

玩转 51 单片机

—— “傀儡师”实验板配套教程 V1. 1

发布日期：2017 年 11 月 20 日

前言

从接手这个项目到此时此刻，不知不觉已经过了 11 个月了。

中间我一直以各种比赛和其他项目为重，一直用着空余时间和伙伴一起一点一点地做起来。从项目的讨论到模块设计到板子成型再到程序和例程的编写。回想起来，这个过程中，有找不到问题所在时的痛苦、有项目进行不下去时的踌躇和迷茫、有突破困难时的欢呼雀跃。当然，最多的是看着实验板从零开始，慢慢成长成能拿得出手的产品，一种油然而生的成就感。

傀儡师实验板从一开始，就是一个刚出生的婴儿。它不同于市面上已有的任何一款单片机实验板。因为市面上琳琅满目，却又千篇一律。如果我们直接按照他们的思路做一块实验板，那我们不就成了搬运工了吗？我们想要的，是一款从来没有的实验板。所以从一开始，我们就用自己的思路来设计这款真正属于我们自己的实验板。

从接触实验板到开始各类比赛到现在。随着见识的增加，对实验板也有着自己的见解。在总结市面上几款 51 单片机实验板后，我们得出结论——要适应这个市场，就必须设计出一款适合大学生比赛的实验板。

说到比赛，刚好我们经历了从校内到校外各种各样的比赛。比赛需要什么功能，那些功能更能适应比赛，我们比谁都清楚。所以，根据比赛中经常用到的功能，我们都有选择性的加到我们的实验板上。而且我们把实验板做得足够的小，让大家可以直接用实验板参加比赛，而不是又浪费钱买最小系统。总结起来，傀儡师实验板将是一块即可以用来学习，也可以用来比赛的高性价比实验板。

差不多一年的光阴。对人生来说如白驹过隙，但对于珍贵的大学四年来说，却是珍贵的一年。希望多少年后，回首过去，傀儡师实验板会是我大学四年中，最耀眼的一颗星。同时也真心希望得到这块实验板的你，能从中获得一门技术，多做项目，收获独属于你的大学回忆。

钟键文

2017 年 11 月 5 日

广东工业大学

目录

前言.....	- 2 -
目录.....	- 3 -
准备篇.....	- 8 -
一. 硬件介绍——瞧，这是一块特别的板子.....	- 9 -
二. 软件准备——工欲善其事，必先利其器.....	- 13 -
2.1 软件下载.....	- 13 -
2.2 软件破解.....	- 16 -
2.3 串口驱动.....	- 21 -
二. 单片机基础知识——关于单片机的八卦事.....	- 23 -
3.1 浅谈 51 单片机.....	- 23 -
3.2 计算机语言.....	- 24 -
3.3 数制.....	- 25 -
3.4 TTL 电平信号.....	- 26 -
3.5 寄存器与特殊功能寄存器.....	- 27 -
3.6 从 STC89 过渡到 STC12.....	- 28 -
入门篇.....	- 29 -
一、点亮 LED——单片机中的“HELLO WORLD！”.....	- 30 -
1.1 硬件准备.....	- 30 -
1.2 知识点讲解.....	- 32 -
1.3 软件实现.....	- 34 -
1.4 下载验证.....	- 42 -
1.5 思考总结.....	- 45 -
二、流水灯——流动闪烁的 LED.....	- 46 -
2.1 硬件准备.....	- 46 -
2.2 知识点讲解.....	- 47 -
2.3 软件实现.....	- 48 -
2.4 下载验证.....	- 54 -
2.5 思考总结.....	- 54 -
三、电压比较器——模数转换入门.....	- 55 -

3.1 硬件准备.....	- 55 -
3.2 知识点讲解.....	- 56 -
3.3 软件实现.....	- 58 -
3.4 下载验证.....	- 59 -
3.5 思考总结.....	- 59 -
四、 PWM 入门——控制技术中的万金油.....	- 61 -
4.1 知识点讲解.....	- 61 -
4.2 软件实现.....	- 62 -
4.3 仿真验证.....	- 63 -
4.4 下载验证.....	- 67 -
4.5 思考总结.....	- 67 -
进阶篇.....	- 68 -
一、 按键——程序由我掌控.....	- 69 -
1.1 硬件准备.....	- 69 -
1.2 知识点讲解.....	- 70 -
1.3 软件实现.....	- 71 -
1.4 下载验证.....	- 77 -
1.5 思考总结.....	- 77 -
二、 定时器——单片机里的小闹钟.....	- 79 -
2.1 知识点讲解.....	- 79 -
2.2 软件实现.....	- 81 -
2.3 下载验证.....	- 84 -
2.4 思考总结.....	- 84 -
三、 PWM 进阶——控制技术中的万金油 2.....	- 86 -
3.1 知识点讲解.....	- 86 -
3.2 软件实现.....	- 89 -
3.3 下载验证.....	- 93 -
3.4 思考总结.....	- 96 -
四、 串口通讯——来和单片机悄悄话.....	- 97 -
4.1 知识点讲解.....	- 97 -
4.2 软件实现.....	- 100 -
4.3 下载验证.....	- 104 -

4.4 思考总结.....	- 106 -
应用篇.....	- 107 -
一、数码管静态显示——LED 变身高富帅.....	- 108 -
1.1 硬件准备.....	- 108 -
1.2 知识点讲解.....	- 109 -
1.3 软件实现.....	- 112 -
1.4 下载验证.....	- 124 -
1.5 思考总结.....	- 125 -
二、数码管动态显示——LED 变身白富美.....	- 126 -
2.1 知识点讲解.....	- 126 -
2.2 软件实现.....	- 127 -
2.3 下载验证.....	- 132 -
2.4 思考总结.....	- 132 -
三、角度舵机——机电一体化大门 1.....	- 133 -
3.1 硬件准备.....	- 133 -
3.2 知识点讲解.....	- 134 -
3.3 软件实现.....	- 136 -
3.4 下载验证.....	- 138 -
3.5 思考总结.....	- 139 -
四、直流电机——机电一体化大门 2.....	- 141 -
4.1 知识点讲解.....	- 141 -
4.2 软件实现.....	- 143 -
4.3 下载验证.....	- 148 -
4.4 思考总结.....	- 148 -
五、喇叭——我是贝多芬.....	- 149 -
5.1 硬件准备.....	- 149 -
5.2 知识点讲解.....	- 149 -
5.3 软件实现.....	- 151 -
5.4 下载验证.....	- 159 -
5.5 总结思考.....	- 159 -
六、咪头——模数转换进阶.....	- 160 -
6.1 硬件准备.....	- 160 -

6.2 知识点讲解.....	- 161 -
6.3 软件实现.....	- 162 -
6.4 下载验证.....	- 166 -
6.5 总结思考.....	- 167 -
七、超声波测距——隔空测距于无形.....	- 168 -
7.1 知识点讲解.....	- 168 -
7.2 软件实现.....	- 169 -
7.3 下载验证.....	- 173 -
7.4 总结思考.....	- 174 -
八、温度传感器——感受世间冷暖.....	- 175 -
8.1 硬件准备.....	- 175 -
8.2 知识点讲解.....	- 177 -
8.3 软件实现.....	- 178 -
8.4 下载验证.....	- 186 -
8.5 总结思考.....	- 187 -
九、RGBLED——全彩精灵.....	- 188 -
9.1 硬件准备.....	- 188 -
9.2 知识点讲解.....	- 189 -
9.3 软件实现.....	- 190 -
9.4 下载验证.....	- 197 -
9.5 总结思考.....	- 197 -
十、红外遥控——决策千里之外.....	- 199 -
10.1 硬件准备.....	- 199 -
10.2 知识点讲解.....	- 200 -
10.3 软件实现.....	- 202 -
10.4 下载验证.....	- 209 -
10.5 总结思考.....	- 209 -
十一、 I^2C ——总线协议入门.....	- 211 -
11.1 知识点讲解.....	- 211 -
11.2 软件实现.....	- 213 -
十二、OLED—— I^2C 实战演练.....	- 216 -
12.1 硬件准备.....	- 216 -

12.2 知识点讲解.....	- 217 -
12.3 软件实现.....	- 217 -
12.4 下载验证.....	- 232 -
12.5 总结思考.....	- 232 -
结语.....	- 234 -

准备篇

工欲善其事，必先利其器。单片机是一门集软件、硬件一体的一门学科。软件硬件如同白昼与黑夜，两者缺一不可。接下来，让我们以软件做盾，以硬件为剑，来武装自己，一起进入单片机的神奇世界尽情探险！

一. 硬件介绍——瞧，这是一块特别的板子

在实验板的开发过程中，身边一直有人问我一个问题：你们的实验板有什么特别？“要做出有特色的实验板”，这是我和这个项目的伙伴陈欣一直努力的方向。毕竟市面上的51单片机实验板琳琅满目，但重复得又千篇一律。为了让我们的实验板即实用又别出心裁，能够成为大浪淘沙中成为最后留下的金子，我们做出了很多改变与革新。

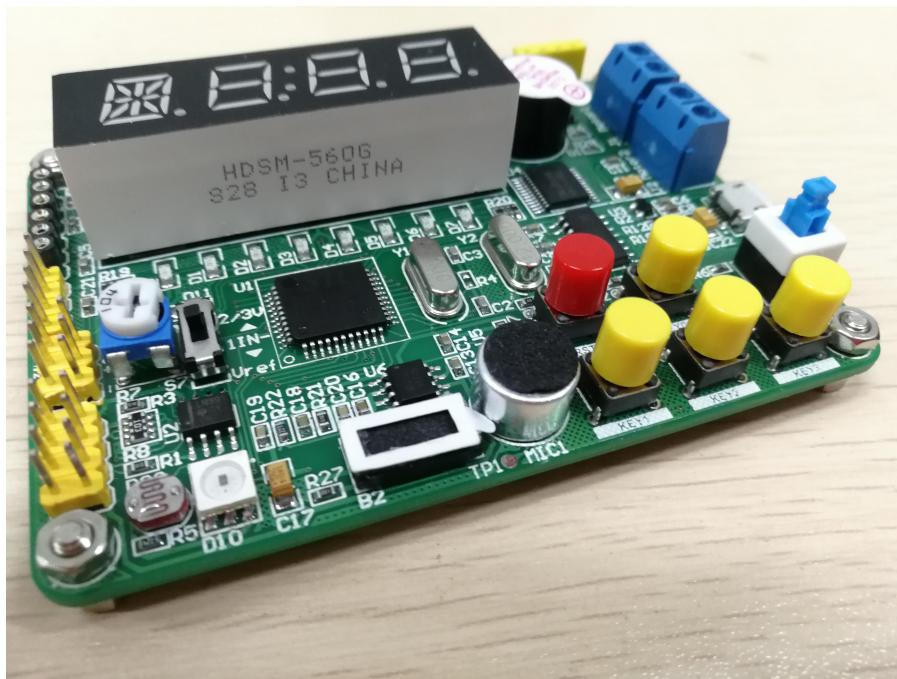


图 1.1.1 傀儡师实验板

首先从芯片的选择上，我们抛弃了STC89芯片而选择它的增强型STC12。相比于市面上一直在用的STC89系列，STC12它的功能更加强大，能方便实现更多功能，大大提升实验板的实用性。



图 1.1.2 STC12 芯片

其次在模块上，我们剔除如今实际应用已经很少用到但市面上很多 51 单片机实验板仍然保留的功能：矩阵键盘、LED 点阵、1602 液晶接口和 12864 液晶接口。这些功能在实际应用中已经甚少实用。如果你学习了 51 单片机相关知识，即使要使用起来也是非常简单。如果把这些功能放在实验板上是无谓地浪费资源，因此我们经过考虑选择了剔除。

在数码管方面，我们一改大多数实验板采用的 8 位的八段数码管，采用的是 4 位的米字加八段的数码管。它能既有米字显部分也有八段显部分，米字显示部分能显示很多八段显示部分不能显示的英文字母。更节省空间更实用（放心，我们能把 4 位的数码管玩得比 8 位数码管更溜）。



图 1.1.3 实验板上数码管

另外我们取消了 1602 液晶和 12864 液晶的接口，在显示方面我们选用更实用的 OLED 接口来替代。在实际应用中，OLED 更加实用、方便、清晰和美观，它的功能完全可以替代 1602 液晶和 12864 液晶。

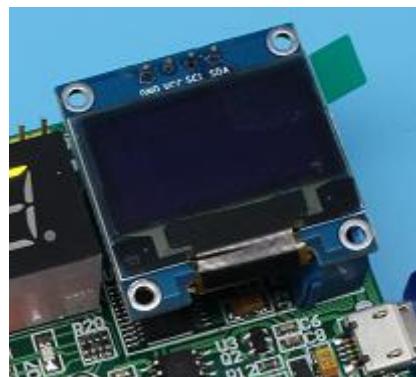


图 1.1.4 实验板上 OLED

在其他接口方面，我们还留出了红外接收接口、温度传感器接口和超声波测距模块接口。这三个功能是在我们总结出很多单片机类比赛都要用到这些功能而设计的。而且为了达到灵活使用的目的，这些模块是可拆卸的，用到时只需将模块插到接口即可。

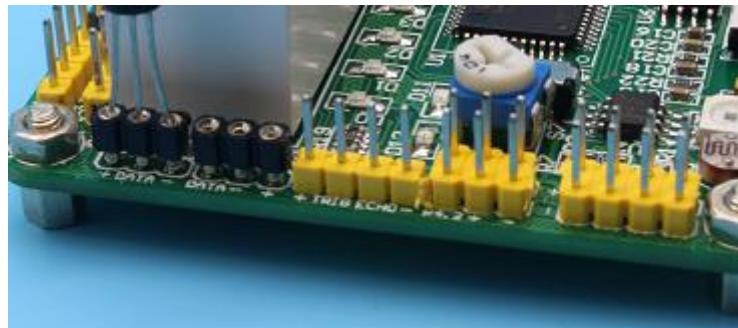


图 1.1.5 实验板上各种外接接口

另外实验板特地留出了两路电机输出和两路舵机输出。这两个是在小车项目或一些机械结构中必须用到的功能。



图 1.1.6 实验板上电机控制接口

除了这些很多实验板都有的功能外，我们还新增了几个创新模块：

电压比较器模块。这个模块可以让大家从更低的门槛去了解模数转换的工作原理。



图 1.1.7 实验板上电压比较器模块

RGBLED 模块。RGBLED 是一种可以发出绚烂全彩颜色的神奇 LED 灯。



图 1.1.8 实验板上 RGBLED

喇叭和咪头模块。通过喇叭模块，大家可以演奏各种音调歌。通过咪头模块，我见将各种声音装换成电信号。结合这个两个模块，大家可以自己通过咪头录音，通过采样后从喇叭播放出来。



图 1.1.9 实验板上喇叭以及咪头

最后，也是最重要的，在保证模块尽量丰富的前提下，我们将实验板做到尽可能的小。经过精心布局，我们成功将实验板做到 63*88mm 的大小，远远小于市面上的实验板面积。把实验板做到手掌大小的目的，除了携带方便外，最重要的是在一些比赛上你可以直接用我们的实验板充当主控系统，而无需再买一块最小板。很多单片机相关的比赛都和小车或者动作控制有关，这也是为什么我们留有两路电机输出和两路舵机输出的一个重要原因。

