GPIO 输出驱动指示灯

1. 实验目的

- 掌握 STC15W4K32S4 系列单片机 GPIO 口四种工作模式。
- 掌握 LED 驱动电路的设计：控制方式、限流电阻的计算和需要考虑的因素。
- 掌握对单个和多个指示灯的驱动程序设计及程序控制方式。

2. 实验内容

- 编写程序控制单个指示灯闪烁。
- 编写程序实现流水灯功能。

3. 硬件电路设计

3.1. 开发板指示灯硬件电路

LED(Light Emitting Diode)是发光二极管的简称，在很多设备上常用它来做为一种简单的人机接口，如网卡、路由器等通过 LED 向用户指示设备的不同工作状态。所以，我们习惯把这种用于指示状态的 LED 称为 LED 指示灯。

进取者 STC15 开发板上设计了 2 个 LED 指示灯，我们可以通过编程驱动 LED 指示灯点亮、熄灭、闪烁，从而达到状态指示的目的，LED 指示灯驱动电路如下图所示。

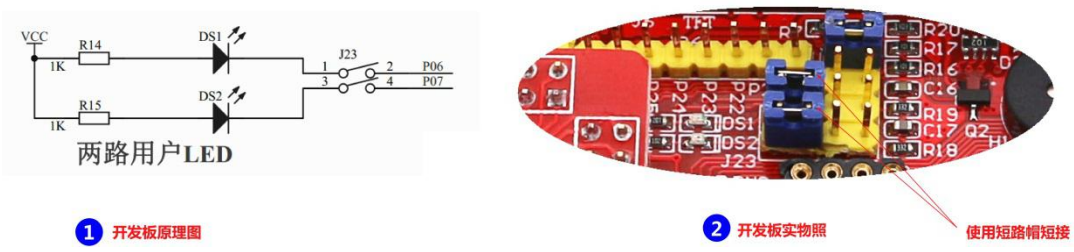


图 1：LED 指示灯驱动电路

✧ 2 个 LED 指示灯占用的单片机的引脚如下表：

表 1：LED 引脚分配

LED	颜色	引脚	说明
DS1	蓝色	P0.6	非独立 GPIO
DS2	红色	P0.7	非独立 GPIO

✧ 注：独立 GPIO 表示开发板没有其他的电路使用这个 GPIO，非独立 GPIO 说明开发板有其他电路用到了该 GPIO。

LED 指示灯驱动电路是一个很常见、简单的电路，但它也是一个典型的单元电路，对于初学者来说，类似常用的典型电路必须要掌握，不但要知其然、还要知其所以然。

接下来，我们来分析一下这个简单的 LED 指示灯驱动电路。

LED 驱动电路设计的时候，要考虑两个方面：控制方式和限流电阻的选取。

3.2. 控制方式

LED 指示灯控制方式分为高电平有效和低电平有效两种，高电平有效是单片机 GPIO 输出高电平时点亮 LED，低电平有效是单片机 GPIO 输出低电平时点亮 LED。

3.2.1. 低电平有效的控制方式

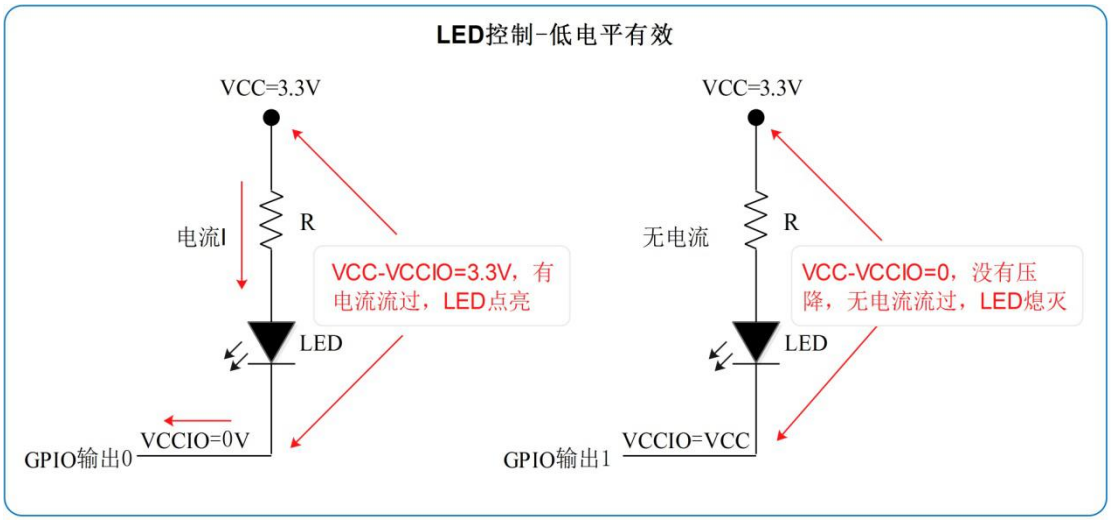


图 2：LED 控制-低电平有效原理

低电平有效控制方式中，当单片机的 GPIO 输出低电平(逻辑 0)的时候，LED 和电阻 R 上的压降等于（VCC-VCCIO = 3.3V），这时候，因为存在压降，同时，这个电路是闭合回路，这就达到了电流产生的两个要素，LED 上会有电流流过，LED 被点亮。

当单片机的 GPIO 输出高电平(逻辑 1)的时候，LED 和电阻 R 上的压降等于（VCC-VCCIO = 0V），这时候，因为 LED 上没有压降，当然不会有电流流过，所以 LED 熄灭。

3.2.2. 高电平有效的控制方式

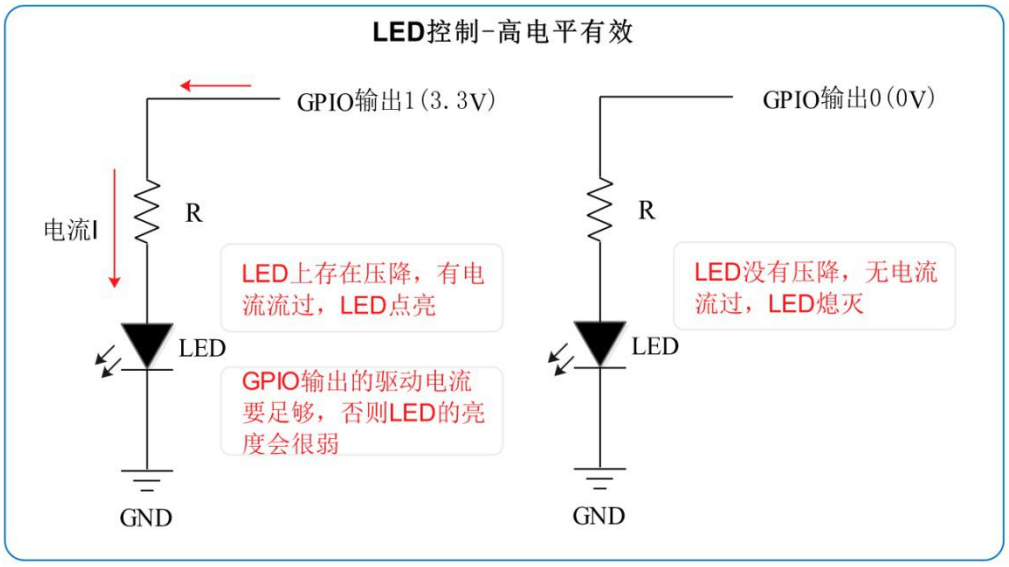


图 3：LED 控制-高电平有效原理

高电平有效控制方式中，由单片机的 GPIO 输出电流驱动 LED，当单片机的 GPIO 输出高电平(逻辑 1)的时候，LED 上存在压降，因为电路是闭合回路，所以会有电流流过，这时 LED 被点亮，但要注意，单片机的 GPIO 要能提供足够的输出电流，否则，电流过小，会导致 LED 亮度很弱。

