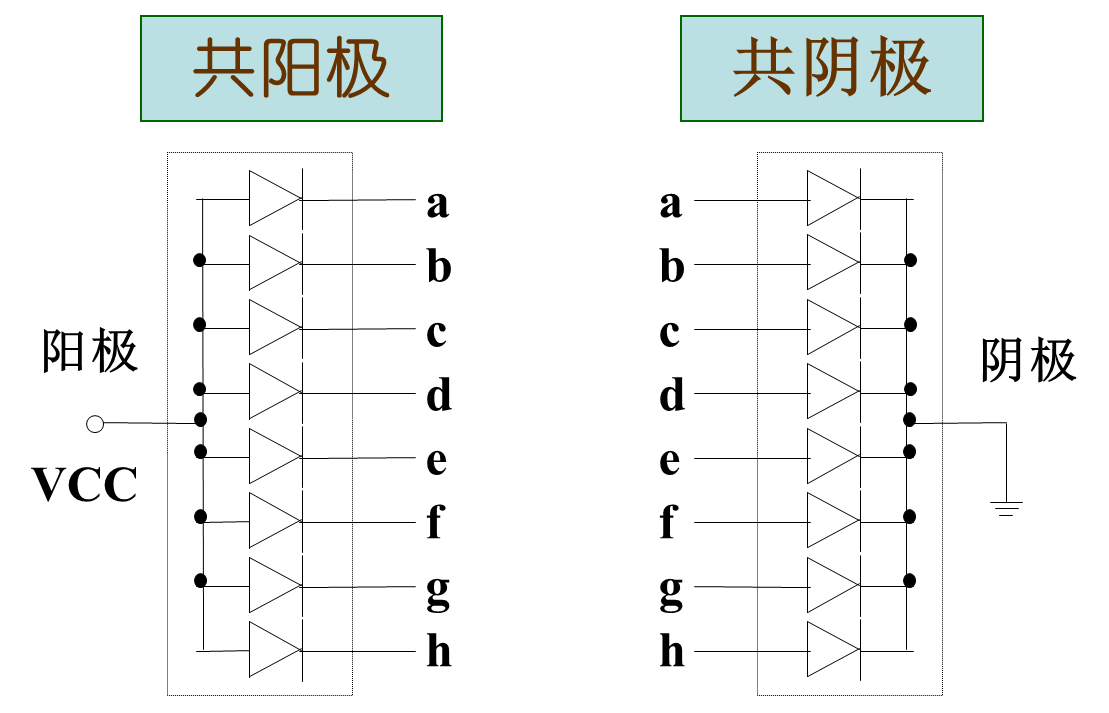
通过7个发光段的不同组合，可以显示：

* 0～9
* A～F，（实现16进制数的显示）
* 个别特殊字符：如－、P 等

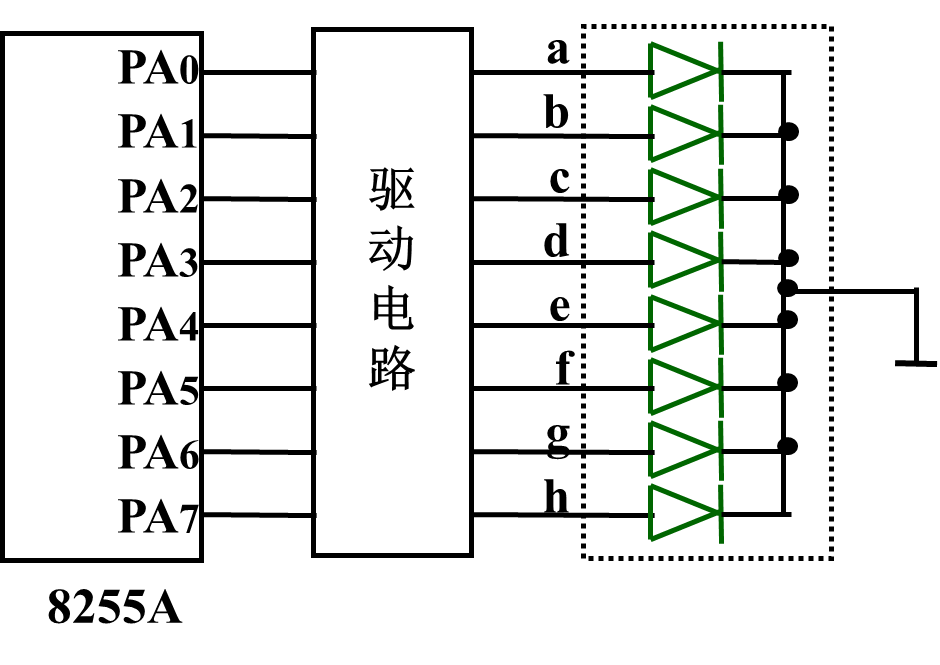
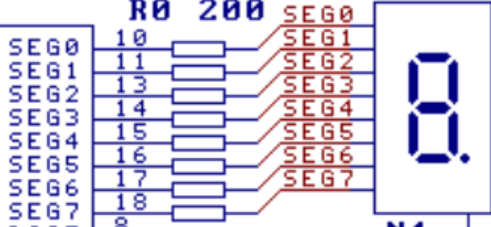