二. 软件准备——工欲善其事，必先利其器

本教程内容所涉及的软件仅供教学使用，不得用于商业用途。个人或公司因商业用途导致的法律责任，后果自负。

2.1 软件下载

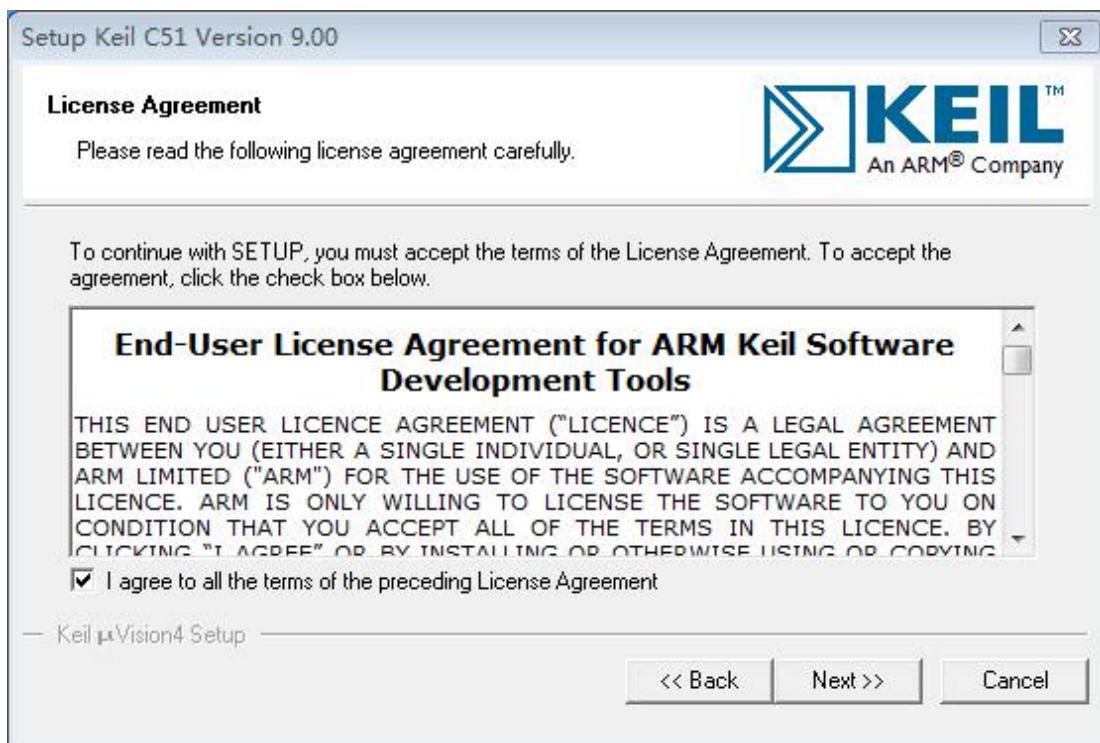
学过 C 语言的同学都清楚，C 语言的开发需要 1 个集成开发环境（IDE）。所谓集成开发环境，就是 1 个让你进行开发的平台。单片机的开发同样需要 1 个集成开发环境（其实说白了就是 1 个开发软件）。在接下来的 51 单片机学习中，我们使用的是功能非常强大的 Keil4。下面，就让我手把手教大家安装 Keil4！**注意，Keil 仅能在 Windows 系统上运行！！！**

首先打开光盘中的软件->Keil4->c51v900.exe

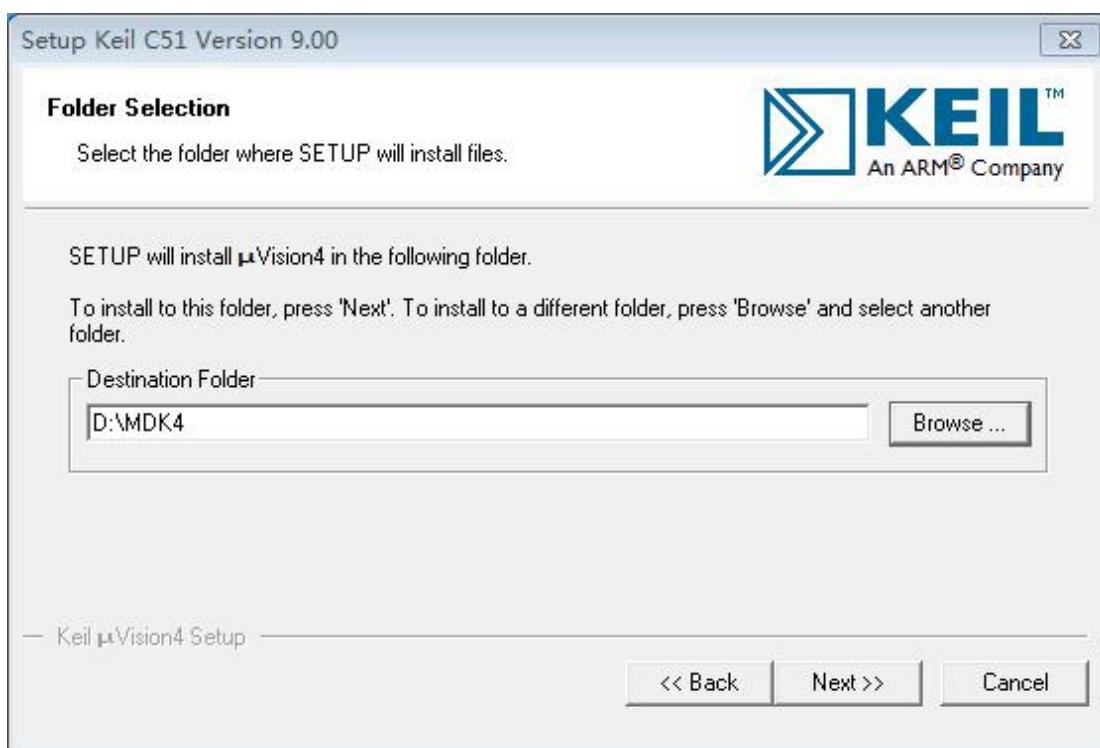
按 Next 下一步



打钩同意协议后，按 Next 下一步



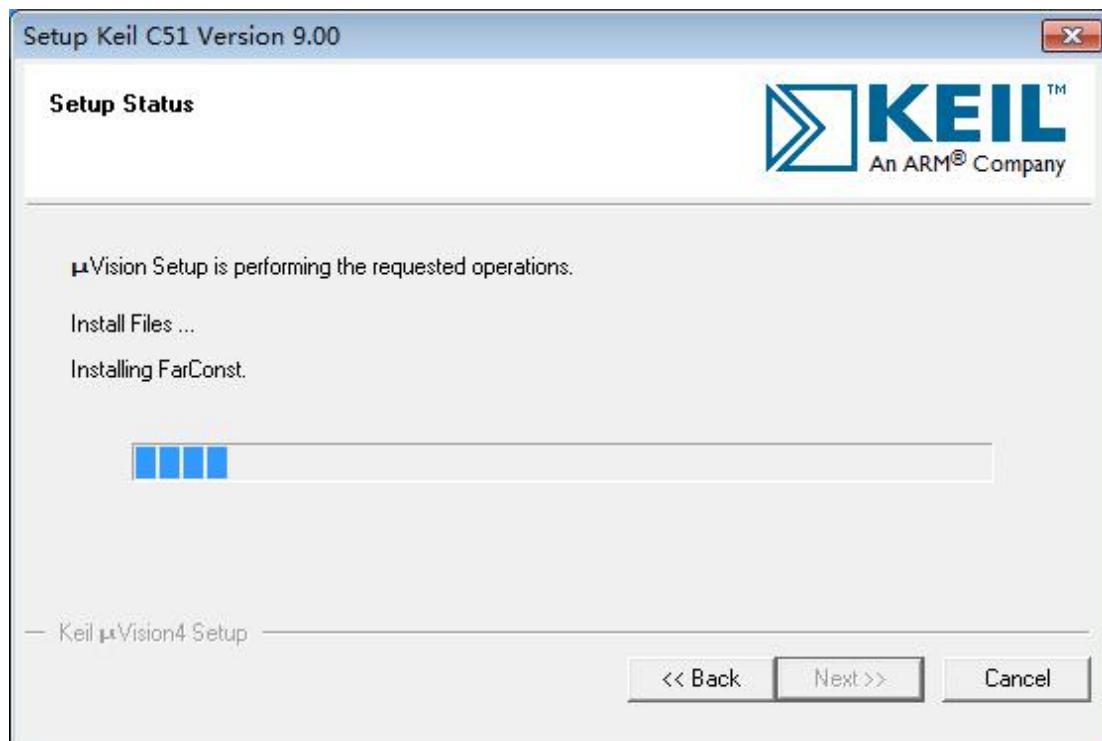
在安装目录选择中，建议以按默认的目录安装，也可以安装到自己想安装的目录。
但千万要注意！文件名不能有中文！文件名不能有中文！文件名不能有中文！
(重要的事要说三遍)



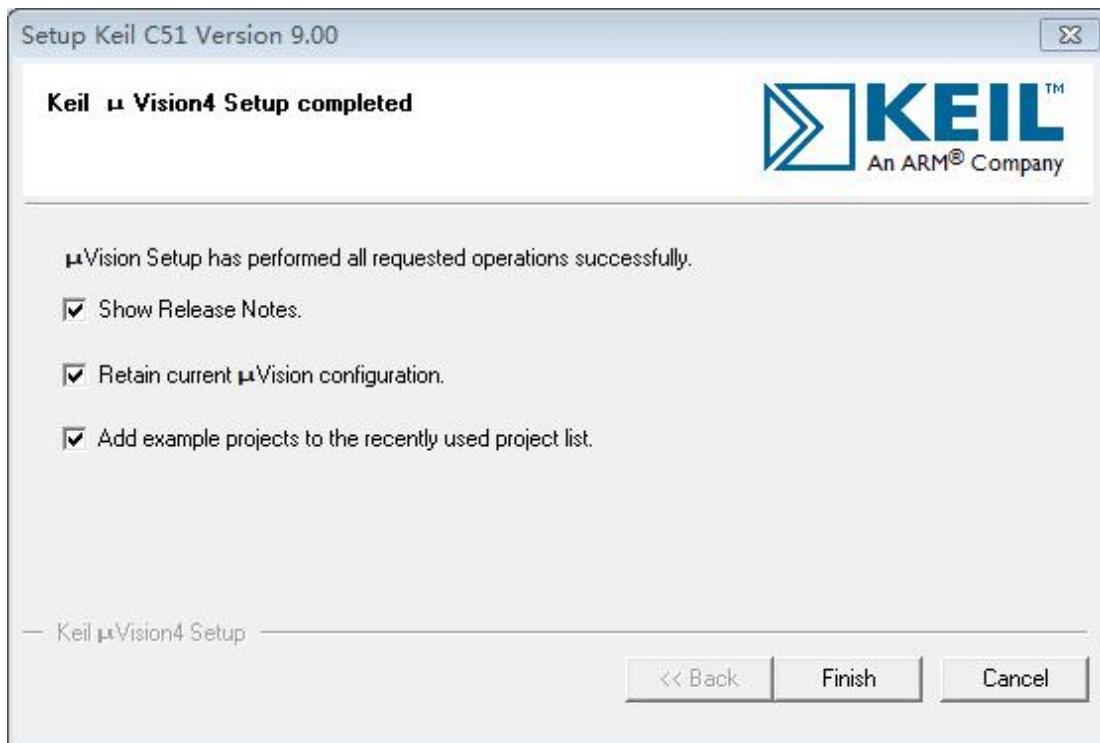
个人信息（随便填）



然后进行安装



最后完成下载

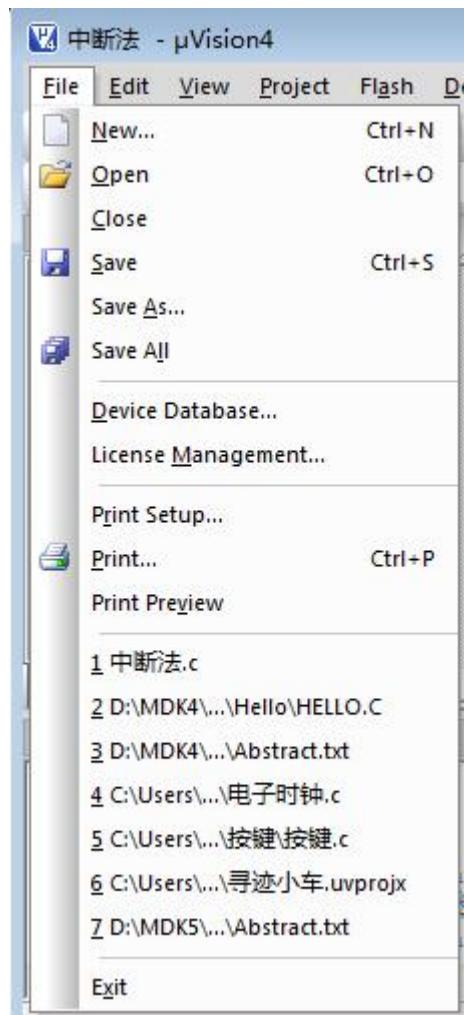


2.2 软件破解

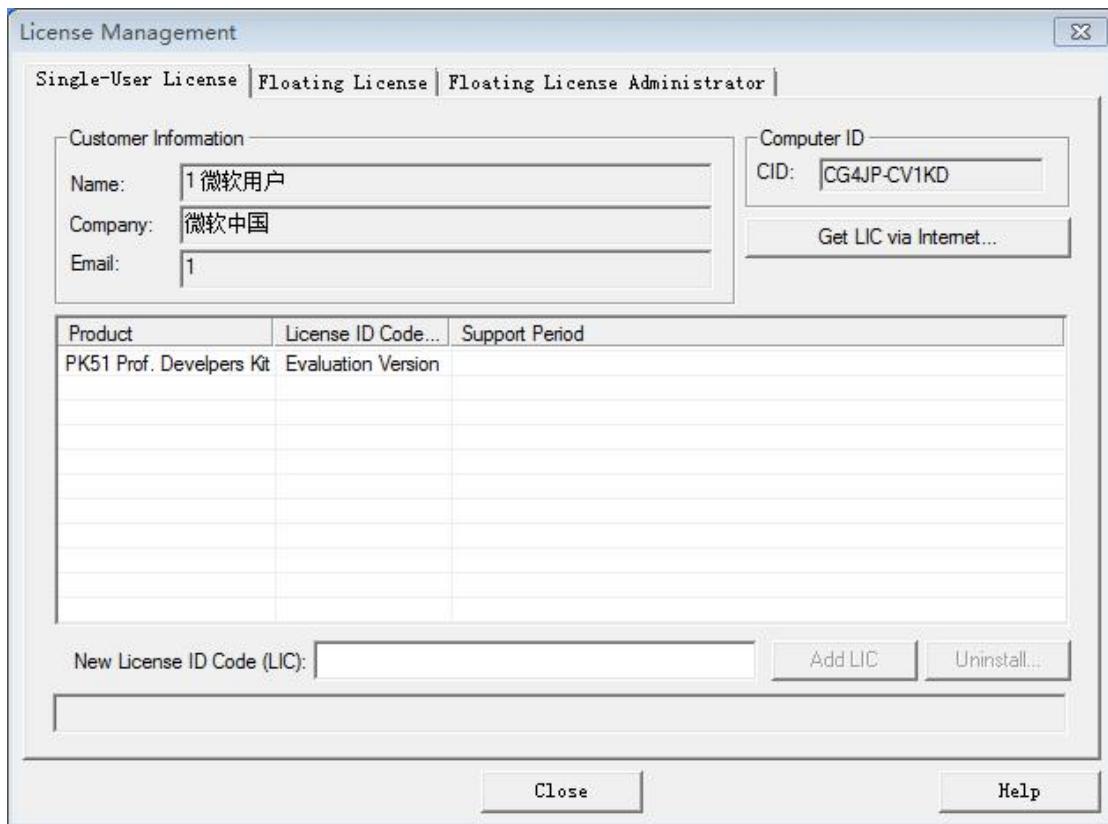
因为软件是未破解的，所以你写的程序的大小上会有所限制。接下来，就让我带大家进行 Keil4 的破解。

首先打开安装好的 keil4，在左上角的目录栏中打开 file

选择 License Management...