当单片机的 GPIO 输出低电平(逻辑 0)的时候，LED 和电阻 R 上的压降等于 0V，这时候，LED 上没有电流流过，LED 熄灭。

3.2.3. 选择哪种方式来控制 LED

绝大多数情况下，我们会选择使用低电平有效的控制方式，如艾克姆科技进取者 STC15 开发板中的 LED 指示灯就是低电平有效。这样设计指示灯驱动电路的好处是：

- 单片机 GPIO 口低电平时的灌入电流一般比高电平时的拉电流要大，能提供足够的电流驱动 LED。
- 单片机上电或复位启动时，GPIO 口一般都是高阻输入，用低电平有效的控制方式可以确保 LED 在上电或复位启动时处于熄灭状态。

✧ 注：当单片机是 5V 单片机时选择指示灯控制方式的原理是一样的。

3.3. LED 限流电阻的选取

3.3.1. 限流电阻的计算

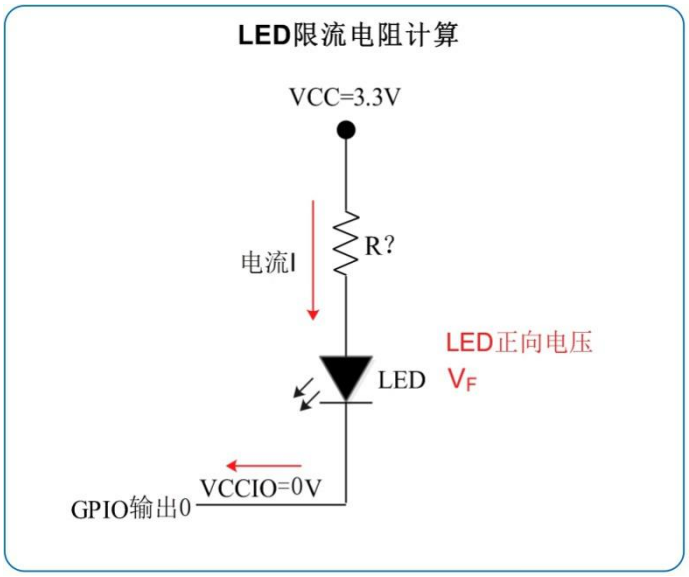


图 4：LED 限流电阻计算

由上图可以看出，LED 限流电阻的计算公式如下：

$$R = \frac{VCC - V_F}{I}\ \Omega$$

其中，VCC=3.3V，V_F 是 LED 的正向压降，LED 的数据手册都会给出正向电流为 20mA 时测试的 V_F 的范围，下图是一款 0603 LED 的实物图和参数。在参数表中可以看到正向电流为 20 mA 时 V_F 最小值是 1.6V，典型值是 2.0V，最大值是 2.6V。

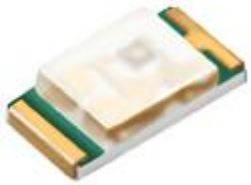


图 5：封装 0603 的 LED

参数名称 Parameter	符号 Symbol	条件 Condition	最小值 Min.	典型值 Typ.	最大值 Max.	单位 Unit
反向电流 Reverse Current	I_R	V_R=5V	-	-	10	μA
视角度 View Angle	$2\theta_{1/2}$	-	-	130	-	deg.
正向电压 Forward Voltage	V_F	电流为20mA时测试的V_F范围	1.6	2.0	2.6	V
峰值波长 Peak Wavelength	λ_P			630		nm
主波长 Dominant Wavelength	λ_d	I_F=20mA	615	622	630	nm
半波宽度 Spectrum Radiation Bandwidth	$\Delta\lambda$		-	15	-	nm
光强 Luminous Intensity	I_V		80	120	220	mcd

图 6：0603 LED 参数图

计算时 V_F 的值可以用典型值来进行估算，对于电流，需要根据经验值和对 LED 亮度的要求相结合来确定，一般经验值是(2~5)mA，不过要注意，只要亮度符合自己的要求，电流低于 2mA 也没有任何问题。

电流为 2mA 时限流电阻值计算如下：

$$R = \frac{3.3 - 2.0}{0.002}\Omega = 650\Omega$$

3.3.2. 限流电阻的选择

根据上一节对限流电阻计算公式的描述及对选择的封装 0603 的指示灯参数的了解，计算出限流电阻的理论值是 650Ω，但 650Ω不是一个常用的电阻值，所以我们需要选择一个电阻值为 650Ω左右的常用的电阻器。

在进取者 STC15 开发板上，我们选择的限流电阻的电阻值是最常用的 1K 的电阻器，选择限流电阻后，还需要实际测试观察 LED 的亮度是否符合自己的需求，经过实际测试观察，进取者 STC15 开发板上使用 1K 限流电阻时亮度符合我们的要求，由此，限流电阻的阻值确定为 1K。当然，如果实际项目中觉得亮度不够，可以将电阻值适当减小一些，如使用 680Ω或 510Ω的电阻器作为限流电阻。

还有一点需要强调的是，不同颜色的指示灯在即使同一亮度时所需的限流电阻不一定是相同的。这也是为什么有些产品的面板上有不同颜色的指示灯，各指示灯所使用的限流电阻不一样的原因。

3.4. STC15W4K32S4 系列单片机 GPIO 口

STC15W4K32S4 系列单片机 GPIO 口数量取决于芯片引脚的个数，芯片引脚个数和芯片封装模切相关。正常情况下，GPIO 口数量是所选择单片机引脚个数减去 2，因为单片机需要 2 个引脚作为供电引脚（电源正 VCC、电源负 GND）。

3.4.1. 芯片封装

封装，Package，是把集成电路装配为芯片最终产品的过程，简单地说，就是把 Foundry 生产出来的集成电路裸片（Die）放在一块起到承载作用的基板上，把管脚引出来，然后固定包装成为一个整体。

STC15W4K32S4 系列有多种封装。按芯片引脚个数来分类，如下：

- 28引脚：SOP28和SKDIP28。
- 32引脚：LQFP32。
- 40引脚：DIP40。
- 44引脚：LQFP44。
- 48引脚：LQFP48和QFN48 。
- 64引脚：LQFP64S、LQFP64L和QFN64 。

✧ 注：单片机每个 GPIO 口驱动能力（输出模式时）均可达到 20mA，但引脚数≥40 的单片机整个芯片最大不要超过 120mA，引脚数≤32 的单片机整个芯片不要超过 90mA。进取者 STC15 开发板采用的芯片是 LQFP48 封装的，GPIO 数量是 46 个。

3.4.2. GPIO 口工作模式

STC15W4K32S4 系列单片机所有 GPIO 口均有 4 种工作模式：准双向口/弱上拉（标准 8051 输出口模式）、推挽输出/强上拉、高阻输入（电流既不能流入也不能流出）、开漏输出。下面针对内部结构图进行分析。

