第5章

通用输入/输出接口GPIO

本章学习目标

- 了解 GPIO 功能及寄存器操作；
- 了解 S5PV210 芯片的 GPIO 控制器及其应用。

5.1 GPIO 硬件介绍

5.1.1 GPIO 概述

GPIO(General Purpose Input/Output,通用输入/输出接口),通俗点讲就是一些引脚,可以通过它们向外输出高低电平,或者读入引脚的状态,这里的状态也是通过高电平或低电平来反应,所以 GPIO 接口技术可以说是众多接口技术中最为简单的一种。

GPIO 接口具有更低的功率损耗、布线简单、封装尺寸小、控制简单等优点,故其使用非常广泛,在嵌入式系统中占有很大的比重。GPIO 接口通常至少有两个寄存器,即“通用 I/O 控制寄存器”和“通用 I/O 数据寄存器”,数据寄存器的各位直接引到芯片外部供外部设备使用,各位上对应的信号是输入、输出还是其他特殊功能,可以通过设置控制寄存器中的对应位独立地控制。除这两种基本的寄存器外,有时还有上拉寄存器,通过它可以设置 I/O 输出模式是高阻态的,还是带上拉电平输出的,或不带上拉电平输出的。在 S5PV210 中,还引入了驱动强度控制寄存器来调节输出电流的强度,此外,还有功耗控制寄存器用来设置相应引脚的功耗。这些额外增加的寄存器可以使电路设计变得简单,在信号控制上也方便很多。

5.1.2 S5PV210 的 GPIO 寄存器

S5PV210 的 GPIO 寄存器在数量和功能上比之前的 S3C2440 增加了许多,有些 PIN 脚不能作为通常的输入/输出引脚来用,比如不能直接用作 OneNAND 控制信号和数据信号、I2S 接口等。另外,GPIO 接口组寄存器由 4 位来控制,扩展了 GPIO 引脚的功能。所以 S5PV210 的 GPIO 已不仅仅只有 GPIO 的功能,同时向后也是兼容的。本章只讨论常用的 GPIO 相关的知识,对于其他的功能引脚不做特别介绍,在后续章节用到时再做分析。

1. S5PV210 的 GPIO 寄存器总览

GPA0：8 in/out port——带流控的 2×UART；

GPA1：4 in/out port——带流控的 2×UART 或 1×UART；

GPB：8 in/out port——2×SPI 总线接口；

GPC0：5 in/out port——I2S 总线接口、PCM 接口、AC97 接口；

GPC1：5 in/out port——I2S 总线接口、SPDIF 接口、LCD_FRM 接口；

GPD0：4 in/out port——PWM 接口；

GPD1：6 in/out port——3×I2C 接口、PWM 接口、IEM 接口；

GPE0、1：13 in/out port——Camera 接口；

GPF0、1、2、3：30 in/out port——LCD 接口；

GPG0、1、2、3：28 in/out port——4×MMC 通道(通道 0 和 2 支持 4 位、8 位模式，通道 1 和 3 仅支持 4 位模式)；

GPI：低功率 I2S 接口、PCM 接口(不使用 in/out port)，通过 AUDIO_SS PDN 寄存器配置低功耗 PDN；

GPJ0、1、2、3、4：35 in/out port——Modem 接口、CAMIF、CFCON、KEYPAD、SROM ADDR[22:16]；

MP0_1，2，3：20 in/out port——外部总线接口(EBI)信号控制(SROM、NF、OneNAND)；

MP0_4_5_6_7：32 in/out memory port——EBI；

MP1_0～8：71 DRAM1 port(不使用 in/out port)；

MP2_0～8：71 DRAM2 port(不使用 in/out port)；

ETC0、ETC1、ETC2、ETC4：28 in/out ETC port——JTAG、Operating Mode、RESET、CLOCK(ETC3 保留)。

2. 特征介绍

S5PV210 关键特征如下：

- 支持 146 个可控的 GPIO 中断；
- 支持 32 个可控外部中断；
- 支持 237 个多功能输入/输出接口；
- 支持在系统睡眠模式下引脚可控(除 GPH0、GPH1、GPH2 和 GPH3)。