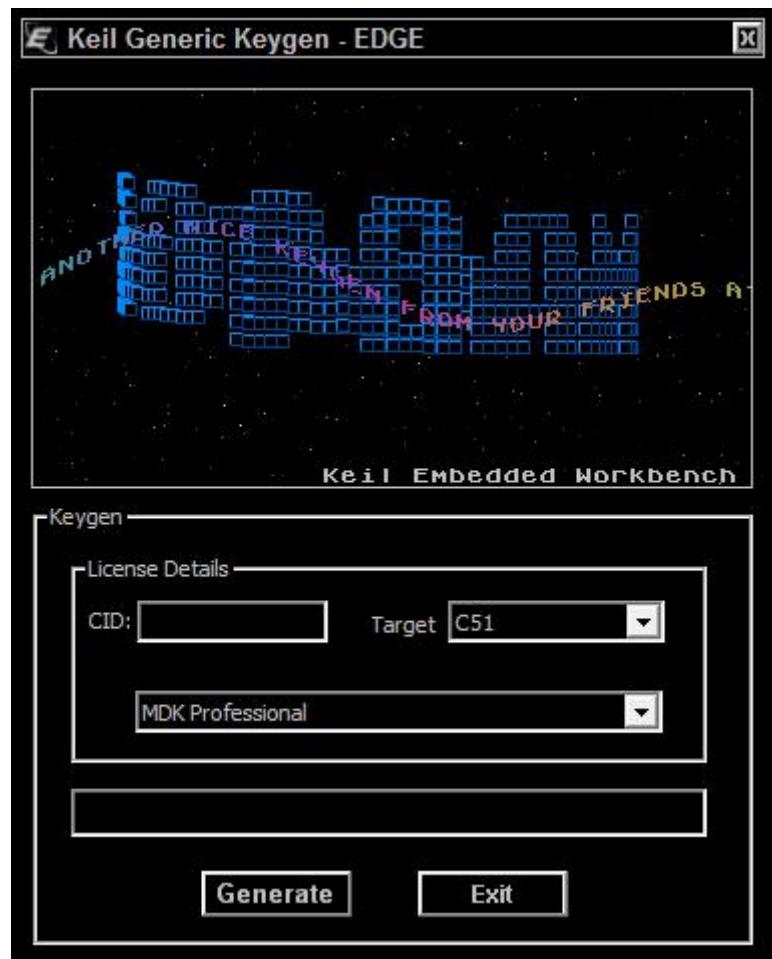
复制右上角的 CID



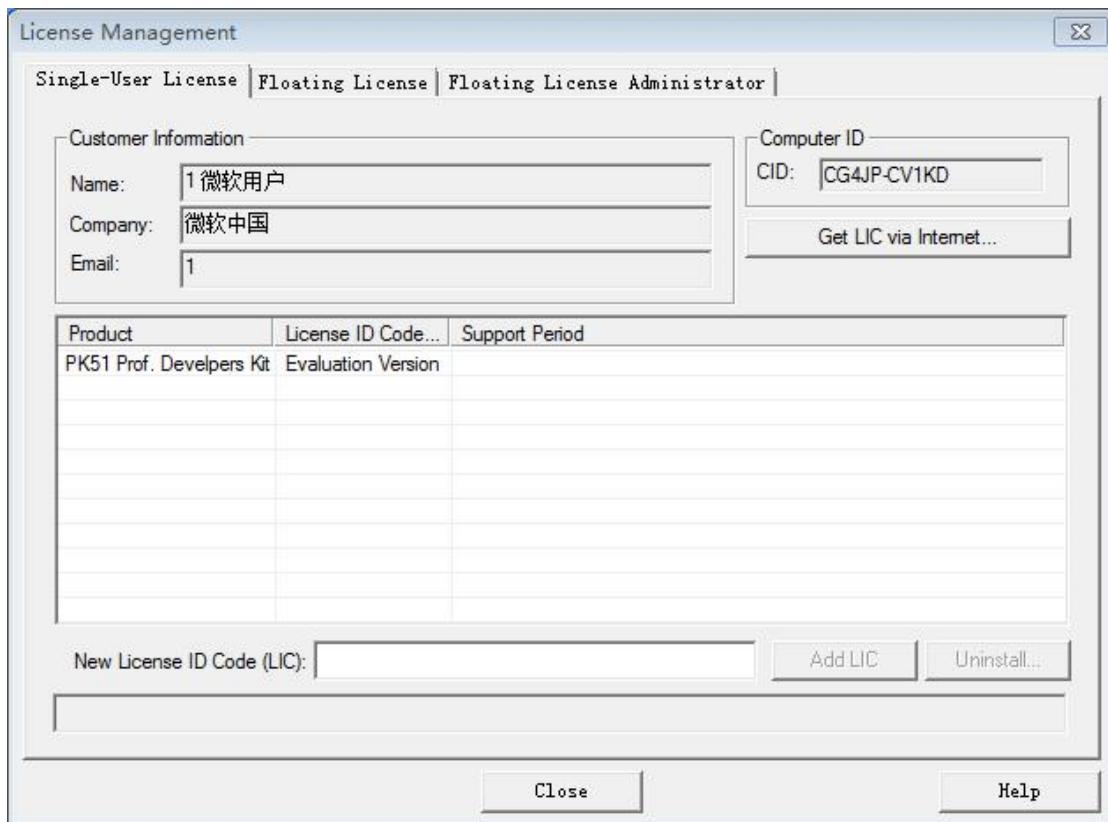
打开光盘中的软件->Keil4->keygen.exe

这里有两点要注意：1、对于注册机，杀毒软件一般会报错并误删，此时只需恢复文件并把文件信任即可；2、破解机的背景音乐很魔性，值得一听。

把刚刚复制的 CID 粘贴到 CID 栏中，Target 选择 C51，按 Generate 生成 1 个注册码，复制之。



返回到 Keil4 中，将刚刚复制的破解码粘贴到下方 New License ID Code(LIC)右侧的框中，按 Add LIC (实测可能会注册失败，此时你只需要返回上一步，按 Generate 重新生成 1 个注册码，使用这个注册码进行注册即可)

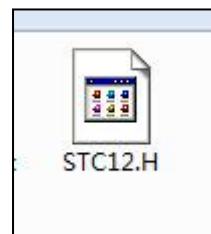


可以看到，我们的 Keil4 已经可以用到 2020 年了！

Product	License ID Code (LIC)	Support Period
*** Unknown Product ***	EPYII-TGKXG-A6XQP-MM1Z1-HVEYP-5W1ZK	Expires: Oct 2020

注册成功后，我们就要添加适合我们单片机型号的头文件。对于头文件不了解的同学，我们以后会说明添加头文件的作用。这是后事，现在我们只需做好前期准备工作即可。

返回光盘中的 3.软件->1.Keil4 中



我们发现会有 1 个文件叫 “STC12.H” 。我们把这个文件复制，然后前往我们下载 Keil4 的目录中，打开文件夹 “C51” ，找到文件夹 “INC” ，将 “STC12.H” 粘贴到文

件夹中，这就完成头文件的添加。（这一步非常重要！即使你以前安装过 keil 也要执行这个步骤，否则程序将不能通过编译！）

2.3 串口驱动

单片机中，代码最终是通过硬件体现出来的。所以，我们就要把我们在电脑上写的程序烧录进实验板中。我们实验板用到的是 CH340 转串口（就是数据线连电脑那端）给单片机下载程序，所以我们需要 CH340 驱动。这个我们已经在光盘中为大家准备好了。

打开光盘中的软件->实验板 USB 转串口 CH340 驱动->CH340SER.EXE

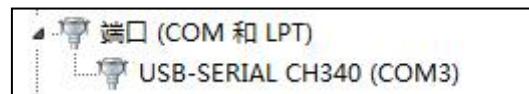
按安装



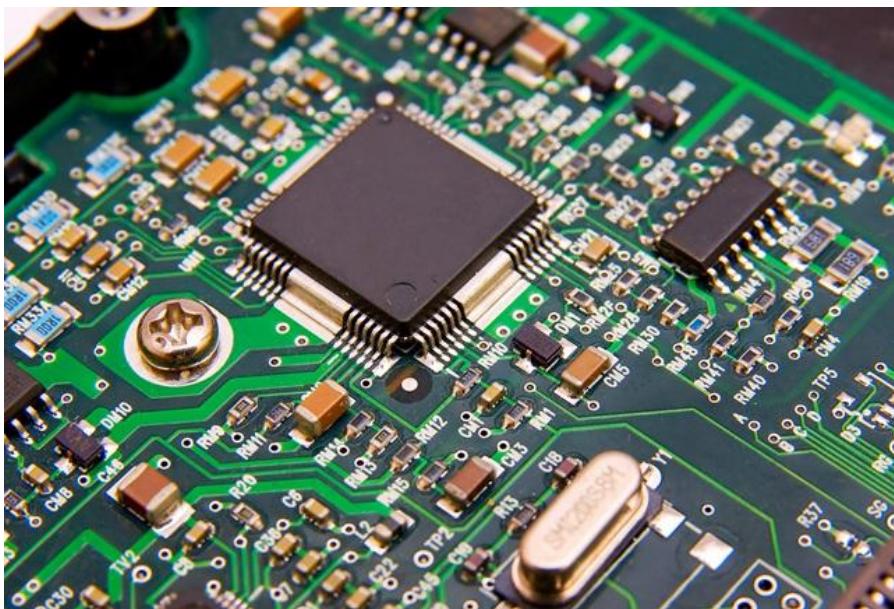
提示安装成功即可



链接实验板, 右击打开我的电脑->属性->设备管理器（此步因系统而异，不过最终都是要找到设备管理器）。找到端口处有该栏，说明驱动安装已经没问题了。



二. 单片机基础知识——关于单片机的八卦事



好了，板子我们已经认识了，软件我们也安装好了，大家是不是摩拳擦掌，想马上动手编写我们第一个单片机程序呢。但别着急，为了让大家在接下来玩单片机中少点困惑，这里我必需为大家简单讲解一下一些关于单片机的八卦事。掌握这些必备概念，对接下来我们玩单片机帮助非常大。

3.1 浅谈 51 单片机

关于单片机，这里我就不浪费篇幅去介绍它那很长很长的发展史。有兴趣的同学可以自行上网搜索，相信网上的资料肯定比我写出来的更加详细。

单片机（micro controller unit）绝对不是特指一种 51 单片机，它和计算机一样，经过漫长的发展和改进，有这各种各样的型号。单片机全称单片微型计算机。说白了，它就是一台计算机，有 CPU、ROM、RAM 等硬件。但它很小，所以适合完成各种控制任务。日常生活中的产品只要涉及控制的，上到空调、汽车，下到遥控、摄像头，十有八九都要依赖单片机来实现。

其中我们的主角——51 系列单片机以其优越的性能从茫茫“机”海中脱颖而出，在几十年间经久不衰，称为一代经典。

而 51 单片机又并不是指某一款单片机，它是对所有兼容 Intel 8031 指令系统的单片机的统称。最早的 51 单片机是由 Intel 公司于 1976 年推出的 MCS-48。后来 Intel 授权数个半导体公司生产设计 51 单片机。各个公司又按照需求在原来的 MCS-48 上加以改进，衍生出各种 51 单片机型号，使得 51 家族迅速发展壮大。如今 51 单片机发展出 60 多种的型号，可应对用户不同的需求。其中，国内由 STC 公司推出的 STC89C52 更是经典中的经典。凭借性能和价格的优势，市面上绝大多数教学用实验板都采用 STC89C52。但毕竟科技是不断进步的，为了更能让大家更能体验到单片机的乐趣，我们并没有随大流采用 STC89C52，而是采用功能更强大的改进型 STC12C5A60S2（简称 STC12）。



STC89C52 (DIP 封装)



STC12C5A60S2 (QFP 封装)

3.2 计算机语言

C Language

一般地，51 单片机的编程用到汇编语言或 C 语言。虽然对于同一个目标，用两种语言都可以实现，但考虑到实用性，绝大多数情况下 51 单片机编程会首选 C 语言。C 程序经过编译器编译，变成只有 1 和 0 的机器语言，再烧录进单片机，驱使单片机工作。当然，同样的工作你也可以用汇编来完成，而且用汇编语言编写出来的程序效率

更高。而且一些情况下，如底层代码的编写、一些头文件的编写等等也会用到汇编语言。但我们现在注重学习单片机的应用层，C 语言的通俗易懂、语言层次明了清晰，非常利于我们入门单片机。因此接下来我会用 C 语言带领大家玩转 51 单片机。**所以特别注意！本教程的编写是立足于你已经对 C 语言有一定的基础之上，而对于还没接触过 C 语言的同学可以边学习 C 语言，边学习本教程。**

51 单片机使用的 C 语言，又不同于大家之前所学的标准 C 语言，而是 C 语言的“方言”版本——C51。C 语言运行于普通的 PC 平台，而 C51 运行于单片机平台。听到这里，你是不是心寒了一下，心想是不是又要学一门新的计算机语言（的确对于一些同学来说，计算机语言的学习不是简单地的事）。但无需担心，C51 传承自 C 语言，在语法和结构上基本一致。不同的，只是 C51 用到的头文件不同，增加了几个新的数据类型等，这些在接下来的玩耍中我们会逐一接触到。相信有进取心的你，可以轻松掌握 C51！

3.3 数制

日常生活中，我们最常用到的是十进制，但机器只识别二进制。后来为了简化二进制的表达，又出现了十六进制。由于 51 单片机内部结构（这些以后会详细说明），编程时会经常用到 8 位的二进制代码。熟练掌握十进制、二进制和十六进制的转换，对以后的编程是大有裨益的。关于十进制转换为二进制这里就不加赘述（这是高中知识，就不浪费大家时间了，如果忘记的同学可以网上复习），我们就重点学习一下 8 位二进制到十六进制的转换。

将 8 位二进制从高位到低位依次从左到右排列，从正中间截取得到低四位（右侧）和高四位（左侧），再分别把低四位和高四位转换为十六进制。举个例子：

二进制：01011101，高四位 0101 对应十六进制为 5，低四位 1101 十六进制为 D，得 01011100 十六进制为 5D。

值得注意的是，一个十六进制数转换为二进制后是占 4 位的！如果是十六进制的 1 转换为二进制是 0001。Windows 系统自带的计算器大家可以在“查看”中设置成程序员模式。用它大家可以在编程中快速完成数制的转换。但在十六进制转换成二进制时，计算器会自动隐藏了前面的 0。如十六进制 1 转换成二进制它只会显示 1，这时候大家不要忘记在前面用 0 补全 4 位。

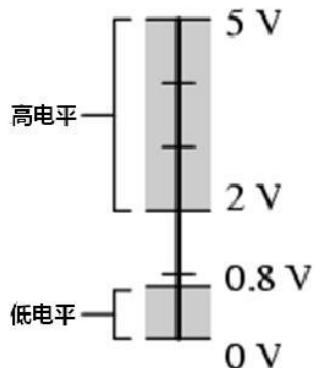
当你获得 1 个 8 位二进制时，**注意它是从高位到低位依次从左到右排列的！**

3.4 TTL 电平信号

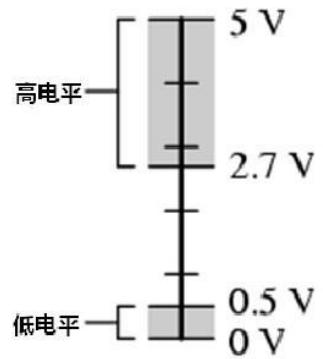
大家会不会疑惑，单片机只认 1 和 0，那怎样让单片机辨认 1 和 0 呢？为了达成共识，对于 51 单片机，我们引入一个公认的标准——TTL 逻辑电平（三极管-三极管逻辑电平，transistor transistor logic）信号系统。为什么我们要特定说明 51 单片机呢，这是因为 51 单片机内部的逻辑电路是由 TTL 门电路构成的。那相对应的，也有的单片机内部是由 CMOS 门电路构成，例如 STM32，那这类单片机就要采用 CMOS 逻辑电平了。TTL 逻辑电平信号系统说是系统，其实大家只需简单记住最基本的两条：1、+5V 电压等效为“1”，为高电平；2、0V 电压等效为“0”，为低电平。

但电平其实是一个电压范围，并不一定说+5V 才为高电平，0V 才为低电平。不同单片机对于高低电平的划分有不同的标准。对于本教程使用的增强版 51 芯片 STC12，输入信号时，大于+2V 即为高电平，等效为“1”；小于+0.8V 即为低电平，等效为“0”。而输出信号时，大于+2.7V 即为高电平，等效为“1”；小于+0.5V 即为低电平，等效为“0”。如下图：