■ 准双向口/弱上拉模式：

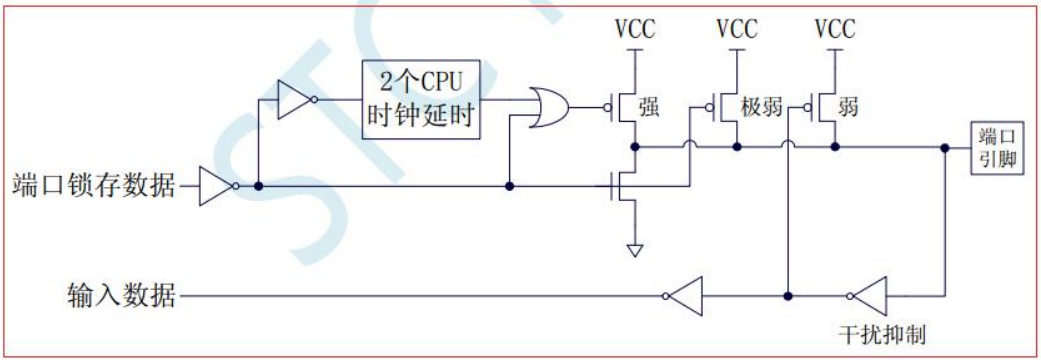
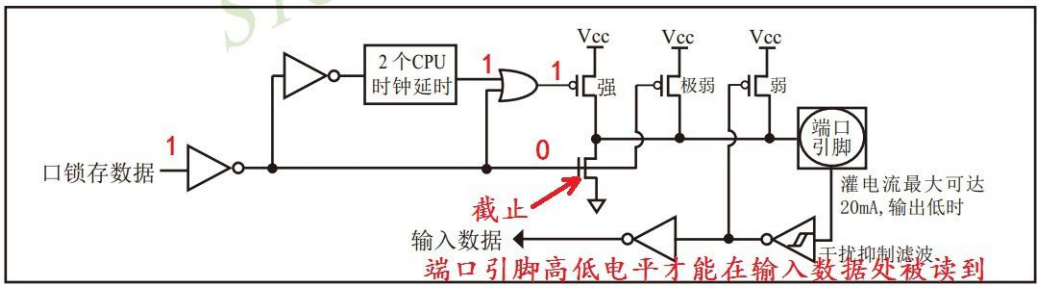


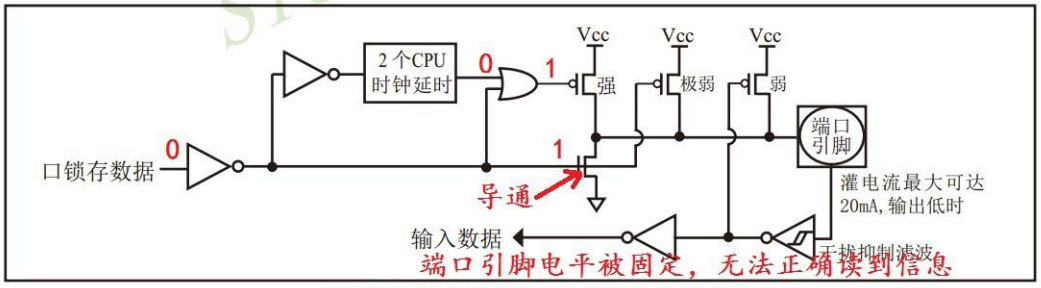
图 7：GPIO 准双向口/弱上拉模式内部框图

1） 准双向口（弱上拉） 输出类型可用作输出和输入功能而不需要重新配置端口输出状态。这是因为准双向口有 3 个上拉晶体管可适应输入输出不同的需要。

2） 手册中有这样一句话：准双向口（弱上拉）在读外部状态前，要先锁存为‘1’，才可读到外部正确的状态。下图分别就锁存数据为‘1’和‘0’时进行了分析。



锁存数据为1时，准双向口内部结构分析



锁存数据为0时，准双向口内部结构分析

图 8：GPIO 准双向口/弱上拉模式内部框图分析

3） 由上图分析可知，准双向口（弱上拉）在读外部状态前，如果锁存为‘0’，则 GPIO 引脚状态被固定，无法读到外部正确的状态。

■ 推挽输出/强上拉模式：

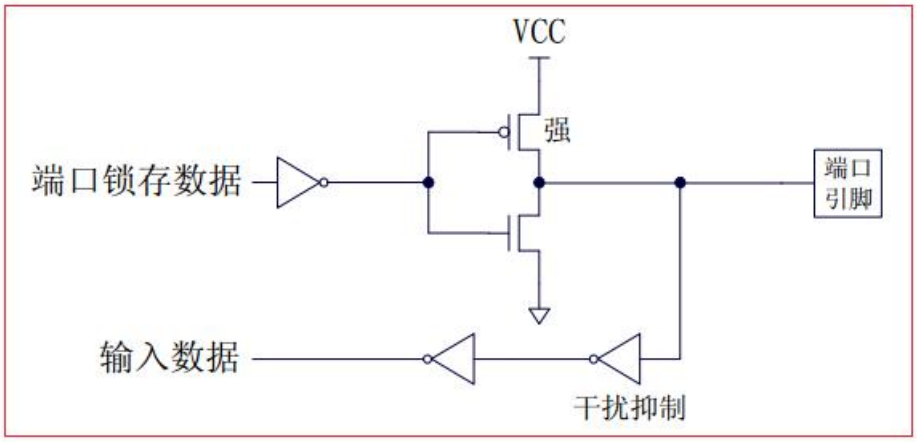


图 9：GPIO 推挽输出/强上拉模式内部框图

1） 强推挽输出配置的下拉结构与开漏输出以及准双向口的下拉结构相同，但当锁存器为 1 时可提供持续的强上拉。所以，推挽输出一般用于需要更大驱动电流的情况。

2） 在控制 LED 时，如果采用的是高电平有效的控制方式，则控制 LED 的单片机 GPIO 口必须配置成推挽输出/强上拉模式方可。

■ 高阻输入模式：

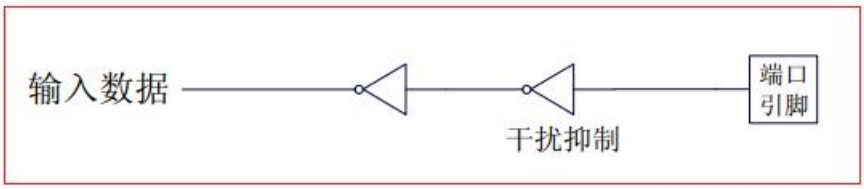


图 10：GPIO 高阻输入模式内部框图

1） 因带有一个施密特触发输入以及一个干扰抑制电路，GPIO 配置为高阻输入时，电流既不能流入也不能流出 GPIO 口。

2） 我们下载程序时常看到这样一段话“注意: STC15W4K32S4 系列的芯片，上电后所有与 PWM 相关的 IO 口均为高阻态，需将这些口设置为准双向口或强推挽模式方可正常使用”。这里说的高阻态即是 GPIO 口被设置了高阻输入模式。

■ 开漏输出模式：

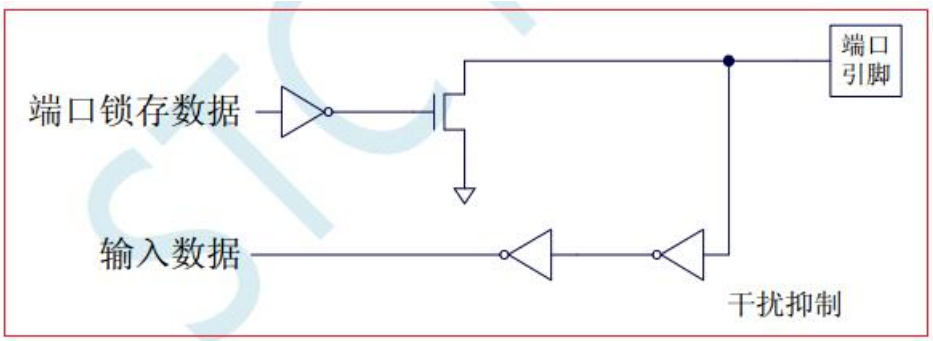


图 11：GPIO 开漏输出模式内部框图

1） 开漏模式既可以读外部状态也可以对外输出高电平或低电平。

2） 如果要正确读外部状态或需要对外输出高电平时，需外加上拉电阻。

4. 软件设计

4.1. IO 寄存器汇集

STC15W4K32S4 系列单片机提供了 24 个用于操作 IO 的寄存器，如下表所示：

表 2： IO 相关寄存器

序号	寄存器名	读/写	功能描述
1	P0	读/写	P0 端口数据寄存器。
2	P1	读/写	P1 端口数据寄存器。

3	P2	读/写	P2 端口数据寄存器。
4	P3	读/写	P3 端口数据寄存器。
5	P4	读/写	P4 端口数据寄存器。
6	P5	读/写	P5 端口数据寄存器。
7	P6	读/写	P6 端口数据寄存器。
8	P7	读/写	P7 端口数据寄存器。
9	P0M0	可写	P0 端口配置寄存器 0。
10	P0M1	可写	P0 端口配置寄存器 1。
11	P1M0	可写	P1 端口配置寄存器 0。
12	P1M1	可写	P1 端口配置寄存器 1。
13	P2M0	可写	P2 端口配置寄存器 0。
14	P2M1	可写	P2 端口配置寄存器 1。
15	P3M0	可写	P3 端口配置寄存器 0。
16	P3M1	可写	P3 端口配置寄存器 1。
17	P4M0	可写	P4 端口配置寄存器 0。
18	P4M1	可写	P4 端口配置寄存器 1。
19	P5M0	可写	P5 端口配置寄存器 0。
20	P5M1	可写	P5 端口配置寄存器 1。
21	P6M0	可写	P6 端口配置寄存器 0。
22	P6M1	可写	P6 端口配置寄存器 1。
23	P7M0	可写	P7 端口配置寄存器 0。
24	P7M1	可写	P7 端口配置寄存器 1。