3. S5PV210 的 GPIO 功能介绍

S5PV210 的 GPIO 包含两部分，即带电部分(alive-part)和不带电部分(off-part)，对于 alive-part 模式下的 GPIO 寄存器，在睡眠模式时提供电源，所以寄存器中的值不会丢失；在 off-part 模式下则不同。S5PV210 的功能如图 5-1 所示。

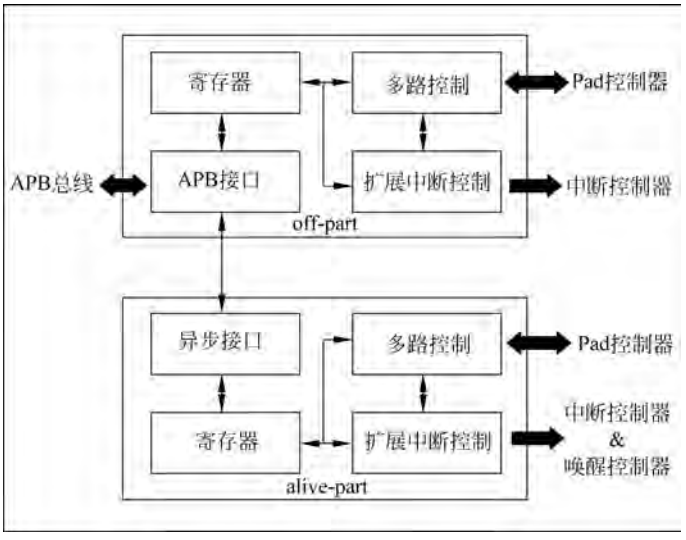

图 5-1　GPIO 功能模块图

5.1.3 实验用到的寄存器详解

S5PV210 的 GPIO 寄存器非常多，每个接口组有两种类型的控制寄存器，一种工作在正常模式，另一种工作在掉电模式(STOP、DEEP-STOP、SLEEP mode)，下面只针对本章实验用到的 GPIO 接口 GPC0 进行介绍，其他的 GPIO 接口用法可以依葫芦画瓢。GPC0 的控制寄存器有 GPC0CON、GPC0DAT 、GPC0PUD、GPC0DRV、GPC0CONPDN 和 GPC0PUDPDN，前面 4 类工作在正常模式，后面 2 类工作在掉电模式。

1) GPC0CON 寄存器

此寄存器为 GPC0 引脚的控制寄存器，主要用途是配置各引脚的功能，此引脚对应的内存地址是 0xE020_0060。表 5-1 是 S5PV210 手册里关于 GPC0_3、GPC0_4 引脚的配置信息。

表 5-1 GPC0 控制寄存器

GPC0CON	位	描 述	初 始 状 态
GPC0CON[4]	[19:16]	0000 =输入 0001 =输出 0010 = I2S_1_SDO 0011 = PCM_1_SOUT 0100 = AC97SDO 0101～1110 =保留 1111 = GPC0_INIT[4]中断	0000
GPC0CON[3]	[15:12]	0000 =输入 0001 =输出 0010 = I2S_1_SDI 0011 = PCM_1_SIN 0100 = AC97SDI 0101～1110 =保留 1111 = GPC0_INIT[3]中断	0000

从表中可以看出，每 4 位控制一个引脚(GPC0_3 或 GPC0_4)，当值为 0b0000 时引脚被设为输入功能，当值为 0b0001 时设为输出功能，当值为 0b0010～0b0100 时引脚设为特殊功能引脚，当值为 0b1111 时设为中断引脚，0b0101～0b1110 保留未使用。

2) GPC0DAT 寄存器

此引脚对应的内存地址是 0xE020_0064，该寄存器决定了引脚的输入或输出电平的状态，当引脚设为输入时，通过读寄存器可知对应引脚电平状态是高还是低。当引脚设为输出时，写寄存器对应位可使引脚输出高电平或低电平；当引脚被设为功能引脚时，如果读寄存器对应引脚的值是不确定的。实验使用的引脚是 GPC0DAT[4:3]。