数码管的连接：

* 共阳极
* 共阴极



需要专用的驱动电路，为LED提供电流，也可以用电阻做简单的限流：

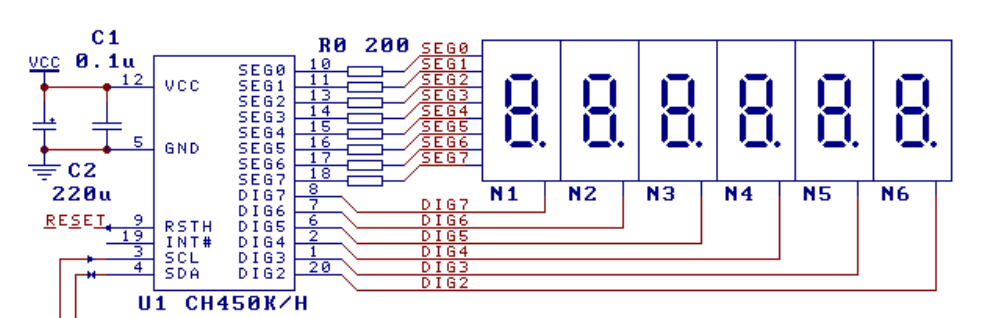
 

数码管的显示：

通过不同的编码，显示不同的数组和字符：



多个LED数码管的显示：



共阴极连接，轮流控制N1到N6的阳极为高电平，每次只显示一个数字。

快速轮流显示N1到N6，就可以利用视觉暂留现象，以为6个数字同时显示了。

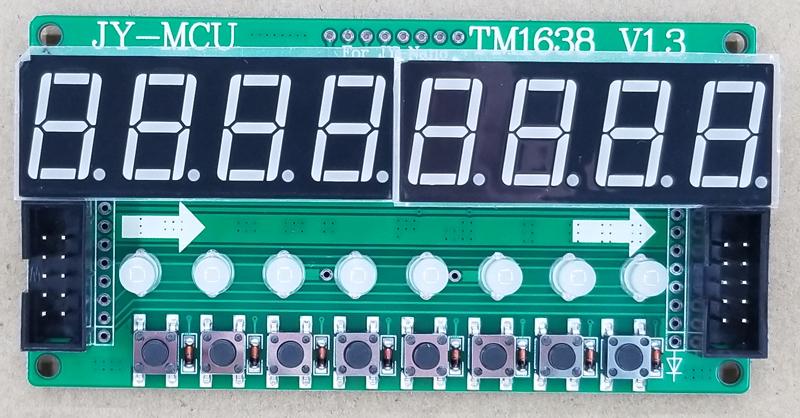
# 矩阵键盘和数码管的应用

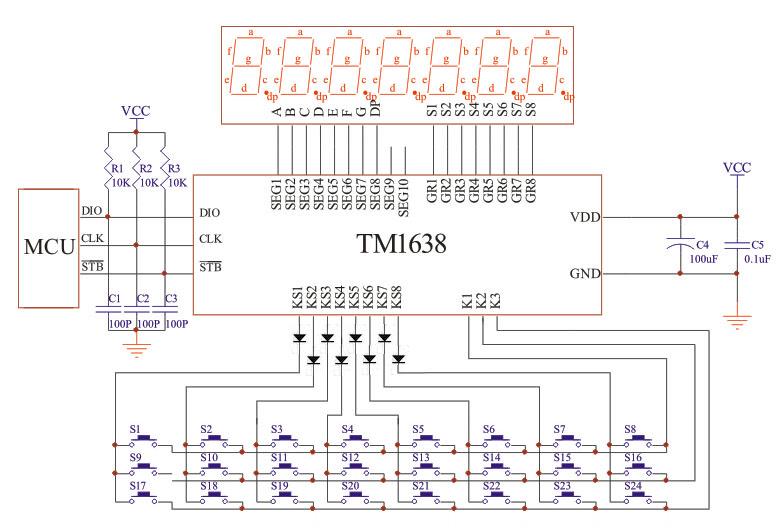
矩阵键盘和数码管，即使使用了扫描功能，依旧占用了过多的引脚。在嵌入式系统中，主控MCU一般不会使用宝贵的引脚资源直接控制矩阵键盘和数码管。

通过异步串行通信端口控制的显示模块，除电源VCC和GND之外，只需要两个引脚,如果使用485总线，可以控制多个设备：

通过专用芯片TM1638控制的数码管和键盘模块，与主控MCU之间，通过IIC总线通信，除电源VCC和GND之外，只需要两个引脚，这两个引脚，可以同时控制一百多个这样的模块，将在后面课程中介绍：





键盘、LED灯专用芯片，可以显示8个数码管，扫描3×8个按键，只需2-3根线与主控MCU相连，极大的节省了MCU的引脚资源。

即使有了专用芯片，也需要了解矩阵键盘和数码管的工作原理。正确连线、快速查错。