✧ 注：选择的单片机封装不同，具有的端口不同。比如 LQFP48 引脚单片机没有 P6 和 P7 端口，在程序设计时操作 P6 和 P7 端口对应的寄存器是没有意义的。

4.2. 寄存器解析

首先普及一个常用知识点：为什么说 STC15W4K32S4 系列单片机是 8 位单片机呢？这个 8 位指的是什么？

一般来说某个单片机或微处理器是几位，指的是“机器字长”。每个单片机或微处理器

最基本的功能是算术逻辑运算，而算术逻辑运算的主要部件是“算术逻辑单元（ALU）”。机器字长即是指 ALU 的数据位宽，也就是指令能直接处理的二进制位数。

通常单片机或微处理器的寄存器的位宽等于 ALU 的位宽，所以一般可通过识别单片机或微处理器寄存器的位宽来确定该单片机或微处理器是多少位的。我们所接触的 STC 的单片机，其寄存器都是 8 位的，所以 STC 的单片机都是 8 位的单片机。以后学习 STM32F103 系列的微处理器，其寄存器是 32 位的，所以会说 STM32F103 系列微处理器是 32 位的。

4.2.1. 端口数据寄存器

下图是对端口数据寄存器 P0、P1、P2、P3、P4、P5、P6、P7 的描述，端口数据寄存器各位代表对应端口的 IO 口，比如，P0 端口数据寄存器 B0 位代表 P0.0 口，B7 位代表 P0.7 口。

符号	地址	B7	B6	B5	B4	B3	B2	B1	B0
P0	80H	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0
P1	90H	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0
P2	A0H	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0
P3	B0H	P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0
P4	C0H	P4.7	P4.6	P4.5	P4.4	P4.3	P4.2	P4.1	P4.0
P5	C8H	-	-	P5.5	P5.4	P5.3	P5.2	P5.1	P5.0
P6	E8H	P6.7	P6.6	P6.5	P6.4	P6.3	P6.2	P6.1	P6.0
P7	F8H	P7.7	P7.6	P7.5	P7.4	P7.3	P7.2	P7.1	P7.0

图 12：端口数据寄存器

4.2.2. 端口配置寄存器

端口配置寄存器 PnM1 和 PnM0 都是 8 位的寄存器，PnM1 和 PnM0 寄存器必须组合使用才能正确地配置 IO 口工作模式。

STC15W4K32S4 系列单片机所有 IO 口均有 4 种工作模式：准双向口/弱上拉（标准 8051 输出口模式）、推挽输出/强上拉、高阻输入（电流既不能流入也不能流出）、开漏输出。每个 IO 口工作模式由 PnM1 和 PnM0 寄存器中的相应位控制。如下图。

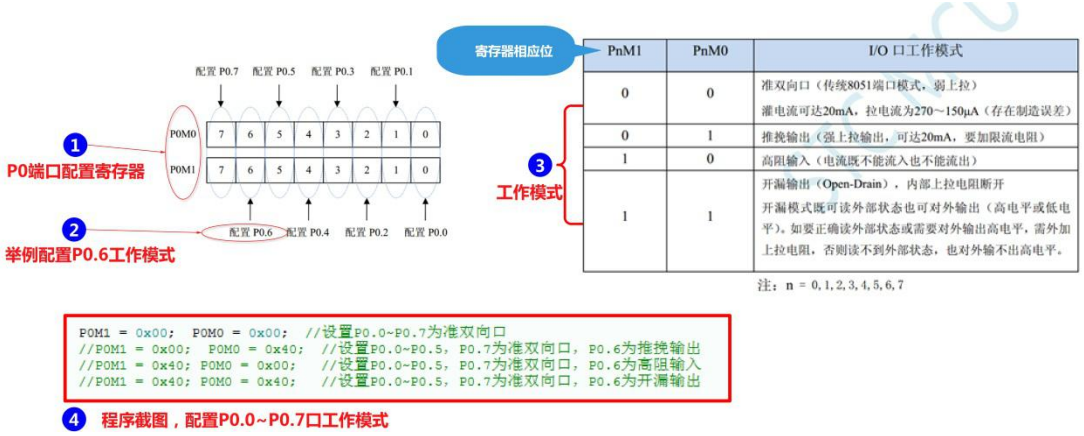


图 13：GPIO 端口配置

4.3. GPIO 驱动 LED 实验（寄存器版本）

✧ 注：本节对应的实验源码是：“实验 2-1-1：GPIO 驱动 LED（寄存器版本）”。

4.3.1. 头文件引用和路径设置

■ 需要宏定义部分及引用的头文件

因为在“main.c”文件中使用了 STC15 的头文件“15W4KxxS4.h”，所以需要引用下面的头文件。在头文件“15W4KxxS4.h”中需要确定主时钟取值，所以宏定义主时钟值。

```
#define MAIN_Fosc       11059200L   //定义主时钟
#include    "15W4KxxS4.H"
```

在程序设计中会用到定义变量的类型，为了定义变量方便，将较为复杂的“unsigned int”和“unsigned char ”进行了宏定义。

```
#define  uint16   unsigned int
#define  uint8    unsigned char
```

这样，再定义变量时可直接使用“uint16”和“uint8”来取代“unsigned int”和“unsigned char ”即可。

■ 需要包含的头文件路径

本例需要包含的头文件路径如下表：

表 3：头文件包含路径

序号	路径	描述
1	..\User	15W4KxxS4.h 头文件在该路径，所以要包含。

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

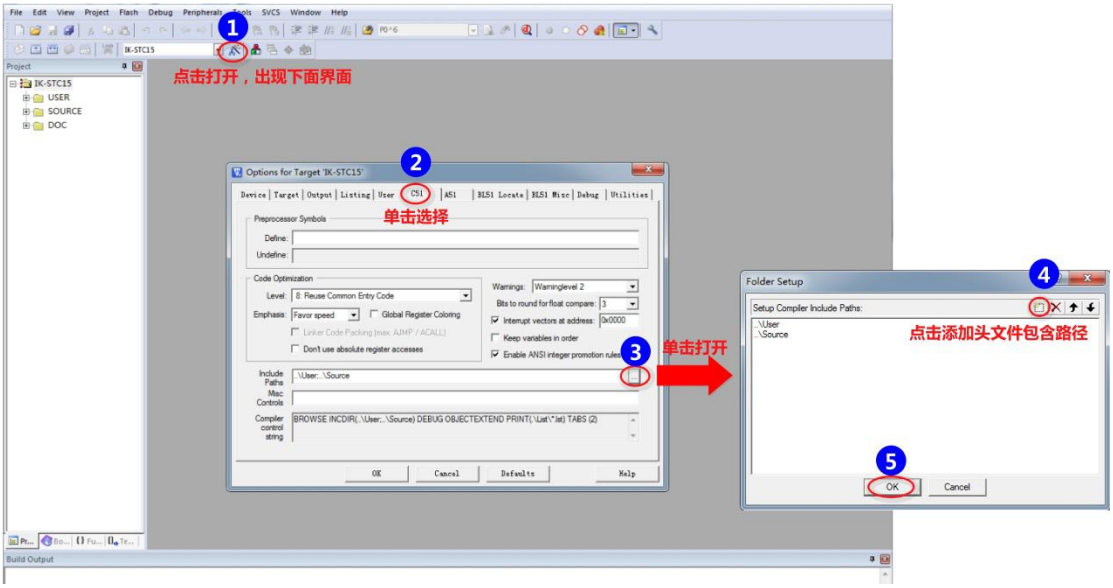


图 14：添加头文件包含路径

4.3.2. 编写代码

首先介绍下毫秒级的延时函数。控制指示灯亮和灭需要中间有足够的间隔时间，这个间隔一般通过延时函数实现。微秒级的延时时间很难控制，这和主频大小及其精度有密切关系。但毫秒级的延时还是可以控制的，下面给出在 11.0592MHZ 下的毫秒延时函数，仅供参考。