3) GPC0PUD 寄存器

此引脚对应的内存地址是 0xE020_0068，使用两位来控制 1 个引脚。当值为 0b00 时，对应引脚无上拉/下拉电阻；当值为 0b01 时，有内部下拉电阻；当值为 0b10，有内部上拉电阻；当值为 0b11 时为保留。

上拉/下拉电阻的作用是当 GPIO 引脚处于高阻态(既不是输出高电平，也不是输出低电平，相当于没有接入)时，它的电平状态由上拉电阻或下拉电阻决定。

4) GPC0DRV 寄存器

此引脚对应的内存地址是 0xE020_006C，该寄存器为接口组驱动能力控制寄存器，主要

用于调节引脚的电流强度，S5PV210 给出了 4 种强度，由 2 位控制，数值越大强度越大。通常对于高速信号或较弱的周边装置，可调大对应引脚的强度值，在实际使用中需要合理使用该寄存器，强度越大电流消耗也越大。

5）GPC0CONPDN 寄存器

此引脚对应的内存地址是 0xE020_0070，用 2 位来控制引脚的功能。当值为 0b00 时，引脚输出低电平；当值为 0b01 时，输出高电平；当值为 0b10 时，对应引脚被设为输入；当值为 0b11 时，引脚保持原来的状态。

6）GPC0PUDPDN 寄存器

此引脚对应的内存地址是 0xE020_0074，用 2 位来控制引脚。当值设为 0b00 时，无上拉/下拉电阻；当值为 0b01 时，有下拉电阻；当值为 0b10 时，有上拉电阻；当值为 0b11 时保留。GPCOPUDPDN 寄存器的功能与 GPCOPUD 类似。

5.2　S5PV210 的 GPIO 应用实例


视频讲解

5.2.1　GPIO 实验

1. 实验目的

利用 S5PV210 的 GPC0_3、GPC0_4 这两个 GPIO 引脚控制 2 个 LED 发光二极管，分别用汇编语言和 C 语言实现。

2. 实验原理

如图 5-2 所示，LED1、LED2 分别与 GPC0_3、GPC0_4 相连，中间接两个 NPN 型三极管。当 GPIO 引脚输出高电平时三极管与地(GND)导通，LED 灯亮；反之，输出低电平，LED 灯就灭。这里的三极管有点像“开关”，控制着 LED 的亮灭。

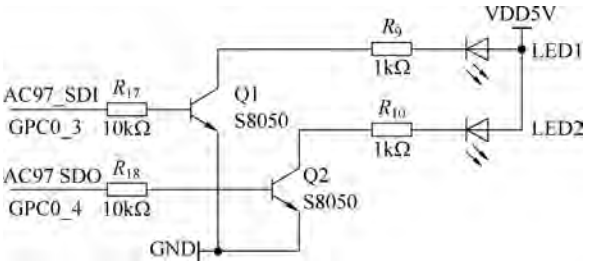

图 5-2　GPIO 功能模块图

注：从本章开始所有的实验都基于 TQ210 开发板设计与测试，但是原理适用于所有 ARM 架构学习板，读者有兴趣可以将本书所有实验移植到其他学习板上，做到学以致用。

5.2.2　程序设计与代码详解


视频讲解

1. 用汇编语言实现点亮 LED1

在上一章我们对 S5PV210 的启动过程以及 SD 启动卡的制作都进行了详细的介绍，下面主要分析程序设计部分，源代码存放在/opt/hardware/ch6/led_on，汇编语言操控 GPIO 的代码如下所示，相关代码在 led_on.S 文件中。

```
01 .text
02 .global _start                /* 声明一个全局的标号 */
03 _start:
04  ldr  r0,   = 0xE0200060   /* GPC0CON 寄存器的地址为 0xE0200060 */
```

```
05  ldr r1, [r0]              /* 读出 GPC0CON 寄存器原来的值 */
06  bic r1, r1, #0xf000       /* bit[15:12]清零 */
07  orr  r1,  r1,  #0x1000 /* 设置 GPC0_3[15:12] = 0b0001 */
08  str  r1,  [r0]            /* 写入 GPC0CON,配置 GPC0_3 为输出引脚 */
09  ldr  r0,  = 0xE0200064  /* GPC0DAT 寄存器的地址为 0xE0200064 */
10  ldr  r1,  [r0]            /* 读出 GPC0DAT 寄存器原来的值 */
11  bic  r1,  r1,  #0x8     /* bit[3]清零 */
12  orr  r1,  r1,  #0x8     /* bit[3] = 1 */
13  str  r1,  [r0]            /* 写入 GPC0DAT,GPC0_3 输出高电平 */
14 halt_loop:
15   b  halt_loop             /* 死循环,不让程序跑飞 */
```