TTL电平输入信号阈值



TTL电平输出信号阈值



对于既不是高电平又不是低电平的电压区域，叫浮空状态。这种状态是不稳定的，单片机有时认为是高，有时认为是低。所以我们应该避免电路中出现浮空状态。

总结，虽然+5V 为“1”，0V 为“0”，但反过来“1”就不是+5V，“0”就不是0V 了。在一些电路分析中，我们往往需要用电表来测量某处的电压值来判断 TTL 信号的高还是低，因此我们要熟记 STC12 的电平特性。

3.5 寄存器与特殊功能寄存器

寄存器，顾名思义，寄存器（register）就是用来存放东西的器件。其实，说寄存器是器件并不准确。在数字电路中，我们将用于存储一组数据的存储电路叫做寄存器。

单片机执行各种各样的工作所需的指令或数据都存取于寄存器，可见寄存器是单片机能够完成各种功能的重要基石！而特殊功能寄存器（SFR，special function register）则是单片机中各种各样用途广泛的寄存器，他不但可以存放数据，而且还连带更复杂的逻辑电路，用来实现各种各样的功能。单片机的很多工作都离不开特殊功能寄存器，我们就简单看看传统 51 系列单片机有哪些特殊功能寄存器：

符号	名称	地址
ACC*	累加器	E0H
B*	B 寄存器	F0H
PSW*	程序状态字	D0H
SP	栈指针	81H
DPTR	数据指针	
DPL	数据指针低字节	82H
DPH	数据指针高字节	83H
P0*	端口 0	80H
P1*	端口 1	90H
P2*	端口 2	0A0H
P3*	端口 3	0B0H
IP*	中断优先级控制	0B8H
IE*	中断使能控制	0A8H
TMOD	定时器/计数器模式控制	89H
TCON*	定时器/计数器控制	88H
T2MOD	定时器/计数器 2 模式控制	0C9H
T2CON*	定时器/计数器 2 控制	0C8H
TH0	定时器/计数器 0 高字节	8CH
TL0	定时器/计数器 0 低字节	8AH
TH1	定时器/计数器 1 高字节	8DH
TL1	定时器/计数器 1 低字节	8BH
TH2	定时器/计数器 2 高字节	0CDH
TL2	定时器/计数器 2 低字节	0CCH

RCAP2H	T/C2 截获寄存器高字节	0CBH
RCAP2L	T/C2 截获寄存器低字节	0CAH
SCON*	串行控制	98H
SBUF	串行数据缓冲器	99H
PCON	电源控制	87H

表 1.2.1 传统 51 单片机寄存器

可见，每个特殊功能寄存器在单片机中都是有相应的地址的。凡是占 8 位的特殊功能寄存器，在编程中我们都可以直接操作而无需声明（详细的我们会在入门篇第二节讲到）。其中，带 “*” 号的特殊功能寄存器，我们在编程中可以直接操作它的某一位。虽然这些寄存器琳琅满目，我不急着给大家解释它们的功能，也不要大家把它们都背下来，我们会在以后的运用中慢慢了解它们。

另外说点小细节。在寄存器地址中我们用到 xxH 这样的表示方式。其实这是汇编语言表示十六进制的方式。在 C51 中，我们表示十六进制是在前面加 0X，例如 0X8E。

3.6 从 STC89 过渡到 STC12

这部分是写给对 STC89 型 51 单片机有了解的同学看的。如果是单片机刚入门的同学可以直接跳过。

STC12 对 STC89 有几点不同，可能会影响到接下来的编程，所以要在这里特别说明一下：

- ① 增加了 P4 口。原来大家用的头文件 “reg52.h” 没有对 P4 进行定义，因此编程时如果用到 STC12 的 P4 口是就会编译不通过。这就是我们为什么要加入头文件 “STC12.H” 的原因了。 “stc12.h” 里有对 P4 口以及一些 STC12 相对于 STC89 新增的一些特殊功能寄存器的定义；
- ② 省去里定时器 2。因此如果想用 T2 产生波特率就请改用独立波特率发生器产生波特率；
- ③ 指令执行速度全面提升。最快的指令快 24 倍，最慢的指令快 3 倍。另外 STC89 单片机是 12T 单片机，就是 12 个机器周期等于 1 个时钟周期；而 STC12 是 1T 单片机，即 1 个机器周期等于 1 个时钟周期。这样一来，一些用 for 或 while 等函数实现的延时函数就要自己重新调整了。但定时器仍默认为 12T 模式，而你也可以通过寄存器设置成 1T 模式。

入门篇

做好充分的准备工作后，我们就正式进入单片机的世界了。单片机能干什么？学单片机会不会很难？可能你还怀揣着这些不安的想法。别担心，我们将一步紧跟一步，从最基本的模块开始了解单片机。马上，我们将通过操作最常见的 LED 灯开始，来初探单片机的独特魅力！

一、点亮 LED——单片机中的“HELLO WORLD！”



图 2.1.1 直插式 LED

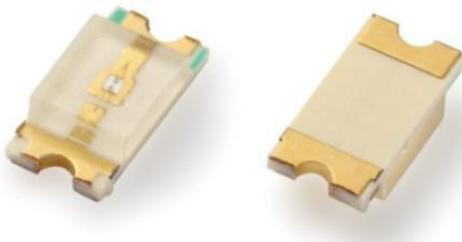


图 2.1.2 贴片式 LED

还记得你的第一个 C 语言程序吗？一般大家编写的第一个 C 语言程序都是在屏幕上显示“HELLO WORD!”，因为它最简单，最容易上手。在单片机中，我们也是从最简单、最基本的东西开始玩起。下面，我们就从 LED 和蜂鸣器开始，一起步入单片机的世界！

1.1 硬件准备

Light Emitting Diode，发光二极管，也就是我们日常说的 LED。LED 是有极性的，我们正向给 LED 一个适当的（太低 LED 会不亮，太高 LED 会烧坏）电压，LED 就会发光。

而蜂鸣器是一种有源发声器件。它运用我们最常见震动发声原理。蜂鸣器也是有极性的，只要我们正向给蜂鸣器一个适当的电压，就可以驱动蜂鸣器里面的震动元件发出声音。

不管是 LED、蜂鸣器，还是实验板上各式各样的元器件，都是集成在印刷电路板 PCB 上的。PCB 上的电路是被覆盖着的，那我们连元器件的电路情况都没搞懂，何谈控制它们呢？所以单片机的学习，永远离不开电路原理图的辅助。这就像你来到一个陌生的地方需要地图来指引一样，电路原理图可以让你看到电子元件与单片机的连接情况，这将指引你编写正确的程序。

打开光盘中的 1.实验板资料 -> 1.傀儡师原理图

分别找到 LED0 和蜂鸣器的位置，观察它们的电路连接：

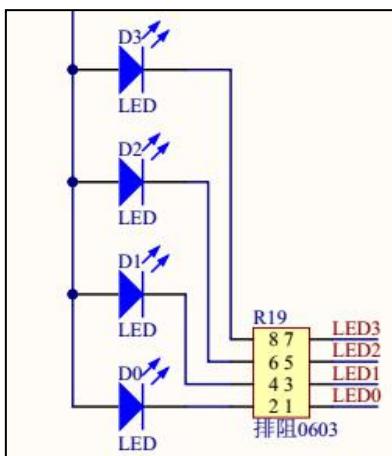


图 2.1.3 LED0 电路连接

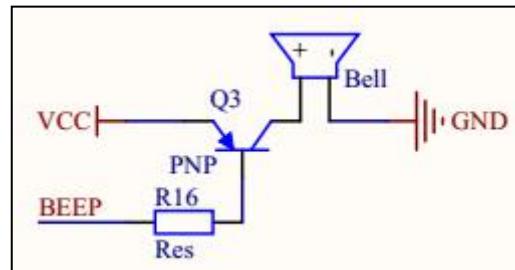


图 2.1.4 蜂鸣器电路连接

在原理图中，如果两个元件上的引脚上的编号相同，就说明在实际中是相连的。我们在看看单片机所在的位置：

外接信号	板内信号	显示类信号	IO 口	U1
BEEP			P1.5	1 MOSI/ADC5/P1.5
COMP1			P1.6	2 MISO/ADC6/P1.6
COMP2			P1.7	3

图 2.1.5 单片机与蜂鸣器相连引脚

VCC	37	P0.0	LED0
P0.0	36	P0.1	LED1
P0.1	35	P0.2	LED2
P0.2	34	P0.3	LED3
P0.3	33	P0.4	LED4
P0.4	32	P0.5	LED5
P0.5	31	P0.6	LED6
P0.6	30	P0.7	LED7
P0.7	29		

图 2.1.6 单片机与 LED 相连引脚

可以看到，LED0~LED3 接 1 个排阻，LED4~LED7 接 1 个排阻。排阻其实就是并排一起的电阻，这里就是相当于 8 盏 LED 灯各接了 1 个电阻。LED0 正极串联 1 个保护电阻后接到 VCC（即单片机的工作电源，51 单片机工作电源为 +5V），负极与单片机 P0.0 相连接；蜂鸣器正极接三极管集电极，负极与实验板的地相连接。三极管基极接单片机 P1.5，发射极接 VCC。

1.2 知识点讲解

LED0 和三极管基极分别与单片机的 P0.0 和 P1.5 相连。细心的你也一定发现，原理图中单片机引脚中还有很多 PX.X 的引脚。这些引脚，都会单片机与外界联系起来的重要“桥梁”——I/O 口。

I/O 口，英文全名为 Input/Output。字面就可以看出，他们是单片机信号输出和输入的“通道”。通过软件，它可以被设置为高电平（但电压比工作电源 VCC 稍低）和低电平（0V）。单片机复位后（即单片机回到初始的默认状态。单片机复位有两种方法：重新上电和按复位按键），51 单片机 I/O 口默认是高电平。单片机的大部分工作，都是通过控制 I/O 口来实现的。可想而知，I/O 口是单片机开发的基石。

我们实验板搭载的是升级版 51 单片机 STC12，它有 5 组 I/O 口：P0、P1、P2、P3、P4，每组 I/O 口有 8 位。以第一组 I/O 口为例，依次为 P0.0、P0.1、P0.2、P0.3、P0.4、P0.5、P0.6、P0.7。**注意！不管是组还是位，都是从 0 开始的！**

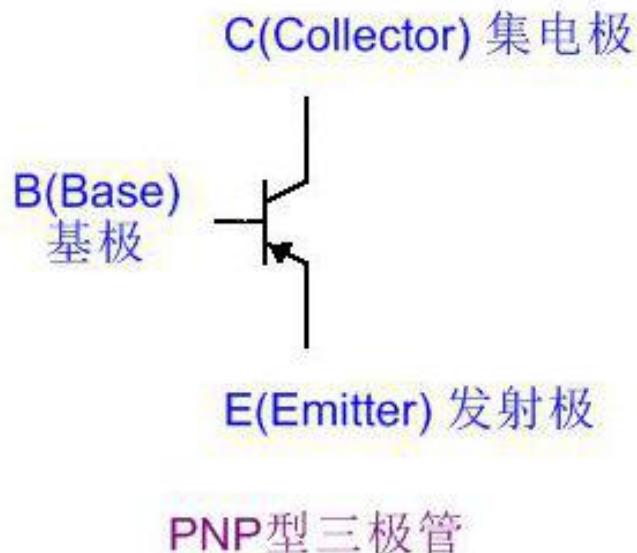
我们的 st12 有 44 个引脚。除了 40 个 IO 口外还有单片供电引脚 VCC、地线 GND 和两个晶振输入引脚 XTAL1 和 XTAL2。从 1 到 44 引脚对应的功能可以在实验板原理图和芯片数据手册中查看。而在实物中，我们正对芯片，使小圆点位于左下方。小圆点下方第一个引脚为第 1 引脚，然后逆时针一直数过去就是 2 至 44 引脚了。这种识别引脚的方法适用于绝大多数芯片。



2.1.7 贴片 STC12 近照

让我们再回顾原理图中 LED0 和蜂鸣器的电路连接。LED0 正极串联 1 个保护电阻后接到 VCC，负极与单片机 P0.0 相连接；蜂鸣器正极先通过三极管集电极，三极管发射极接 VCC，基极接单片机 P1.5，蜂鸣器负极与地相接。因为 I/O 口默认是高电平，所以，默认情况下 LED0 正负极之间和三极管发射极和基极之间没有足够的压降，都不会工作。如果我们想让两者工作起来，就得把两者的控制端给拉低，即通过软件把 P0.0 和 P1.5 设置为低电平。这样一来，LED0 正负极之间和三极管发射极和基极之间就有足够的压降来驱动他们工作了！

对于没有模电知识的同学，我就简单说一下三极管在这里的作用。这里我们采用的是 PNP 三极管，他三个引脚分别为基极（B）、发射极（E）和集电极（C），如下图所示：



PNP型三极管

2.1.8 PNP 三极管简图

三极管在这里充当开关的作用。为什么我们不像 LED 一样，正极接 VCC，负极接 I/O 口来控制蜂鸣器呢？原因在于，51 单片机的 I/O 口最大负载电流只有几毫安，而要驱动蜂鸣器工作，电流至少要有 20mA。如果我们直接把蜂鸣器正极接 VCC，负极接单片机 I/O 口，那蜂鸣器工作时电路中电流大于 I/O 的最大负载电流，就有可能烧坏单片机。

所以我们采用 PNP 三极管充当开关（这有点类似继电器）。通过上图我们看到，发射极有个箭头指向基极。因为发射极接 VCC，基极接 P1.5，当 P1.5 被拉低时，三极管发射极和基极之间有足够的压降驱动三极管工作。当三极管工作时，电流就从发射

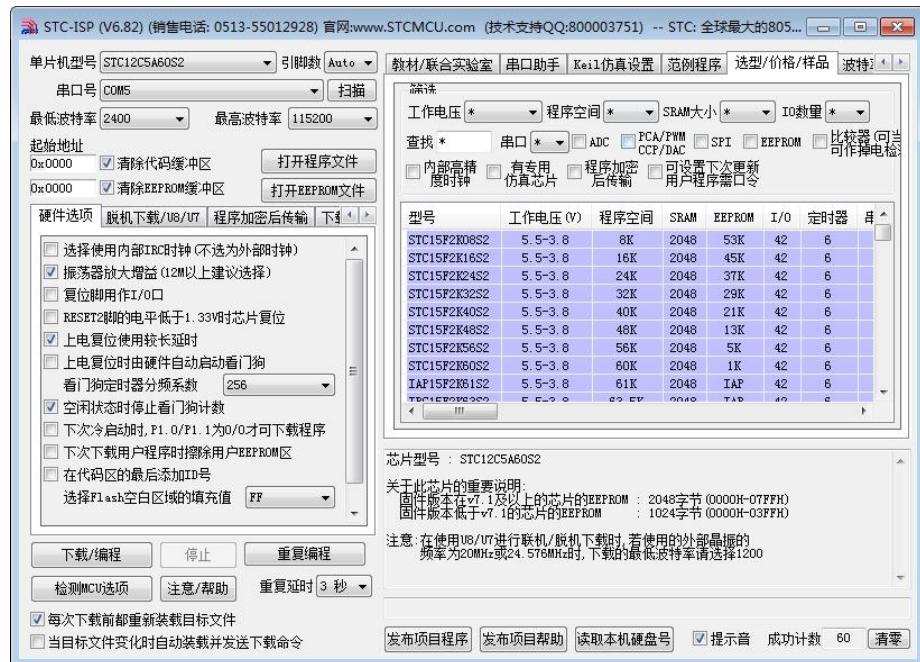
极流向集电极，从而驱动蜂鸣器工作，避免过大的电流流向单片机。另外三极管也有电流放大的作用，增强驱动蜂鸣器的能力。