代码清单：毫秒延时函数

```
1.  /*************************************
2.  功能描述：延时函数
3.  入口参数：uint16 x ，该值为 1 时，延时 1ms
4.  返回值：无
5.  *************************************/
6.  void delay_ms(uint16 x)
7.  {
8.      uint16 j,i;
9.      for(j=0;j<x;j++)
10.     {
11.         for(i=0;i<1100;i++);
12.     }
13. }
```

在对端口数据寄存器介绍时我们简单介绍过控制单片机 GPIO 的过程。需要再说明的地方是关于两个关键字“sfr”和“sbit”。

sfr 是 Keil C51 为能直接访问 51 内核单片机中的 SFR 而提供了一个关键词，其用法是：

1）sfrt 变量名=地址值。

sbit 是定义特殊功能寄存器的位变量。其用法有三种：

1） sbit 位变量名＝地址值。

2） sbit 位变量名＝SFR 名称^变量位地址值。

3） sbit 位变量名＝SFR 地址值^变量位地址值。

✧ 注：SFR 是 Special Function Register(特殊功能寄存器)的缩写。

程序清单：头文件“15W4KxxS4.h”定义 P0 端口部分

```
1.  sfr P0          =   0x80;   //1111,1111 端口 0
2.  sbit P00        =   P0^0;
3.  sbit P01        =   P0^1;
4.  sbit P02        =   P0^2;
5.  sbit P03        =   P0^3;
6.  sbit P04        =   P0^4;
7.  sbit P05        =   P0^5;
8.  sbit P06        =   P0^6;
9.  sbit P07        =   P0^7;
```

然后，在主函数中先对 P0 口进行模式配置，针对 P0.6 口是被配置了准双向口，后主循环中将用户指示灯 DS1 点亮，延时 200ms，再熄灭，再延时 200ms 的过程，这样可观察到指示灯 DS1 不停闪烁的现象。

代码清单：主函数

```
1.  int main()
2.  {
3.  /////////////////////////////////////////////////
4.  //注意: STC15W4K32S4 系列的芯片,上电后所有与 PWM 相关的 IO 口均为
5.  //       高阻态,需将这些口设置为准双向口或强推挽模式方可正常使用
6.  //相关 IO: P0.6/P0.7/P1.6/P1.7/P2.1/P2.2
7.  //         P2.3/P2.7/P3.7/P4.2/P4.4/P4.5
8.  /////////////////////////////////////////////////
9.      P0M1 = 0x00;    P0M0 = 0x00;    //设置 P0.0~P0.7 为准双向口
10.    //P0M1 = 0x00;  P0M0 = 0x40;    //设置 P0.0~P0.5，P0.7 为准双向口，P0.6 为推挽输出
11.     //P0M1 = 0x40; P0M0 = 0x00;    //设置 P0.0~P0.5,P0.7 为准双向口,P0.6 为高阻输入
12.     //P0M1 = 0x40; P0M0 = 0x40;    //设置 P0.0~P0.5,P0.7 为准双向口,P0.6 为开漏输出
13.
14.   while(1)
15.   {
16.         P06=0;          //控制 P0.6 端口输出低电平，点亮蓝色指示灯 DS1
```

```
17.         delay_ms(200);
18.         P06=1;          //控制 P0.6 端口输出高电平，熄灭蓝色指示灯 DS1
19.         delay_ms(200);
20.     }
21. }
```

4.3.3. 实验步骤

1. 解压“…\第 3 部分：配套例程源码\ 1 - 基础实验程序\”目录下的压缩文件“实验 2-1-1：GPIO 驱动 LED（寄存器版本）”，将解压后得到的文件夹拷贝到合适的目录，如“D\STC15”。
2. 启动 Keil C51。
3. 在 Keil C51 中执行“Project→Open Project”打开“…\led\projec”目录下的工程“LED.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“LED.hex”位于工程目录下的“Output”文件夹中。
5. 打开 STC-ISP 软件下载程序 。下载使用内部 IRC 时钟，IRC 频率选择为 11.0592MHZ。
6. 程序运行后，可以观察到兰色 LED 灯 DS1 间隔 200ms 闪烁一次。

4.4. GPIO 驱动 LED 实验（库函数版本）

✧ 注：本节的实验源码是在“实验 2-1-1：GPIO 驱动 LED（寄存器版本）”的基础上修改。本节对应的实验源码是：“2-1-2：GPIO 驱动 LED（库函数版本）”。

4.4.1. 工程需要用到的 c 文件

本例需要用到的 c 文件如下表所示，工程需要添加下表中的 c 文件。

表 4：实验需要用到的 C 文件

序号	文件名	后缀	功能描述
1	GPIO	.c	通用输入输出。

该 GPIO.c 是 STC 官方提供的有关 GPIO 配置的函数库。

4.4.2. 头文件引用和路径设置

■ 需要引用的头文件

因为在“main.c”文件中使用了 GPIO 相关的库，所以需要引用下面的头文件。

```
1. #include    "GPIO.h"
```

■ 需要包含的头文件路径

本例需要包含的头文件路径如下表：

表 5：头文件包含路径

序号	路径	描述
1	..\ STC_LIB	GPIO.h 和 config.h 头文件在该路径，所以要包含。
2	..\User	15W4KxxS4.h 头文件在该路径，所以要包含。