整个代码量很少也很简单，这里有两个地方需要注意：

（1）在 S3C2440 等平台上，我们都要先关看门狗，如果是用 C 语言实现，还要设置栈。而在 S5PV210 中，这些动作都已在厂家固化的代码 BL0 里做好了，如对此不清楚，可以再回顾第 4 章的内容。

（2）对寄存器位的修改，通常都是先读出寄存器，然后修改对应位，再写回寄存器，其他位保持不变，这样做的目的是不改变其他引脚状态。

下面是编译用到的 Makefile 文件：

```
01 led_on.bin:led_on.S
02  arm-linux-gcc -c -o led_on.o led_on.S
03  arm-linux-ld -Ttext 0xD0020010 led_on.o -o led_on_elf
04  arm-linux-objcopy -O binary -S led_on_elf led_on.bin
05  arm-linux-objdump -D led_on_elf > led_on.dis
06 clean:
07  rm -f led_on.bin led_on_elf *.o
```

这个 Makefile 文件可以说是一个最基本的 Makefile 语法格式（目标-依赖-命令），各编译工具及其参数在配套资源补充资料第 1 章中有介绍。0xD002_0010 为程序的链接地址（即运行地址，也是 BL1 的有效地址）。这里要再次提醒的是，命令前面一定要有一个 Tab 制表符，这是 Makefile 语法上的规定，否则 make 工具就不认识了！

接下来就可以用 FTP 将所有代码上传到宿主机（Ubuntu 系统）编译，编译非常简单，只要在代码所在的目录下执行 make 命令即可，最终生成二进制格式的 bin 文件。

```
$ cd /opt/examples/ch6/led_on
$ make
```

最后将生成好的 bin 文件按第 4 章介绍的步骤烧写到 SD 卡中，将目标板拨到 SD 卡启动，插入 SD 卡上电即可看到 LED1 灯被点亮。

关于本书所使用的 TQ210 开发板的 SD 卡启动方式和 NAND 启动方式，对应拨码开关的设置如图 5-3 所示。

2. 用 C 语言实现循环点亮两个 LED

用 C 语言实现，其汇编代码就变得更加简单了，只需一个跳转语句。下面分三步来实现 C 语言。

循环点亮两个 LED 灯，代码在/opt/hardware/ch6/led_on_c。

1）启动代码 start.S

```
01 .text
02 .global _start             /* 声明一个全局的标号 */
```

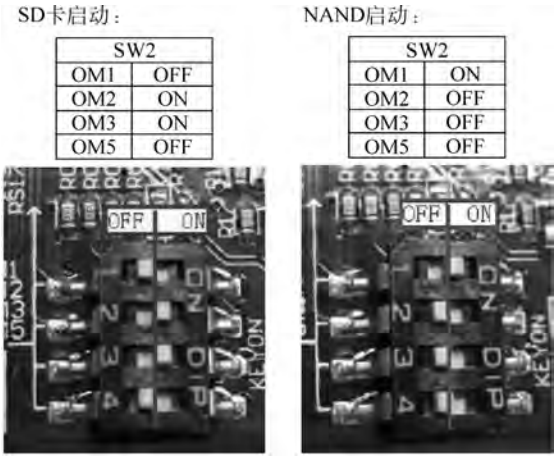

图 5-3　TQ210 开发板启动方式拨码开关设置

```
03 _start:
04   bl main                    /* 跳转到 C 函数中执行 */
05
06 halt_loop:
07   b  halt_loop               /* 死循环,不让程序跑飞 */
```