1.3 软件实现

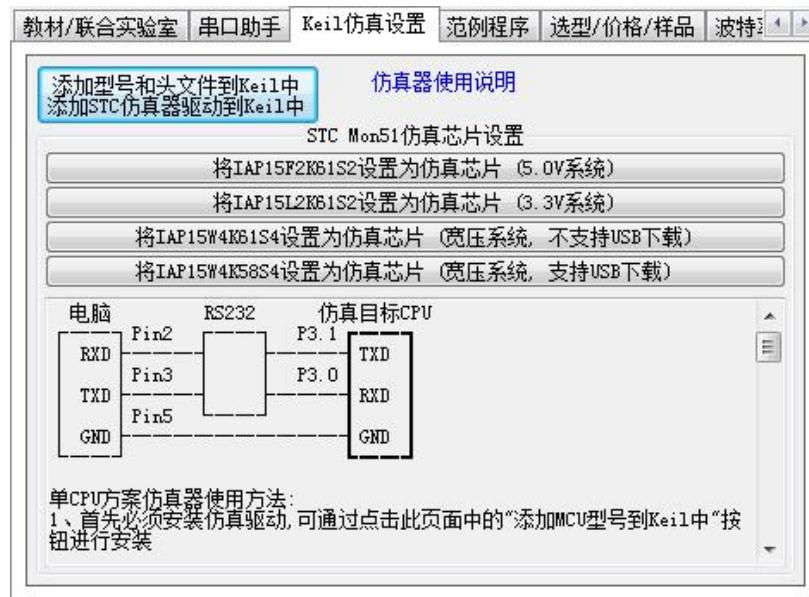
因为大家是第一次使用 Keil4，我就一步步教大家编写第一个单片机工程，让 LED0 和蜂鸣器工作起来！

首先，我们先要将 STC 芯片库添加到 Keil4 中。

我们打开光盘中的->软件->烧录 STC 单片机软件 stc-isp-> stc-isp-15xx-v6.82.exe



在右上方找到 Keil 仿真设置。

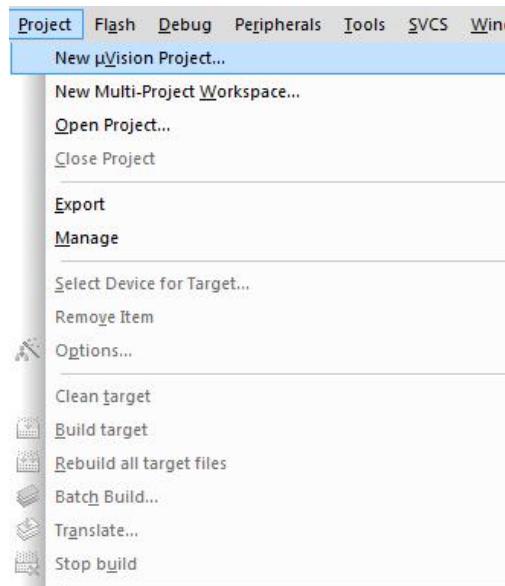


按“添加型号和头文件到 Keil 中”，找到我们安装 Keil4 的目录（就是你安装 Keil4 的地方，默认安装则在 C 盘内）。

按确定后完成芯片库的导入。

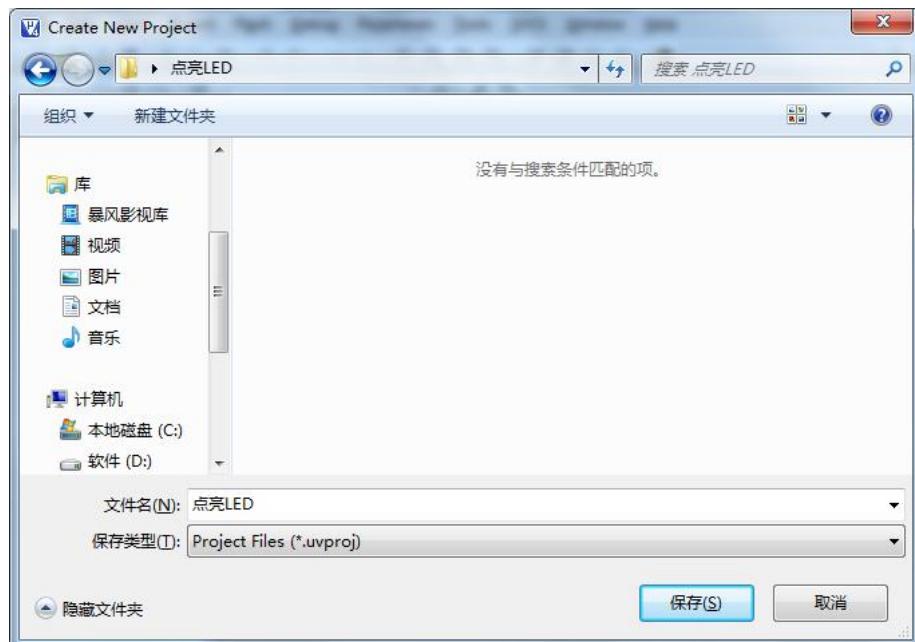


我们打开已经安装好的 Keil4，在上方选项栏中找到 Project，点击 New μVisI/On project 新建 1 个新的工程。

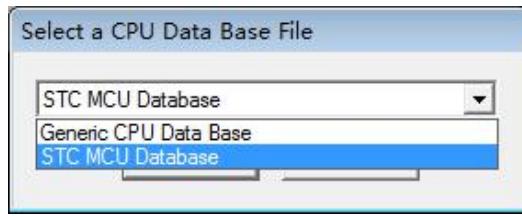


你需要先创建一个文件夹，并将你创建的工程放置在里面。文件夹和工程的名字，应该与你要实现的目标相符，这样以后才能轻松找到你很久以前编写的工程。**这是一个非常重要的习惯！**

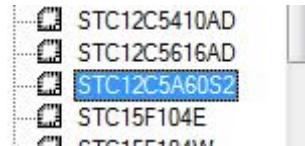
这里我们给文件夹和工程命名为“点亮 LED”。



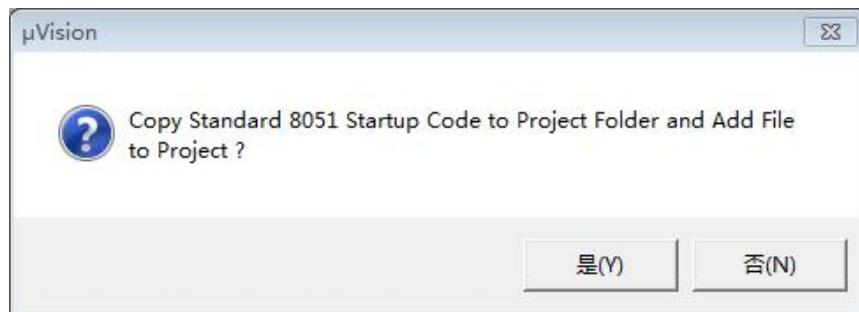
接下来是选择芯片的型号，我们要选择 STC 的库。



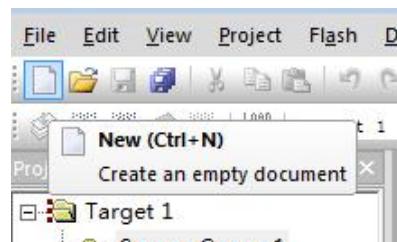
然后选择 STC12C5A60S2



选择芯片型号后，会弹出信息框询问是否加入启动文件，按否。



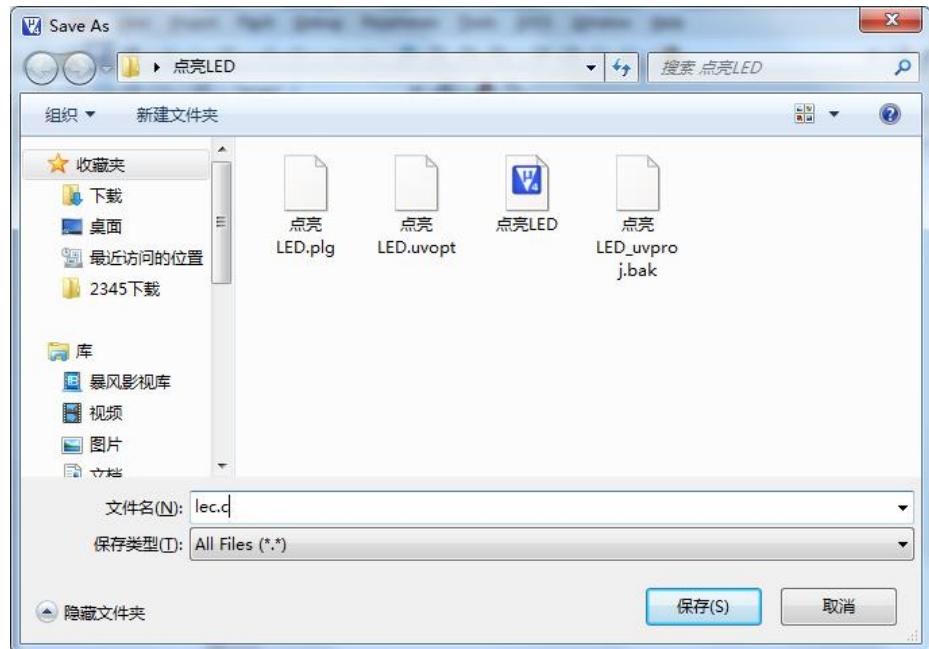
到这里，你只是创建了 1 个工程，但工程里还没有任何程序代码。所以，我们要创建 1 个 C 语言文件。左击 File 下方的 New。



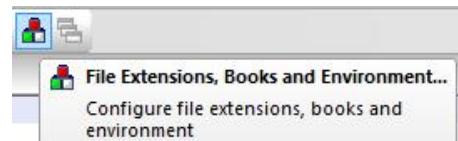
左击旁边的保存。



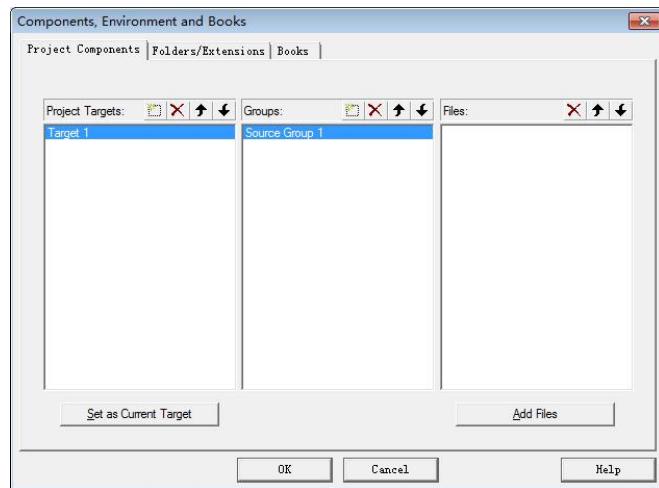
人有名，树有皮。程序文件的名字应当简介明了地反映程序的功能。**不过这里要注意！**文件名要你自己添加后缀，因为我们是用 C 语言编程的，所以后缀为.c。这里，我们取 led.c。



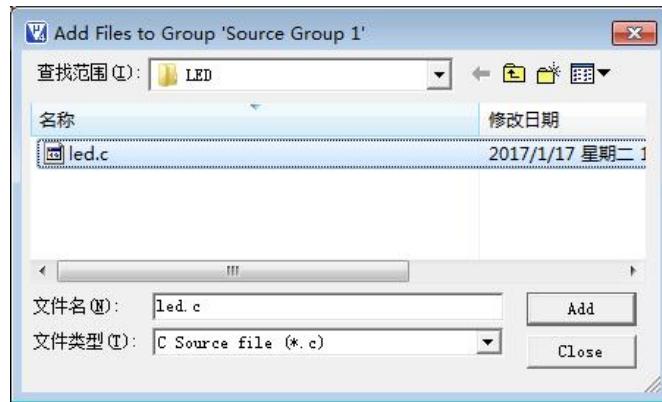
然后我们要把 led.c 文件添加到工程当中。电机上方一个积木状的图标，打开工程管理。



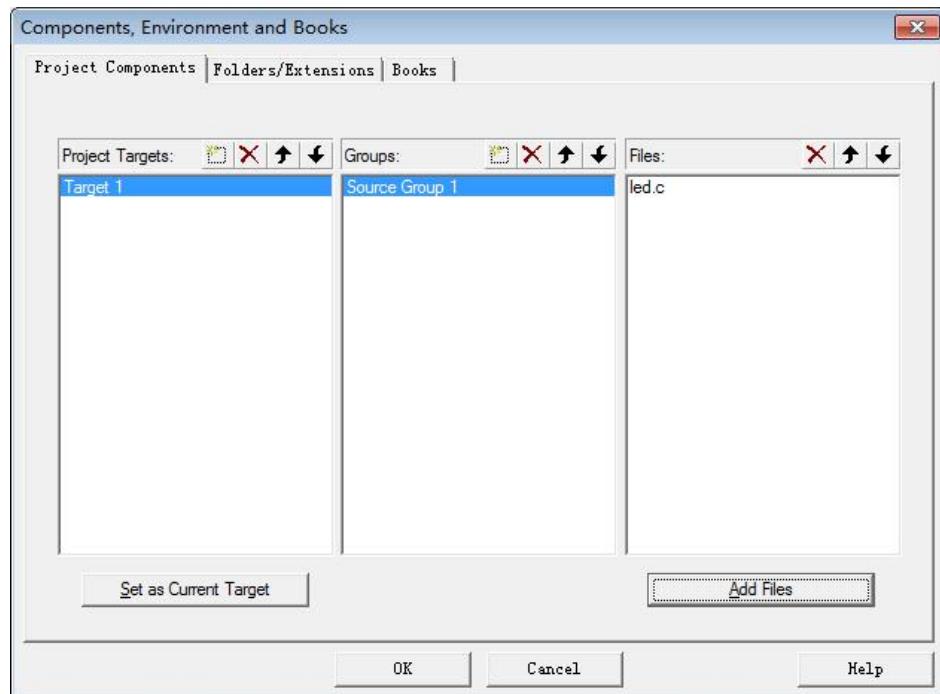
左击右下方的 Add Files。



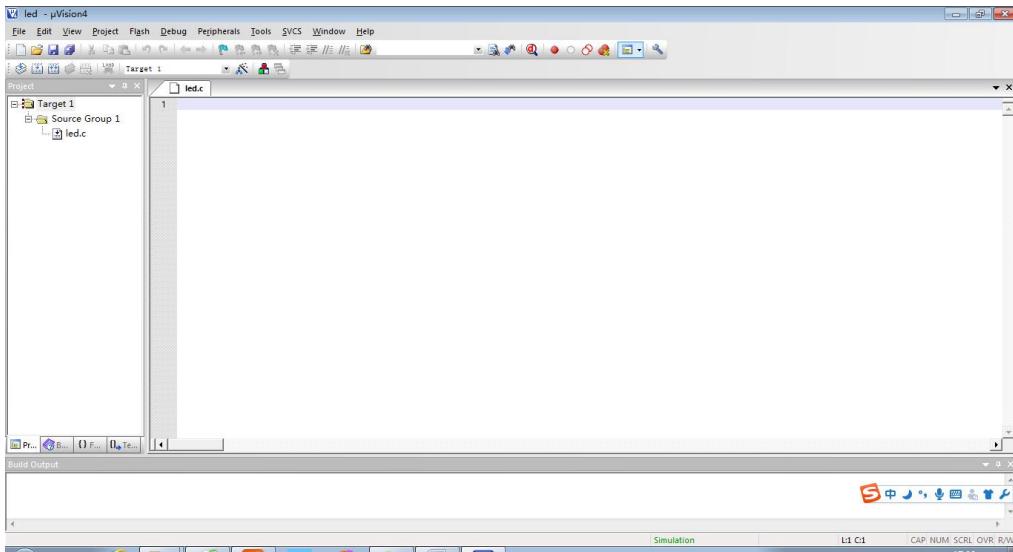
选取我们之前保存的 C 文件，按 Add。注意按 Add 后，对话框并不会消失，需手动按 Close 取消，Add 只需按一次即可。