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

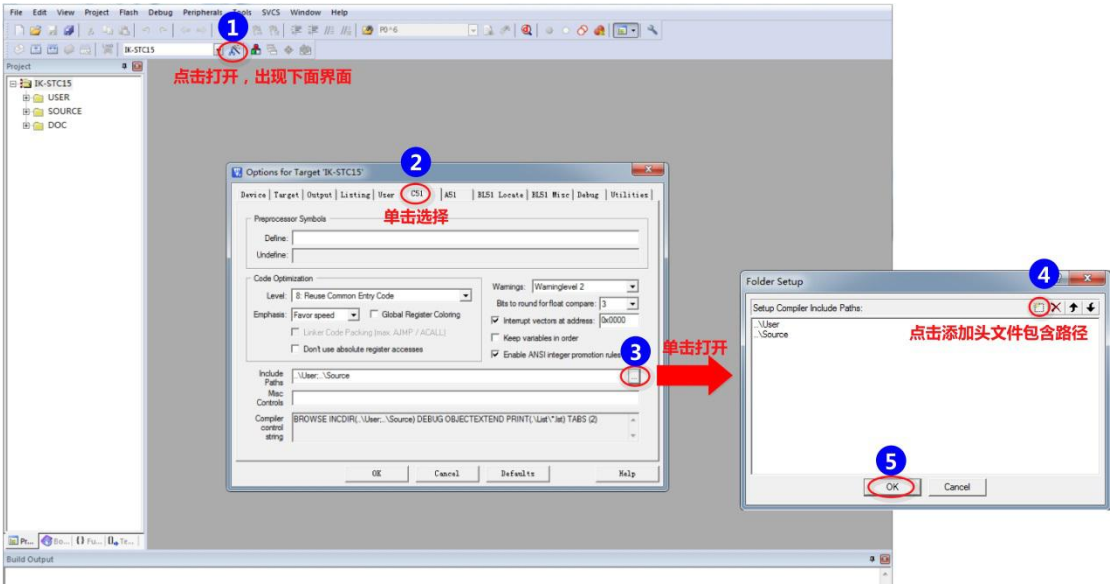


图 15：添加头文件包含路径

4.4.3. 编写代码

首先在 GPIO 口初始化函数中调用库函数 GPIO_Inilize 完成 P0.6 口的工作模式配置，即配置 P0.6 为准双向口。

代码清单：GPIO 口初始化函数

```
1.  /*************************************
2.  功能描述：GPIO 口初始化
3.  入口参数：无
4.  返回值：无
5.  *************************************/
6.  void    GPIO_config(void)
7.  {
8.      GPIO_InitTypeDef  GPIO_InitStructure;
9.
10.     //设置 P0.6 口工作模式
```

```
    GPIO_InitStructure.Mode=GPIO_PullUp;       //配置 P0.6 口为准双向口
      //GPIO_InitStructure.Mode=GPIO_OUT_PP;     //配置 P0.6 口为推挽输出(强上拉)
      //GPIO_InitStructure.GPIO_HighZ;           //配置 P0.6 口为高阻输入
      //GPIO_InitStructure.Mode=GPIO_OUT_OD;     //配置 P0.6 口为开漏输出
    GPIO_InitStructure.Pin=GPIO_Pin_6;
    GPIO_Inilize(GPIO_P0,&GPIO_InitStructure);

}
```

打开库函数 GPIO_Inilize 后会发现，配置 P0.6 口实际最终操作还是 P0M0 和 P0M1 寄存器。代码如下。

程序清单：头文件“15W4KxxS4.h”定义 P0 端口部分

```
//========================================================================
// 函数: u8   GPIO_Inilize(u8 GPIO, GPIO_InitTypeDef *GPIOx)
// 描述: 初始化 IO 口.
// 参数: GPIOx: 结构参数,请参考 gpio.h 里的定义.
// 返回: 成功返回 0, 空操作返回 1,错误返回 2.
//========================================================================
u8  GPIO_Inilize(u8 GPIO, GPIO_InitTypeDef *GPIOx)
{
    if(GPIO > GPIO_P5)                return 1;   //空操作
    if(GPIOx->Mode > GPIO_OUT_PP) return 2;   //错误

    if(GPIO == GPIO_P0)
    {
        if(GPIOx->Mode == GPIO_PullUp)       P0M1 &= ~GPIOx->Pin, P0M0 &= ~GPIOx->Pin;  //上拉准双向口
        if(GPIOx->Mode == GPIO_HighZ)        P0M1 |=  GPIOx->Pin, P0M0 &= ~GPIOx->Pin;  //浮空输入
        if(GPIOx->Mode == GPIO_OUT_OD)       P0M1 |=  GPIOx->Pin, P0M0 |=  GPIOx->Pin;  //开漏输出
        if(GPIOx->Mode == GPIO_OUT_PP)       P0M1 &= ~GPIOx->Pin, P0M0 |=  GPIOx->Pin;  //推挽输出
    }
    if(GPIO == GPIO_P1)
    {
        if(GPIOx->Mode == GPIO_PullUp)       P1M1 &= ~GPIOx->Pin, P1M0 &= ~GPIOx->Pin;  //上拉准双向口
        if(GPIOx->Mode == GPIO_HighZ)        P1M1 |=  GPIOx->Pin, P1M0 &= ~GPIOx->Pin;  //浮空输入
```

```
23.         if(GPIOx->Mode == GPIO_OUT_OD)       P1M1 |=  GPIOx->Pin, P1M0 |=  GPI
    Ox->Pin;  //开漏输出
24.         if(GPIOx->Mode == GPIO_OUT_PP)       P1M1 &= ~GPIOx->Pin, P1M0 |=  GPI
    Ox->Pin;  //推挽输出
25.     }
26.     if(GPIO == GPIO_P2)
27.     {
28.         if(GPIOx->Mode == GPIO_PullUp)       P2M1 &= ~GPIOx->Pin, P2M0 &= ~GPI
    Ox->Pin;  //上拉准双向口
29.         if(GPIOx->Mode == GPIO_HighZ)        P2M1 |=  GPIOx->Pin, P2M0 &= ~GPI
    Ox->Pin;  //浮空输入
30.         if(GPIOx->Mode == GPIO_OUT_OD)       P2M1 |=  GPIOx->Pin, P2M0 |=  GPI
    Ox->Pin;  //开漏输出
31.         if(GPIOx->Mode == GPIO_OUT_PP)       P2M1 &= ~GPIOx->Pin, P2M0 |=  GPI
    Ox->Pin;  //推挽输出
32.     }
33.     if(GPIO == GPIO_P3)
34.     {
35.         if(GPIOx->Mode == GPIO_PullUp)       P3M1 &= ~GPIOx->Pin, P3M0 &= ~GPI
    Ox->Pin;  //上拉准双向口
36.         if(GPIOx->Mode == GPIO_HighZ)        P3M1 |=  GPIOx->Pin, P3M0 &= ~GPI
    Ox->Pin;  //浮空输入
37.         if(GPIOx->Mode == GPIO_OUT_OD)       P3M1 |=  GPIOx->Pin, P3M0 |=  GPI
    Ox->Pin;  //开漏输出
38.         if(GPIOx->Mode == GPIO_OUT_PP)       P3M1 &= ~GPIOx->Pin, P3M0 |=  GPI
    Ox->Pin;  //推挽输出
39.     }
40.     if(GPIO == GPIO_P4)
41.     {
42.         if(GPIOx->Mode == GPIO_PullUp)       P4M1 &= ~GPIOx->Pin, P4M0 &= ~GPI
    Ox->Pin;  //上拉准双向口
43.         if(GPIOx->Mode == GPIO_HighZ)        P4M1 |=  GPIOx->Pin, P4M0 &= ~GPI
    Ox->Pin;  //浮空输入
44.         if(GPIOx->Mode == GPIO_OUT_OD)       P4M1 |=  GPIOx->Pin, P4M0 |=  GPI
    Ox->Pin;  //开漏输出
45.         if(GPIOx->Mode == GPIO_OUT_PP)       P4M1 &= ~GPIOx->Pin, P4M0 |=  GPI
    Ox->Pin;  //推挽输出
46.     }
47.     if(GPIO == GPIO_P5)
48.     {
49.         if(GPIOx->Mode == GPIO_PullUp)       P5M1 &= ~GPIOx->Pin, P5M0 &= ~GPI
    Ox->Pin;  //上拉准双向口
50.         if(GPIOx->Mode == GPIO_HighZ)        P5M1 |=  GPIOx->Pin, P5M0 &= ~GPI
    Ox->Pin;  //浮空输入
```

```
51.         if(GPIOx->Mode == GPIO_OUT_OD)       P5M1 |=  GPIOx->Pin, P5M0 |=  GPI
    Ox->Pin;  //开漏输出
52.         if(GPIOx->Mode == GPIO_OUT_PP)       P5M1 &= ~GPIOx->Pin, P5M0 |=  GPI
    Ox->Pin;  //推挽输出
53.     }
54.     return 0;   //成功
55. }
```

然后，在主函数中先调用 GPIO 口初始化函数，后主循环中将用户指示灯 DS1 点亮，延时 200ms，再熄灭，再延时 200ms 的过程，这样可观察到指示灯 DS1 不停闪烁的现象。

代码清单：主函数

```
1.  int main()
2.  {
3.    GPIO_config();    //设置 P0.6 口为准双向口
4.
5.    while(1)
6.    {
7.          P06=0;          //控制 P0.6 端口输出低电平，点亮蓝色指示灯 DS1
8.          delay_ms(200);
9.          P06=1;          //控制 P0.6 端口输出高电平，熄灭蓝色指示灯 DS1
10.         delay_ms(200);
11.     }
12. }
```