2）循环点亮 LED 灯

```
01 #define GPC0CON     *((volatile unsigned int *)0xE0200060)
02 #define GPC0DAT     *((volatile unsigned int *)0xE0200064)
03
04 #define  GPC0_3_out  (1<<(3*4))
05 #define  GPC0_4_out  (1<<(4*4))
06
07 #define  GPC0_3_MASK  (0xF<<(3*4))
08 #define  GPC0_4_MASK  (0xF<<(4*4))
09
10 void delay(volatile unsigned long dly)
11 {
12   volatile unsigned int t = 0xFFFF;
13   while (dly--)              //Cortex-A8 默认时钟频率都达到了 667MHz
13     for(; t>0; t--);         //循环次数必须设大一点,否则看不出闪烁效果
14 }
15
16 int main(void)
17 {
18   unsigned long i = (1<<3);
19   GPC0CON &= ~(GPC0_3_MASK | GPC0_4_MASK);     //清 bit[15:12]和 bit[19:16]
20   GPC0CON |= (GPC0_3_out | GPC0_4_out);        //配置 GPC0_3 和 GPC0_4 为输出引脚
21
22   while (1)
23   {
24     delay(0x50000);                            //延时
25     GPC0DAT &= ~(0x3<<3);                      //LED1 和 LED2 熄灭
26     if (i == 0x08)
27       i = (1<<4);
28     else
29       i = (1<<3);
```

```
30      GPC0DAT | = i;                                  //循环点亮 LED 灯
31    }
32    return 0;
33 }
```

上述代码的功能实现比汇编稍复杂，这里是循环点亮了两个 LED，不过基本原理是一样的，都是配置寄存器和往寄存器写入数值。在操作方法上，C 语言对寄存器的操作是通过宏函数实现的，这个宏就代表了寄存器的地址，往这个地址中写入数据就可以配置 GPIO 引脚的功能。下面通过一个 C 语言指针的例子来理解宏定义的功能。

```
int a;                                                  //定义一个整型变量 a
int * p;                                                //定义一个整型指针变量 p
p   =   &a;                                             //指针指向变量 a
* p = 6                                                 //变量 a 的值等于 6
```

上面几行代码是 C 语言指针最基本的用法，通常可以把指针看作一个地址，比如这里指针 p 就等于变量 a 的地址，假设变量 a 的地址为 0x12345678，对指针 p 操作就相当于操作地址：

```
int * p;
p = 0x12345678;
```

为方便理解，实际 p = (int *)0x12345678，这里的(int *)是将地址转换为 int 类型。

* p 表示地址中的内容，比如上面的 6，换句话说，变量 a 的值与 * p 是同一个值，现在修改地址 0x12345678 的内容为 8，只需要如下操作即可：

```
* p = 8;                                                //与 a = 8 是等价的
```

也就相当于：

```
* (0x12345678) = 8;或者严格写成 * ((int * )0x12345678) = 8;
```

上面这样写看上去有点别扭，可以定义一个宏来表示：

```
#define A * ((int * )0x12345678)
A = 8;
```

分析到这里，再看看程序中定义的宏与这里的宏是不是很相似了。这里还有一个关键字 volatile 需要说明下，它的本意是“易变的”，由于访问寄存器要比访问内存单元快得多，所以编译器一般都会进行减少存取内存的优化，直接从缓存 cache 中取数据这样会有一个问题，即可能会读取到“脏”数据（数据被其他程序代码修改）。当用 volatile 声明的时候，其实就是告诉编译器对访问该地址的代码不要优化了，使用的时候系统总是从它的内存读取数据，以保证不“脏”。

3）编写 Makefile

```
01 objs := start.o leds.o
02
03 leds.bin: $(objs)
04   arm-linux-ld -Ttext 0xD0020010 -o leds.elf $^
05   arm-linux-objcopy -O binary -S leds.elf $@
06   arm-linux-objdump -D leds.elf > leds.dis
07
08 %.o: %.c
09   arm-linux-gcc -c -O2 $< -o $@
```

```
10 %.o : %.S
11  arm-linux-gcc -c -O2 $< -o $@
12 clean:
13  rm -f *.o *.elf *.bin *.dis
```

这里的 Makefile 文件看似比上一个 Makefile 要复杂，主要是引入了一些变量的用法，目的是加深对 Makefile 的了解，更多 Makefile 知识见配套资源补充资料第 1 章中的内容。