关闭对话框后，我们可以发现 led.c 已经在最右侧的框中的，说明它已经添加到 Source Group1 中了。当然，Source Group1 的名字是可以修改的，只需左击中间框中的 Source Group1，修改后按其他地方即可。然后 Cancel 退出。



最后如下图



虽然以上步骤有点繁琐，但都是建立一个新工程的必要过程，每一步都不可或缺！而且凡事都是熟能生巧，相信只要你多练习几次，就能轻松建立一个自己的工程了。

到这里，前期工作就算完成了。接下来就让我们编写第一个单片机 C 语言程序。在 main.c 文件中写入下列代码：

```
#include "stc12.h"

sbit led0 = P0^0;//sbit 为特殊功能寄存器位变量

void main()
{
    led0 = 0;//点亮 LED0
}
```

有没有发现，这些都是 C 语言的语法和结构，但又不完全是。

首先，头文件 “stc12.h” 是 stc12 单片机编程专用的头文件，它是专门为单片机编程而编写的头文件。它的功能我们在下一节再讲。

其次，一个新的变量类型：sbit，它是 C51 特有的数据类型。在 C 语言里，如果直接写 P0.0，C 编译器是不能识别的。而且 P0.0 也不是一个合法的 C 语言变量名，所以得给它另起一个名字，这里起的名为 led0。可是 led0 是不是就是 P0.0 呢？你这么认为，但编译器可不这么认为。所以必须在它们之间建立联系。这里就使用了 C51 的关键字 sbit 来定义。P0^0 表示第一组 I/O 口 P0 的第 0 位，即原理图上的 P0.0，这种表达方式是既定的。因此，sbit led0 = P0^0 的含义就是将单片机的 P0.0 口命名为 led0，大家使用 led0 就等于使用 P0.0 口了。

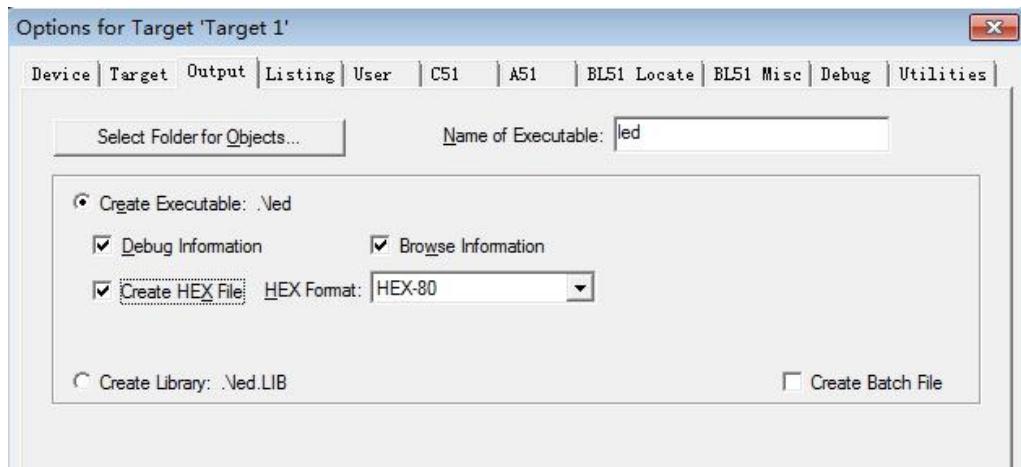
我们上面分析过，因为 LED0 正极串联电阻后与 VCC 相连，正极与单片机 P0.0 相连，只要 P0.0 为低电平，就能点亮 LED0。所以在程序中，只要将 led0，即 P0.0 置 0，既可以把 P0.0 置为低电平。反正，将 led0 置 1，就把 P0.0 置为高电平。

现在，我们已经编写好我们需要的 C 语言程序了，那能不能直接把这个文件直接烧录进单片机呢？显然是不可以。C 语言是高级计算机语言，但计算机只认“1”和“0”。于是，我们就要编译生成单片机可以“看懂”的 hex 文件。

首先，在上方工具栏中找到设置目标选项工具。左击打开



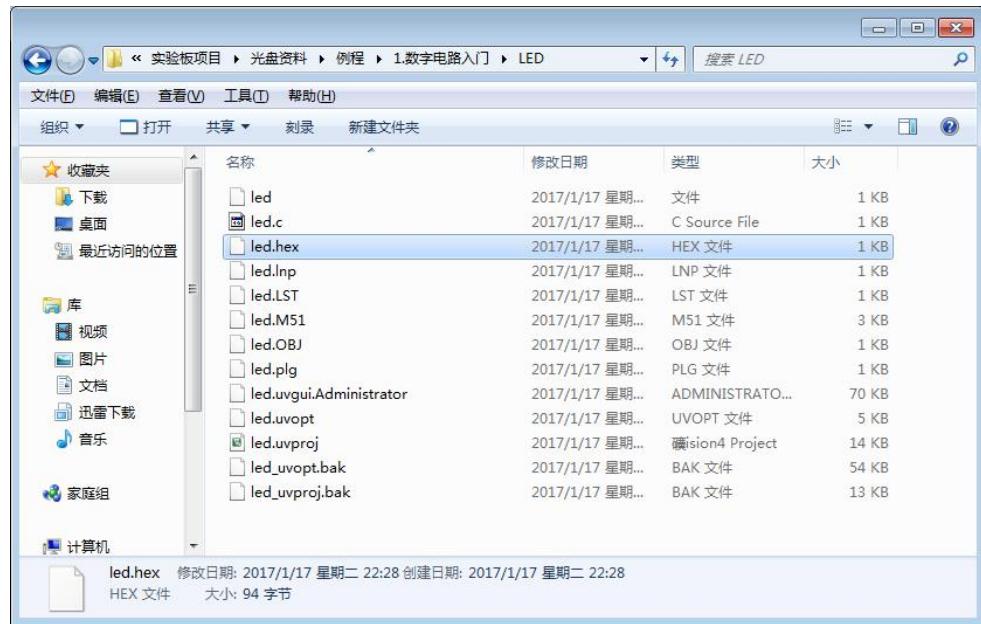
在对话框中选中 Output 页，勾选 Create HEX File 生成 hex 文件。



OK 确认后，在上方工具栏中找到编译按钮，就是图中中间按钮，左击编译。



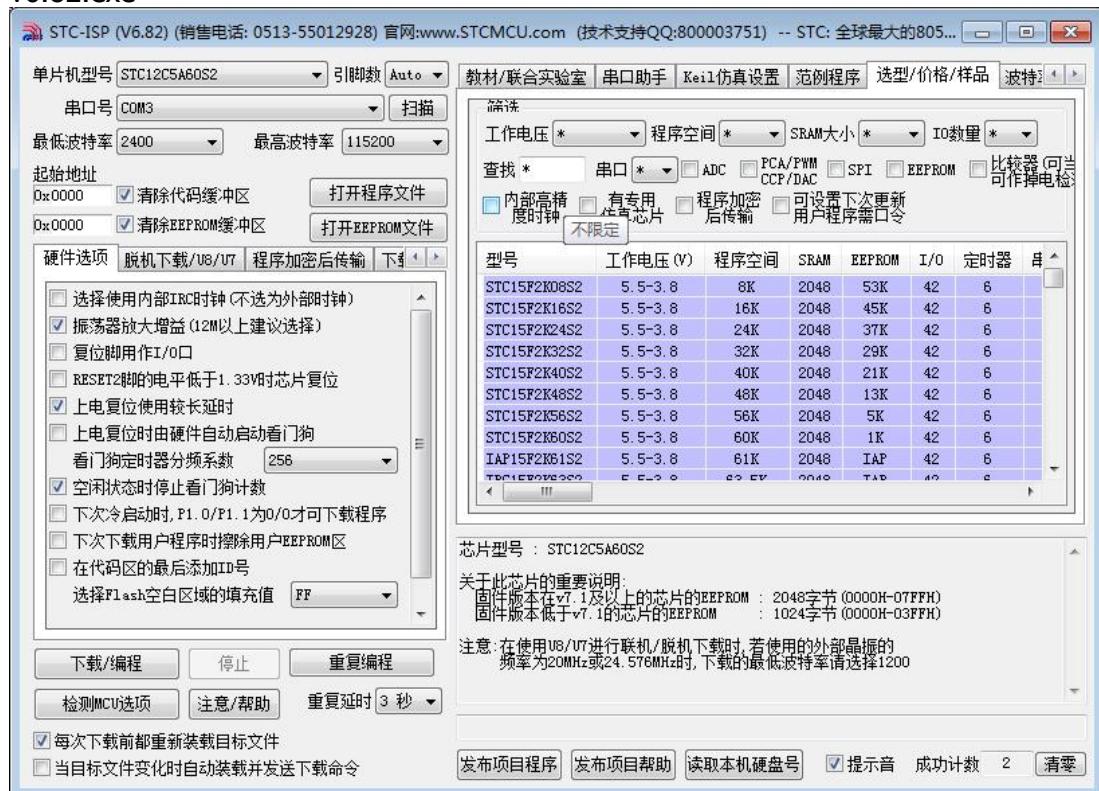
打开工程所在文件，就可以发现一个后缀为 hex 的文件了



1.4 下载验证

最终的最终！我们离目标只有一步之遥了！接下来，只要我们把工程编译生成的 hex 文件烧录进单片机，就能点亮 LED0 了！

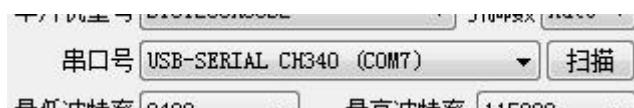
打开光盘中的 3.软件->3.烧录 STC 单片机软件 stc-isp->stc-isp-15xx-v6.82.exe



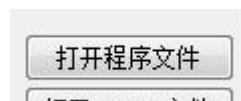
在下载之前，我们要在几个地方做设置。首先是单片机型号设置，**我们要选择 STC12C5A60S2**。STC12C5A60S2 是选在 STC12C5A60S2 系列里的 STC12C5A60S2。因为型号有点多，所以要细心点查找。如果你在接下来的学习中一直使用我们的实验板，以后烧录就可以跳过这步。



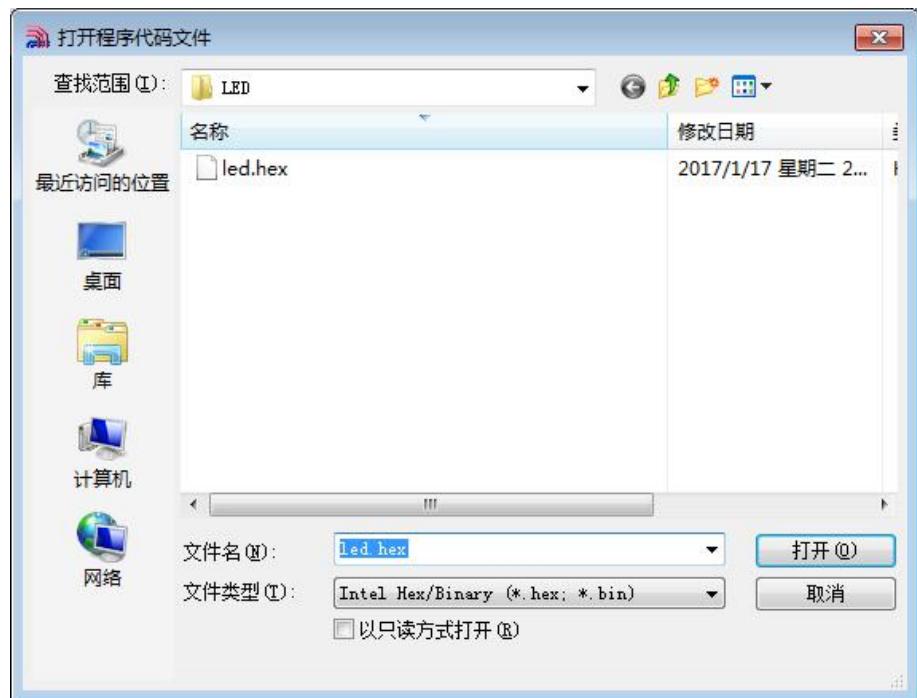
然后串口设置。一般情况下，电脑连接单片机后，软件会自动检测串口。如果软件检测不到，可以手动扫描。



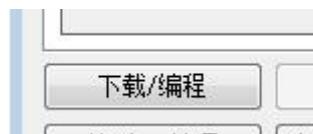
选择要下载的文件。



就是我们刚刚生成的 hex 文件。



在单片机保持通电的情况下，点击下载。

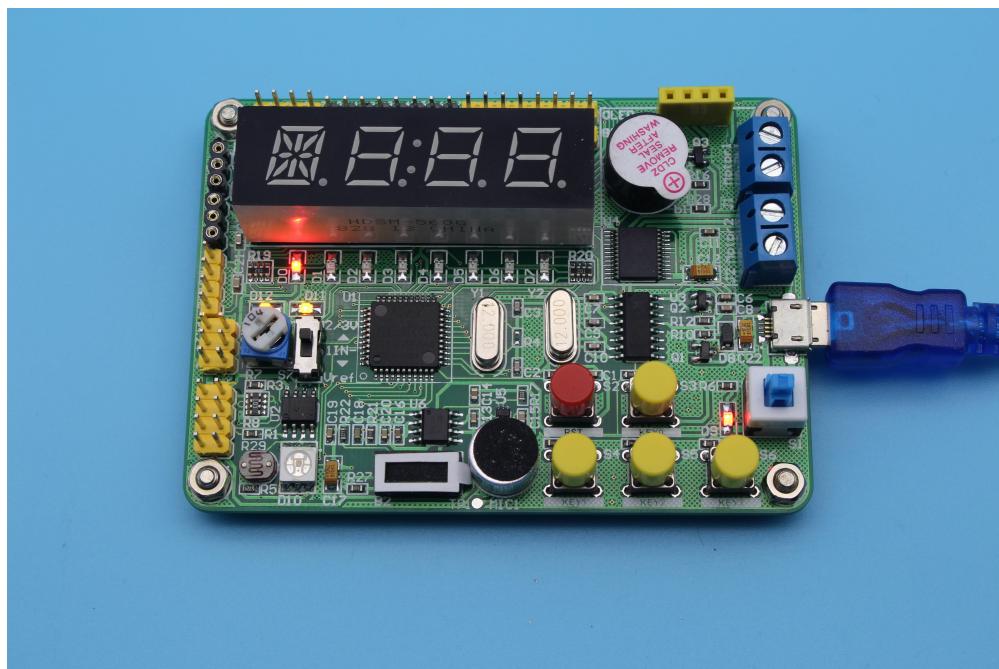


可以发现右下侧的对话框一直报告下载情况，等出现如下图时，说明下载已经成功了。



回头看一下我们的实验板，LED0 是亮的！说明我们成功了！

特别声明：因为下载电路是我们自己研发的，所以第一代傀儡师实验板的下载稳定性稍有欠缺。如果点击下载后没反应，可以按停止键，再按下载。一般第二次下载就会成功。如果多次下载失败，可以尝试先停止下载，长按红色复位键按钮，点击下载。点击下载 0.5 秒后松开红色按钮，就能成功下载。在以后的版本中我们一定会对下载功能加以完善！



1.5 思考总结

不知道你有没有想过一个问题：同一个程序，换一个实验板能实现相同的功能吗？

首先我们回顾一下我们这节是怎样实现点亮一盏 LED 的。我们在介绍完 LED 和蜂鸣器后，马上就讲到了实验板的电路原理图。我们就是以电路原理图为依据，来编程实现点亮 LED 的。由此可见，对于一个单片机控制系统，单片机程序是要建立在这个系统电路原理图的基础上的。

可想而知，每一块单片机实验板因设计者的不同，它的电路设计都不尽相同。对于我们这块实验板来说，LED0 负极接单片机 P0.0，但别的实验板可能接的是 P2.0。所以单片机编程要和电路原理图时刻紧密结合。

那换个板子，程序都要重新编写吗？完全不用——这多亏了 C 语言有良好的移植性。如果我们要将程序移植进另一个单片机实验板（当然，前提是前后单片机型号相同或近似），我们只需对照着前后两者的电路图，对程序中的配置稍作修改即可。