4.4.4. 实验步骤

1. 解压“…\第 3 部分：配套例程源码\ 1 - 基础实验程序\”目录下的压缩文件“实验 2-1-2：GPIO 驱动 LED（库函数版本）”，将解压后得到的文件夹拷贝到合适的目录，如“D\STC15”。
2. 启动 Keil C51。
3. 在 Keil C51 中执行“Project→Open Project”打开“…\led\projec”目录下的工程“led.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“led.hex”位于工程目录下的“Output”文件夹中。
5. 打开 STC-ISP 软件下载程序 。下载使用内部 IRC 时钟，IRC 频率选择为 11.0592MHZ。
6. 程序运行后，可以观察到兰色 LED 灯 DS1 间隔 200ms 闪烁一次。

4.5.流水灯实验（单个 c 文件）

✧ 注：本节的实验源码是在“实验 2-1-1：GPIO 驱动 LED（寄存器版本）”的基础上修改。本节对应的实验源码是：“实验 2-1-3：流水灯（单个 c 文件）”。

4.5.1. 头文件引用和路径设置

本实验需要用到的头文件以及添加头文件包含路径的方法请参考“实验 2-1-1：GPIO 驱动 LED（寄存器版本）”部分。

在程序设计中重新定义了 P0 寄存器的位变量 P0^6 和 P0^7，这是为了控制 GPIO 口比较鲜明地知道其用途。

```
/**********************
引脚别名定义
**********************/
sbit LED_R=P0^7;          //红色 LED 用 IO 口 P07
sbit LED_B=P0^6;          //蓝色 LED 用 IO 口 P06
```

须知，语句“LED_B=0;”和语句“P06=0;”效果是完全一样的；语句“LED_B=1;”和语句“P06=1;”效果也是完全一样的。

4.5.2. 编写代码

首先编写一个函数，该函数会控制 2 个用户 LED 分别依次点亮，代码如下。

程序清单：流水灯点亮函数

```
/*************************************************************************
功能描述：流水灯点亮
入口参数：无
返回值：无
 *************************************************************************/
void LED_Blink(void)
{
        LED_B=0;          //点亮蓝色指示灯
        LED_R=1;          //熄灭红色指示灯
        delay_ms(200);
        LED_B=1;          //熄灭蓝色指示灯
        LED_R=0;          //点亮红色指示灯
        delay_ms(200);
}
```

然后，在主函数中先对 P0.6 和 P0.7 口进行模式配置，后主循环中调用流水灯函数，这样可观察到指示灯 DS1、DS2 被流水点亮。

代码清单：主函数

```
int main()
{
//////////////////////////////////////////////
//注意: STC15W4K32S4 系列的芯片,上电后所有与 PWM 相关的 IO 口均为
//       高阻态,需将这些口设置为准双向口或强推挽模式方可正常使用
//相关 IO: P0.6/P0.7/P1.6/P1.7/P2.1/P2.2
//         P2.3/P2.7/P3.7/P4.2/P4.4/P4.5
//////////////////////////////////////////////
     P0M1 &= 0x3F;   P0M0 &= 0x3F;     //设置 P0.6~P0.7 为准双向口
     //P0M1 &= 0x3F; P0M0 |= 0xC0;   //设置 P0.6~P0.7 为推挽输出
     //P0M1 |= 0xC0;   P0M0 &= 0x3F;   //设置 P0.6~P0.7 为高阻输入
     //P0M1 |= 0xC0;   P0M0 |= 0xC0;   //设置 P0.6~P0.7 为开漏输出

   while(1)
   {
        LED_Blink();      //指示灯流水点亮
     }
}
```

4.5.3. 实验步骤

1. 解压“…\第 3 部分：配套例程源码\ 1 - 基础实验程序\”目录下的压缩文件“实验 2-1-3：流水灯（单个 c 文件）”，将解压后得到的文件夹拷贝到合适的目录，如“D\STC15”。
2. 启动 Keil C51。
3. 在 Keil C51 中执行“Project→Open Project”打开“…\led_blinky\projec”目录下的工程“led_blinky.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“led_blinky.hex”位于工程目录下的“Output”文件夹中。
5. 打开 STC-ISP 软件下载程序 。下载使用内部 IRC 时钟，IRC 频率选择为 11.0592MHZ。
6. 程序运行后，可以观察到兰色 LED 灯 DS1 和红色 LED 灯 DS2 流水点亮。

■ **思考题 1：**分析如果有更多的用户 LED 灯需控制流水点亮，程序如何设计？

4.6.流水灯实验（多个 c 文件）

✧ 注：本节的实验源码是在“实验 2-1-3：流水灯（单个 c 文件）”的基础上修改。本节对应的实验源码是：“实验 2-1-4：流水灯（多个 c 文件）”。

4.6.1. 工程需要用到的 c 文件

本例需要用到的 c 文件如下表所示，工程需要添加下表中的 c 文件。

表 6：实验需要用到的 C 文件

序号	文件名	后缀	功能描述
1	led	.c	包含与用户 led 控制有关的用户自定义函数。
2	delay	.c	包含用户自定义延时函数。

4.6.2. 头文件引用和路径设置

■ 需要引用的头文件

因为在“main.c”文件中使用了控制 led 的函数和延时函数（延时函数没有在 main.c 中定义），所以需要引用下面的头文件。

```
1. #include    "led.h"
2. #include "delay.h"
```

■ 需要包含的头文件路径

本例需要包含的头文件路径如下表：

表 7：头文件包含路径

序号	路径	描述
1	..\ Source	led.h 和 delay.h 头文件在该路径，所以要包含。
2	..\User	15W4KxxS4.h 头文件在该路径，所以要包含。

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

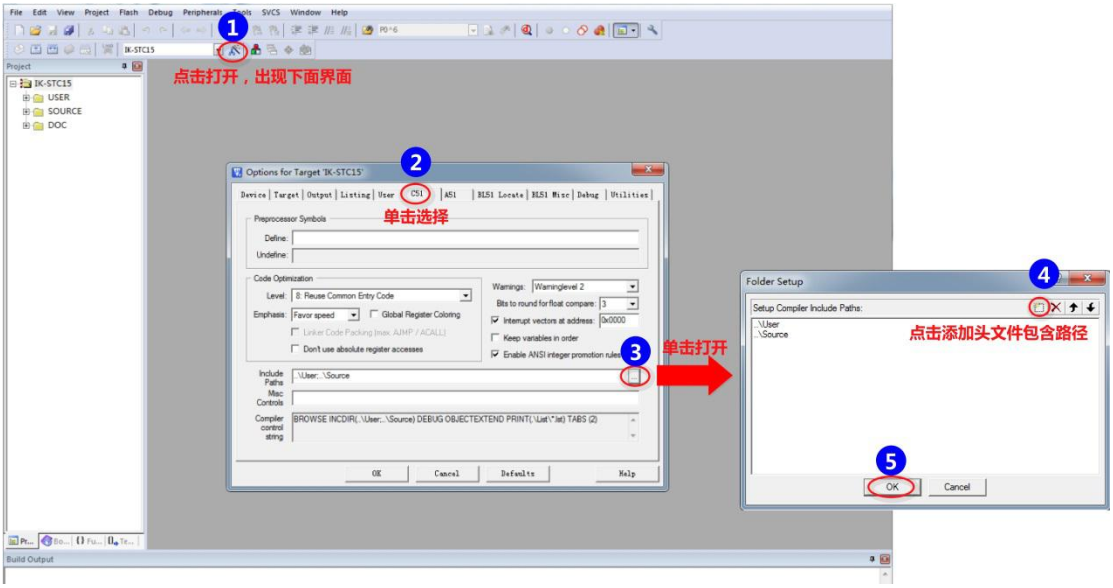


图 16：添加头文件包含路径

4.6.3. 编写代码

首先在 delay.c 文件中编写一个延时函数 delay_ms，该函数是毫秒延时，代码如下。

程序清单：延时函数

```
1.  /*************************************
2.  功能描述：延时函数
3.  入口参数：uint16 x ，该值为 1 时，延时 1ms
4.  返回值：无
5.  *************************************/
6.  void delay_ms(uint16 x)
7.  {
8.      uint16 j,i;
9.      for(j=0;j<x;j++)
10.     {
11.         for(i=0;i<1100;i++);
12.     }
13. }
```