二、流水灯——流动闪烁的 LED

在上一节课中，我们通过软件将 P0.0 拉低来实现点亮一盏 LED。但只点亮一盏 LED 是不是显得很单调？接下来，我们就尝试着点亮多盏 LED，并让实验板上 8 盏 LED 流动闪烁起来！

2.1 硬件准备

嵌入式开发永远离不开电路原理图。同样然我们打开原理图，找到 LED 线性阵列的位置：

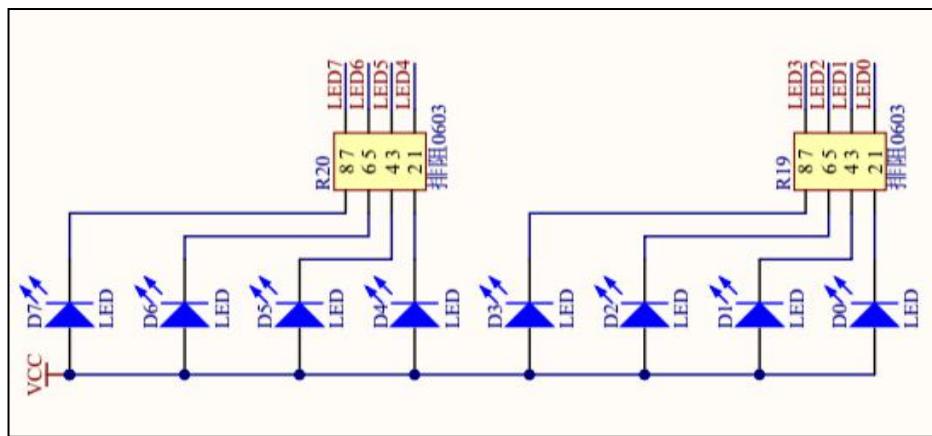


图 2.2.1 LED 线性阵列电路连接

实验板一共有 8 盏 LED，分别位 LED0~LED7，在板子上用 D0~D7 来表示。每个 LED 正极均与 VCC 相连。至于 LED 的负极，LED0~LED3 接 1 个排阻、LED4~LED7 接 1 个排阻后与单片机 P0.0 到 P0.7 相接。

接排阻是起到保护 LED 的作用。上一节介绍 LED 时我们提到，要使 LED 工作，就要给 LED 一个适当的正向电压。如果太低，LED 将不会工作；太高将烧坏 LED。由颜色不同，一般 LED 的工作电压从 1.8V 到 3.6V 不等。如果我们不加保护电阻，那当 I/O 口拉低时，LED 两端的电压将达到 5V，现象是 LED 闪烁一下后冒烟烧坏。保护电阻就是起到分压保护 LED 的作用。

2.2 知识点讲解

在上一课中，我们就通过直接操作 P0.0 来实现点亮一盏 LED。这种直接操作某一位的方法，我们称之为**位操作**。现在如果我们想做出流水灯——让 8 盏 LED 依次点亮，是不是同样可以通过位操作来实现呢。

答案是可以的。但是，这种方法明显笨拙且繁琐。试想一下，如果你要用位操作来完成流水灯，先要用 sbit 变量分别声明 8 个 LED，然后在主函数中依次将 P0 对应的位拉低。就这样，简单的流水灯就要用二十多行程序了。但是，如果我们用并行操作和_crol_ 函数（或_cr0r_ 函数），就可以轻松完成流水灯！

我们在前面学习过，一个十六进制数转换为单片机可识别的二进制占 4 位，两个十六进制数就可以表达 8 位二进制数，数量上刚好等于 P0 的 8 个位。我们就将两个十六进制赋给 P0，直接操作 P0 来完成 LED 的亮与灭。像这样直接给特殊功能寄存器赋值的操作，就叫**并行操作**。所谓并行操作，就像一行人并行排一起走一样，数据是并行一起输出的。以后遇到的一些特殊功能寄存器，只要它的位数是四的倍数，都可以直接操作它，并直接给它赋值。

遇到新事物多思考是一种良好的习惯。不知道你有没有思考过，为什么编译器能知道程序中的 P0 就是单片机中的特殊功能寄存器 P0 呢？单片机中的特殊功能寄存器 P0 它是没有名字的，它有的只是一个地址。之所以我们能在程序中直接用 P0，是因为在头文件中已经把它定义好了！

我们右击程序中第一行的头文件 “stc12.h” ，点击 Open document “stc12.h” ，打开头文件 stc12.h，如图所示：

```
09
10 #include "stc12.h"
11
12 sbit Led0 = P0^1;
```

打开后我们看到一段字里行间都透露着“你肯定看不懂我”的自信的代码。现在不需要你看懂，也不需要去背，你只需知道里面定义了 stc12 内各个特殊功能寄存器和部门寄存器各个位的位定义。通俗点说，引入头文件 stc12.h 就等于告诉编译器 stc12 里面有什么东西可以用。我们将在进阶篇第三节 PWM 中会做更详细的说明。所以引用该头文件后，编译器就可以识别我们在程序中用到的特殊功能寄存器，因此可以直接调用而无需再次声明 P0。

但是，即使是用了并行操作，为了实现流水灯还是需要一步一步地依次点亮、熄灭 LED：先将 0XFE（11111110）赋给 P0 点亮 LED0，再将 0XFD（11111101）赋给 P0

点亮 LED1 以此类推。为了简化这个过程，我们这里用到两个函数——_crol_()和_cror_()函数，它们在头文件 intrins.h 中。如果你想调用者两个函数，就要先在程序开始时加入这个头文件。_crol_()的功能是按位左循环，_cror_()的是按位右循环。使用范例：

```
P0 = 0XFE;  
P0 = _cror_(P0,1);
```

我们先给 P0 赋 0XFE，即 11111110，然后执行_cror_(P0,1)后，P0 中的值循环右移一位，变成 01111111。可见，右移后，本来最低位的 0 会去到最高位。因此_crol_()和_cror_()函数是循环函数。这要和 $P0 = P0 >> 1$ 区别开来。后者叫右移，如果执行右移，左侧将由 0 来填补。

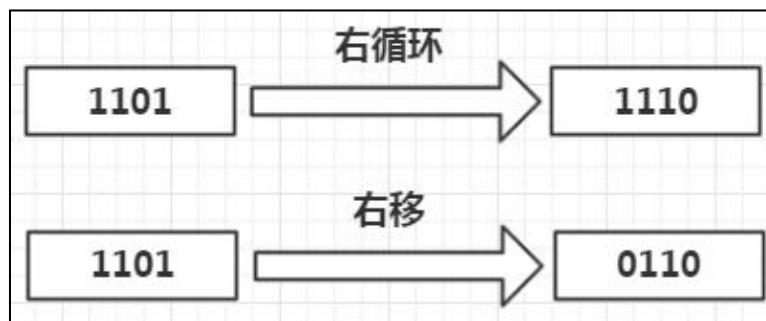


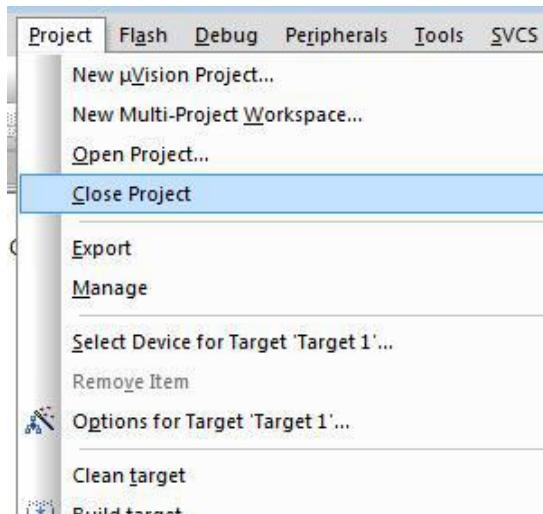
图 2.2.2 右循环和右移

下面，大家接跟着我用并行操作和_crol_()函数，来实现动感十足的流水灯。

2.3 软件实现

关于工程的建立我们已经在第一节课中详细学习到，以后的教程中就不再累赘。所以希望大家要熟悉掌握工程的建立过程。

当我们再次打开 Keil4 时，界面还是会显示上一次的工程。这时我们就选择上方 Project 栏中的 Close Project，关闭工程。



然后我们就可以重新再建立一个工程了。

我们给新的工程起名为“流水灯”，在工程下创建 main.c 文件，写入下列代码：

```
#include "stc12.h"
#include "intrins.h"

#define uint unsigned int//宏定义，用 uint 代替 unsigned int

void Delay_ms(uint z);//声明毫秒延时函数

void main()
{
    P0 = 0XFE;

    while(1)
    {
        P0 = _crol_(P0,1);//循环左移

        Delay_ms(1000);
    }
}

//毫秒延时函数，误差 2us
void Delay_ms(uint z)
{
    uint x,y;

    for(x = z;x > 0;x--)
        for(y = 921;y > 0;y--);
```

可以看到，与第一节课的程序相比，现在的程序并不是一两句就能完事的。随着代码量的增多，良好的编程风格的优势就越明显！好的编程风格至少离不开几点：

①科学地命名变量和函数。变量名和函数名要尽可能的表达变量和函数的用处，而且要用英语，不得使用汉语拼音替代。函数命名我采用的是类似帕斯卡命名法。其实变量和函数的命名方法有很多种方法，也没说那种就是最好的。你可以选择一种你觉得顺手的，并且将其变成习惯，不要三天变一次，否则会增加阅读的难度；

②适当运用的空格、制表符（Tab 键）和空行。空格用于一行代码中运算符号的间隔使语句更清晰，制表符用于 while、for 等语句使程序富有层次感，空行用于把不同用处的代码段分开使程序变得段落分明；

③适量且适当的注释。C 语言的逻辑千变万化，简洁易懂的注释，能让你在以后翻查程序时能快速读懂你过去写的代码。而且这能大大方便他人阅读你的程序（不要说别人写的代码，就算是你写的，如果不加注释，过上一两个月你也看不懂）。当然，累赘且啰嗦的注释只能起到反作用了；

④清晰明了的逻辑。C 语言的奇妙之处之一，就在于达到同一个目标，你可以选择多种方式来实现。条条大路通罗马，但可以走捷径，又何必绕远路呢？所以在确立编程目标后，一定要多思考如何用最简洁明了的逻辑去实现它；

⑤最后，最重要也是最难的，就是要把底层封装好。当你需要时某个功能时，就可以直接调用的是已经封装好的函数。这样一来，你要做一个大项目时就不用又慢慢从底层写起，直接调用之前的文件和函数就行。语言上的结构化，这也是 C 语言的一个强大优势所在。可能现在大家还不能很好理解这点，我们将在应用篇中给大家逐步展现。

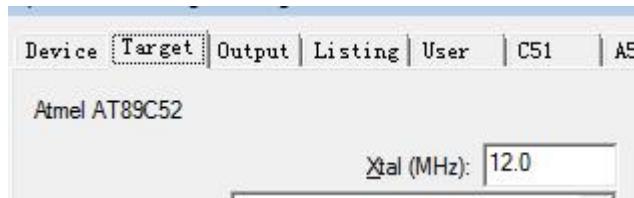
回归程序本身，首先我们看到，这次我们没有对 P0 进行声明。因为我们刚刚说到，在 “stc12.h” 中已经对 P0 进行声明，因此我们无需再次声明 P0。我们可以直接操作 P0，把一个 8 位的十六进制数 0XFE（**再次提醒：十六进制前面要加 0X**），也就是二进制 11111110 赋给 P0，单独点亮 LED0。

然后，我们调用_crol_()函数。可以看到，_crol_()函数接受两个变量，第一个变量是要左循环的值，第二个是左循环一次循环的位数。这里我们循环 P0 的值，每次左循环一位，然后再次赋给 P0，这样就能简单快速点亮一盏 LED 并同时熄灭上一盏 LED。

单片机代码的执行速度是非常快的。对于 stc12，最快的一条指令用时 90ns。如果连续地点亮一盏 LED 然后熄灭上一盏 LED，会超过人眼的分辨极限，这样 8 盏 LED 看起来都是全亮了！为了避免这种情况，我们声明、编写了一个延时毫秒函数 Delay_ms()，放在每一次循环之前，让亮和灭之间稍有间隔。Delay_ms()是两个嵌套

的 for 循环，说白了就是让单片机不断执行一些没意义的循环来达到延时的目的。但我们怎么知道执行 Delay_ms(1000) 的确是用了 1 秒呢？接下来我们就入门一下 Keil 的一个非常强大的功能——调试 (Debug)。

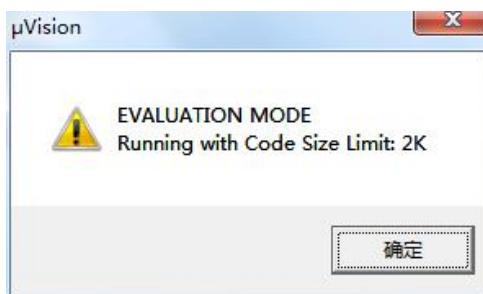
首先我们打开目标选项，在 Target 栏中把晶振 (Xtal) 的频率设置为与我们实验板上的晶振频率 12MHz。（关于晶振，我们会在进阶篇第二节了解到）



然后我们找到 Keil 上方的调试按钮。



可能会有一个警报，可以按确定无视之。



点击进入调试界面。

```

Registers
Register Value
Regs
r0 0x00
r1 0x00
r2 0x00
r3 0x00
r4 0x00
r5 0x00
r6 0x00
r7 0x00
Sys
a 0x00
b 0x00
sp 0x07
sp_max 0x07
PC $ C:0x0021
auxr1 0x00
dptr 0x0000
states 903
sec 0.00007525
psw 0x00

main.c
17 void main()
18 {
19     PO = 0XFE;
20
21     while(1)
22     {
23         PO = _crol_(PO,1); //循环左移
24
25         Delay_ms(1000);
26     }
27 }
28
29 //毫秒延时函数，误差2us
30 void Delay_ms(uint z)
31 {
32     uint x,y;
33
34     for(x = z;x > 0;x--)
35         for(y = 921;y > 0;y--);
36 }
37

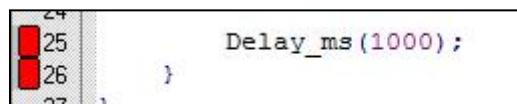
```

Keil 的调试之所以实用，是因为有时候你把程序烧录进单片机后往往会出现出乎意料的情况。而这时单单检查硬件往往是很难解决问题的。因此我们可以从软件调试出发，往往可以发现程序一些隐藏的错误。现在我们通过调试，看看程序执行的时间，而其他功能我们以后会逐一讲解到。

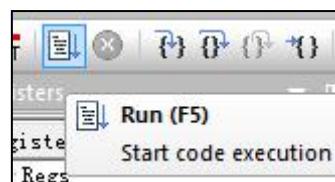
留意调试界面左侧的框中有一栏 sec。它就是 second 的缩写，它反映了程序执行所用的时间。

states	0
sec	0.00000000
psw	0x00

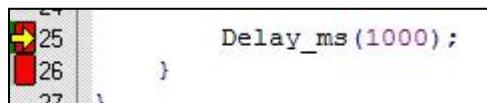
然后我们设置断点。左双击程序中行数标号左侧的深灰色区域，设置断点。设置如下图，红色的两处就是断点。



然后我们就运行程序。左击工具栏中的运行按钮。



可以看到断点处，我们刚刚设置的第一个断点处多了一个黄色箭头。说明程序已经执行到断点处。**这里特别注意！所谓程序执行到断点处，是指程序准备执行断点所在行的代码！也就是说还没执行断点所在行的代码！**