该 delay_ms 函数会在 delay.h 头文件中被声明，这样可以被外部调用。如下。

```
1.  extern void delay_ms(uint16 x);
```

然后在 led.c 文件中封装和 2 个 LED 有关的所有基本操作函数。如下表所示的 5 个函数。

表 8：用户 LED 基本操作函数汇集（详见 led.c）

序号	函数名	功能描述
1	led_on	点亮一个指定的 LED。
2	led_off	熄灭一个指定的 LED。
3	led_toggle	翻转一个指定的 LED 的状态。
4	leds_on	点亮开发板上的 2 个指示灯。
5	leds_off	熄灭开发板上的 2 个指示灯。

LED 基本操作函数程序清单如下：

程序清单：点亮一个指定的 LED

```
/*******************************************************************************
功能描述：点亮一个指定的指示灯(DS1 DS2)
入口参数：uint8 led_idx  (可取值 LED_1 或 LED_2)
返回值：无
*******************************************************************************/
void led_on(uint8 led_idx)
{
  switch(led_idx)
    {
      case LED_1:
     LED_B=0;         //控制 P0.6 端口输出低电平，点亮蓝色指示灯 DS1
        break;
      case LED_2:
     LED_R=0;         //控制 P0.7 端口输出低电平，点亮红色指示灯 DS2
        break;
      default:
        break;
  }
}
```

程序清单：熄灭一个指定的 LED

```
/*******************************************************************************
功能描述：熄灭一个指定的指示灯(DS1 DS2)
入口参数：uint8 led_idx  (可取值 LED_1 或 LED_2)
返回值：无
```

```
5.   ********************************************************************************/
6.  void led_off(uint8 led_idx)
7.  {
8.    switch(led_idx)
9.      {
10.         case LED_1:
11.       LED_B=1;          //控制 P0.6 端口输出高电平，熄灭蓝色指示灯 DS1
12.           break;
13.         case LED_2:
14.       LED_R=1;          //控制 P0.7 端口输出高电平，熄灭红色指示灯 DS2
15.           break;
16.         default:
17.           break;
18.   }
19. }
```

程序清单：翻转一个指定的 LED 的状态

```
1.  /********************************************************************************
2.  功能描述：翻转一个指定的指示灯(DS1 DS2)
3.  入口参数：uint8 led_idx  (可取值 LED_1 或 LED_2)
4.  返回值：无
5.  ********************************************************************************/
6.  void led_toggle(uint8 led_idx)
7.  {
8.    switch(led_idx)
9.      {
10.         case LED_1:
11.       LED_B=~LED_B;        //控制 P0.6 端口输出不同于上一次的电平，翻转蓝色指示灯 DS1
12.           break;
13.         case LED_2:
14.       LED_R=~LED_R;        //控制 P0.7 端口输出不同于上一次的电平，翻转红色指示灯 DS2
15.           break;
16.         default:
17.           break;
18.   }
19. }
```

程序清单：同时点亮开发板上的 2 个指示灯

```
1.  /********************************************************************************
```

```
2.  功能描述：点亮开发板上的 2 个指示灯(DS1 DS2)
3.  入口参数：无
4.  返回值：无
5.   ***********************************************************************/
6.  void leds_on(void)
7.  {
8.          LED_B=0;            //控制 P0.6 端口输出低电平，点亮蓝色指示灯 DS1
9.          LED_R=0;          //控制 P0.7 端口输出低电平，点亮红色指示灯 DS2
10. }
```

程序清单：同时熄灭开发板上的 2 个 LED

```
1.  /***********************************************************************
2.  功能描述：熄灭开发板上的 2 个指示灯(DS1 DS2)
3.  入口参数：无
4.  返回值：无
5.   ***********************************************************************/
6.  void leds_off(void)
7.  {
8.          LED_B=1;            //控制 P0.6 端口输出高电平，熄灭蓝色指示灯 DS1
9.          LED_R=1;          //控制 P0.7 端口输出高电平，熄灭红色指示灯 DS2
10. }
```

在 led.c 文件中还编写了一个流水灯点亮的函数 LED_Blink，该函数调用 LED 的基本函数实现流水点亮 2 个用户指示灯的目的。代码如下。

程序清单：流水灯点亮函数

```
1.  /***********************************************************************
2.  功能描述：流水灯点亮指示灯(DS1 DS2)
3.  入口参数：无
4.  返回值：无
5.   ***********************************************************************/
6.  void LED_Blink(void)
7.  {
8.          leds_off();             //熄灭所有用户指示灯
9.          led_on(LED_1);          //点亮指示灯 DS1
10.         delay_ms(200);
11.         leds_off();             //熄灭所有用户指示灯
12.         led_on(LED_2);          //点亮指示灯 DS2
13.         delay_ms(200);
14. }
```

在 led.h 头文件中会声明可供外部调用的函数，LED_Blink 函数便是其中之一。如下。

```
1.  extern void led_on(uint8 led_idx);
2.  extern void led_off(uint8 led_idx);
3.  extern void led_toggle(uint8 led_idx);
4.  extern void leds_on(void);
5.  extern void leds_off(void);
6.  extern void LED_Blink(void);
```

最后，在主函数中会先对 P0.6 和 P0.7 口进行模式配置，后主循环中调用流水灯函数，这样可观察到指示灯 DS1、DS2 被流水点亮。

代码清单：主函数

```
1.  int main()
2.  {
3.  ////////////////////////////////////////////////
4.  //注意: STC15W4K32S4 系列的芯片,上电后所有与 PWM 相关的 IO 口均为
5.  //       高阻态,需将这些口设置为准双向口或强推挽模式方可正常使用
6.  //相关 IO: P0.6/P0.7/P1.6/P1.7/P2.1/P2.2
7.  //         P2.3/P2.7/P3.7/P4.2/P4.4/P4.5
8.  ////////////////////////////////////////////////
9.      P0M1 &= 0x3F;   P0M0 &= 0x3F;     //设置 P0.6~P0.7 为准双向口
10.     //P0M1 &= 0x3F; P0M0 |= 0xC0;   //设置 P0.6~P0.7 为推挽输出
11.   //P0M1 |= 0xC0;   P0M0 &= 0x3F;   //设置 P0.6~P0.7 为高阻输入
12.   //P0M1 |= 0xC0;   P0M0 |= 0xC0;   //设置 P0.6~P0.7 为开漏输出
13.
14.   while(1)
15.   {
16.       LED_Blink();    //指示灯流水点亮
17.     }
18. }
```

4.6.4. 实验步骤

1. 解压“…\第 3 部分：配套例程源码\ 1 - 基础实验程序\”目录下的压缩文件“实验 2-1-4：流水灯（多个 c 文件）”，将解压后得到的文件夹拷贝到合适的目录，如“D\STC15”。
2. 启动 Keil C51。
3. 在 Keil C51 中执行“Project→Open Project”打开“…\led_blinky\projec”目录下的工程“led_blinky.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程

序，直到编译成功为止。编译后生成的 HEX 文件“led_blinky.hex”位于工程目录下的“Output”文件夹中。

5. 打开 STC-ISP 软件下载程序 。下载使用内部 IRC 时钟，IRC 频率选择为 11.0592MHZ。
6. 程序运行后，可以观察到兰色 LED 灯 DS1 和红色 LED 灯 DS2 流水点亮。