观察程序执行到这里的时间。可以看到，程序从 main 函数第一句开始执行，执行到断点处仅仅用了 0.00007758 秒，可见程序的执行时非常快的。

states	931
sec	0.00007758
usec	0.00

再左击一次运行按钮，发现黄色箭头已经跑到第二个断点处了。仔细观察，你就发现，从第一个断点执行到第二个断点就是执行完 delay_ms(1000)函数。此时我们在看看程序所用的时间。

states	12005515
sec	1.00033108
usec	0.00

两次时间的差值，就是执行完 Delay_ms(1000)所用时间，1000.2535 毫秒，大概就等于 1 秒。所以我们的延时毫秒函数在精度要求不太高的情况下是可行的！但如果对精度的要求比较高（如电子时钟的制作）时，通过嵌套 for 函数得到的延时函数显然就力不从心了，那时我们就必须用到单片机内高精度的定时器。关于定时器，我们将在进阶篇第二节详细了解到。

学过 C 语言的同学都深有体会，当 C 程序跑完 main 函数时，会自动跳出。但是 C51 并不会！当单片机 C 程序跑完 main 程序时，会自动重新执行。所以他跑完一次后会自动循环继续跑第二次。因此习惯地，进入主函数后我们先进行一些初始化，也就是一些配置。然后把要执行的函数放在一个 while(1) 中循环执行，这样就防止了一些初始化被重复执行。

另外大家有没有留意到我们将 unsigned int 宏定义为 uint，后面变量声明时直接用 uint。其实这是有原因的——单片机在绝大多数情况下是不接受负值的。如果这时某个变量因为某种因素变成负值，将导致单片机无法执行程序。所以防止这种情况，除非特殊需要，在单片机编程中尽量使用无符号型变量。而宏定义是为了简化编程。这些都体现了单片机编程的严谨性。

2.4 下载验证

把工程编译生成的 hex 文件烧录进单片机中，可以看到 8 盏 LED 每隔一秒一依次点亮，动感十足的流水灯就这样完成了！

2.5 思考总结

在 Debug 前，我们一定要在目标选项中设置好实验板对应的晶振频率。关于晶振频率，简单来说，它决定了程序执行的快慢。如果实验板上的是 12M 晶振，而你在 Keil 中错填成 24M。那软件调试时程序的执行速度会是实际的两倍。换言之，你调试得到的延时函数在实际应用中会快上一倍。如果不注意这个细节，可能你找上一天都找不到你到底错在哪里。

三、电压比较器——模数转换入门

日常生活中，我们每时每刻都生活在信号的洪流当中：我们看东西有光信号、我们说话有声音信号、我们用通讯用到电磁信号……在电子技术中，信号可以分成两类：模拟信号和数字信号。模拟信号在时间和幅值上是连续的，例如电压、距离等等；相对的，数字信号在时间和幅值上是离散的，例如电脑处理的数据只由“1”和“0”组成。

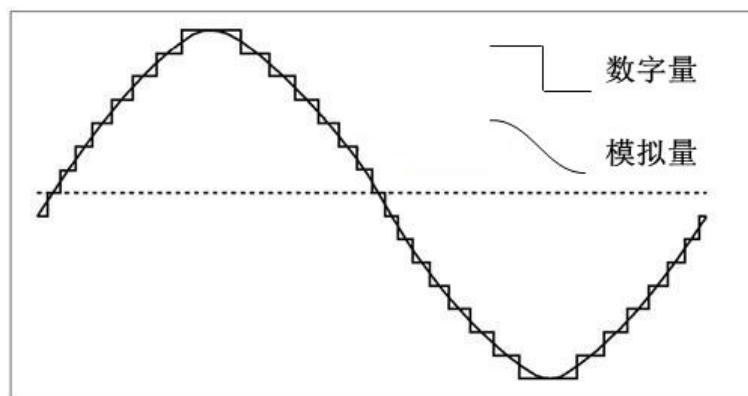


图 2.3.1 模拟量、数字量示意图

我们知道，单片机只能处理数字信号，但实际问题中，我们常常需要把一个模拟信号，如压力、距离等给单片机处理，那该怎么办呢？为了解决单片机处理模拟信号的问题，我们就要用到模数转换器（ADC，Analog to Digital Convert）。在这一节中，我们通过电压比较器这简单的一位模数转换来实现伪触摸开关控制蜂鸣器。

3.1 硬件准备

打开原理图，找到电压比较器模块：

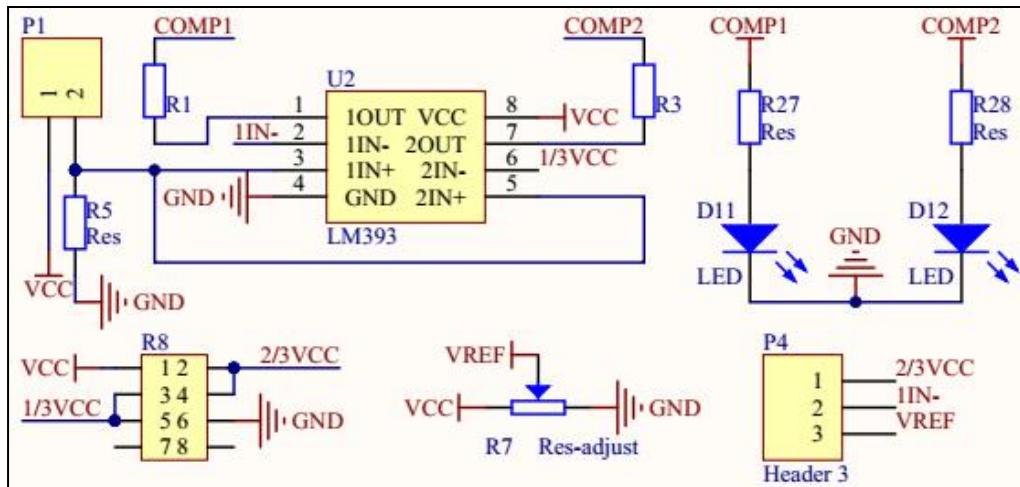


图 2.3.2 电压比较器模块电路连接

电压比较器模块输出脚 COMP1 和 COMP2 与单片机连接的。COMP1 与单片机 P1.6 连接，而 COMP2 与 P1.7 连接。

Pin	Function	Pin	Function
COMP1	P1.6	2	MOSI/ADC5/P1.5
COMP2	P1.7	3	MISO/ADC6/P1.6
	DA 7	4	SCLK/ADC7/P1.7

图 2.3.3 电压比较器模块与单片机连接

另外 COMP1 和 COMP2 分别串联一个电阻后，连接一盏 LED 后再接地。当 COMPx 脚输出高时，就能点亮 LED。

3.2 知识点讲解

我们选用的是双路电压比较器芯片 LM393。它里面有两个一位的电压比较器。1IN- 和 2IN- 分别是两个电压比较器的参考电压输入端；1IN+ 和 2IN+ 分别是两个电压比较器输入端；1OUT 和 2OUT 分别为两个电压比较器信号输出端。以其中 1 个电压比较器为例，若 1IN+ 比 1IN- 大，则 1OUT 输出 1；反之，1OUT 输出 0。

我们可以从下图中更清楚地看到 LM393M 的内部结构：

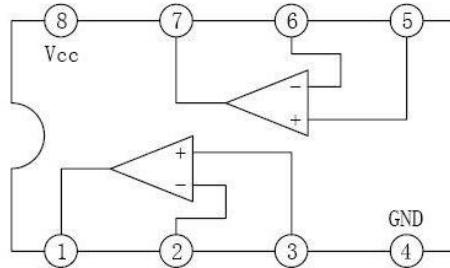


图 2.3.4 LM393M 内部结构

一个电压比较器有两个输入端，为了方便比较，通常我们将其中一个输入端的电压设为定值，称为参考端，输入该端口的电压称为参考电压；而另外一端是用户根据自己需要输入电压信号与参考电压作对比，称为信号输入端，输入该端口的电压称为输入电压。

至于电压比较器的输出，我们不管正、负输入端哪个充当信号端，哪个充当惨开端。只要当正输入端电压大于负输入端电压时，电压比较器输出高电平；反过来当负输入端电压大于正输入端电压时，电压比较器输出低电平。

我们的电路中，1IN+和2IN+并联后接一个插座然后再接VCC。插座可以插各种可变电阻，包括光敏、温敏之类的电阻。以光敏电阻为例，插上插座后，改变环境中光的强弱，光敏电阻阻值会随之变化。光越强，光敏电阻阻值越小，分压就越小，输入到比较器的电压就越大，反之亦然。这样我们就把光信号变成电压信号。我们把光敏电阻分压后的电压信号同时给LM393内两个电压比较器的其中一端：1IN+和2IN+，这两个端口就同时作为两个电压比较器的信号输入端。

而另外两个输入端，其中一个2IN-通过电阻分压后输入固定的 $1/3VCC$ ；另一个1IN-可以通过改变拨码开关的位置来选择输入 $2/3VCC$ 或者调节变阻器从 $0\sim VCC$ 之间一个值作为输入。它们都是作为定值与信号输入端所输入的电压信号做比较，因此它们都是比较端。



图 2.3.5 实验板上的拨码开关

如今，我们将电压信号同时输入信号输入端 1IN+和 2IN+，这个电压信号分别与两个比较端 1IN-和 2IN-的电压比较，分别从 1OUT 和 2OUT 输出比较结果。这样一来，我们就可以根据 1OUT 和 2OUT 的输出情况来判断输入电压与两个参考电压的大小关系了。为了方便观察，我们还从 1OUT 和 2OUT 各接上一盏 LED，这样我们就可以通过观察 LED 的亮灭情况快速判断两个电压比较器的输出情况。

在前两节中，我们都是将 I/O 口作为输出来控制 LED。而如今，我们要通过 I/O 口来读取 1OUT 和 2OUT 的输出情况，我们这就要将 I/O 口设置为输入模式。

51 单片机将 I/O 口设置成输入模式的方法很简单，只需将该 I/O 口置 1，它的电平高低就能随外部变化而变化。51 单片机 I/O 口复位后是默认为高电平的，所以我们可以跳过置 1 直接默认 I/O 口为输入模式。**注意！如果我们通过编程将某个 I/O 口置 0 而又外接高电平，将极有可能将单片机烧坏！所以这种行为万万不可取。**

3.3 软件实现

下面我们以光敏电阻，1IN-输入 2/3VCC 为例，来完成一次简单的 AD 转换。

我们给新的工程起名为“电压比较器”，在工程下创建 main.c 文件，写入下列代码：

```
#include "stc12.h"

//声明蜂鸣器控制引脚及电压比较器采集引脚
sbit beep = P1^5;
sbit cpa1 = P1^6;
sbit cpa2 = P1^7;

void Beep(int n);
void Delay_ms(int m);

void main()
{
    while(cpa1 == 0 && cpa2 == 0)//蜂鸣器不响
    {
        beep = 1;
    }
    while(cpa1 == 0 && cpa2 == 1)//蜂鸣器 1 秒响一次
    {
        Beep(1000);
    }
    while(cpa1 == 1 && cpa2 == 1)//蜂鸣器 0.1 秒响一次
    {
```

```
        Beep(100);
    }
}

//蜂鸣器警报函数
void Beep(int n)
{
    beep = 1;
    Delay_ms(n);
    beep = 0;
    Delay_ms(n);
}

//毫秒延时函数，误差 2us
void Delay_ms(int m)
{
    int x,y;

    for(x = m;x > 0;x--)
        for(y = 921;y > 0;y--);
}
```

3.4 下载验证

将工程编译生成的 hex 文件烧录进单片机。通过改变环境光强我们发现，当光比较弱时，蜂鸣器不响；当稍微有点光时，蜂鸣器以一秒的间隔报警；当环境中光线较强时，蜂鸣器将急促报警。

我们把手指慢慢靠近光敏电阻，借此改变光敏电阻接受到的光的强度。随着我们手指的靠近，蜂鸣器发出的声音也随之变化。这就类似一个触摸开关了。但这并不是真正的触摸开关，所以我们称它为伪触摸开关。

当然，如果我们把光敏电阻换成一个温敏电阻，也就能做一个简单的温度报警系统了！

3.5 思考总结

在本节实验中，我们只用到两个电压比较器，如果我们通过跳线帽将 1IN-设置成输入 2/3VCC，那我们想通过单片机测量一个未知的电压信号时，可能只知道这个电压信号介于 1/3VCC 和 2/3VCC，根本无法做到精确测量。

这就关乎到 AD 的精度问题了：AD 精度越高，你测量出来的数字量就越接近实际的模拟量。毕竟模拟量是连续的，它的精度是无穷无尽的，我们能做的，只有选用精度符合要求的 AD 转换器去把这个模拟量转换为相应精度的数字量。本实验板用的增强型 51 单片机 STC12 内置 1 个 8 路 10 位高速 A/D 转换器，它的精度达到 $1/2^{10}$ 。关于 STC12 的内部 ADC，我们将在应用篇咪头一节中娓娓道来。

四、PWM 入门——控制技术中的万金油

在前面三节中，我们试着用各种各样的方式操作着高低电平。但别误会，我们玩单片机绝！对！不！是！为了如何控制高低电平！单片机的控制方式并不局限于肤浅地把 I/O 口置高或置低。

在这节课中，我们将从将 I/O 口置高和置低为基础，一起了解一种广泛应用于从数模转换、测量、通信到功率控制与变换等许多领域中的控制技术——PWM（Pulse Width Modulation），中文名叫脉冲宽度调制。

4.1 知识点讲解

PWM 作为一种应用广泛的控制技术，在了解 PWM 之前，我们先要引入几个概念：

什么是方波脉冲？所谓脉冲就是短暂起伏的电压（或电流，而在单片机控制中一般指电压），而方波脉冲顾名思义，就是方波形成的脉冲。脉冲分为高脉冲和低脉冲，它们的样子如下图：

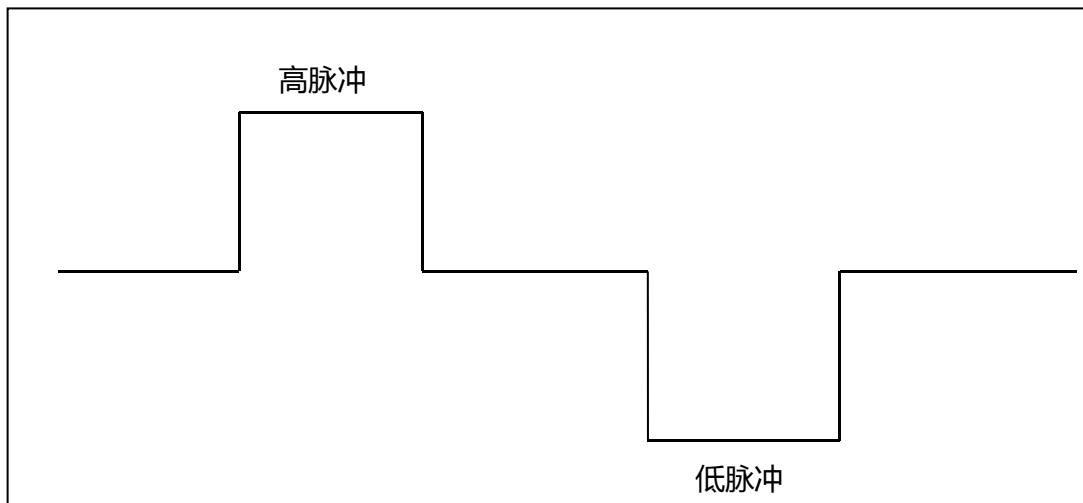


图 2.4.1 脉冲示意图

我们知道，单片机控制信号由高电平和低电平构成，所以基本不会出现上面这种脉冲信号。一般的，在高电平中忽然出现一段低电平，形成低脉冲；反之在低电平中忽然出现一段高电平，形成高脉冲。在 PWM 中我们用到的脉冲就是这种方波脉冲。

大家对脉冲有个基本的认识后，我们就来了解一下 PWM 中方波脉冲的一些基本参数。

①脉冲周期：PWM 所用的方波脉冲周期性重复的脉冲序列。脉冲周期就是两个相邻的脉冲之间的时间。

②脉宽：脉冲有“胖”有“瘦”，为了衡量一个脉冲的“样子”，我们就要引入脉宽的概念。脉宽是指高脉冲或低脉冲的时间。

③占空比：占空比就是周期内高脉冲脉宽占脉冲周期的百分比。因为脉冲的周期是改变的，所以占空比比脉宽更能直观体现脉冲的特性。

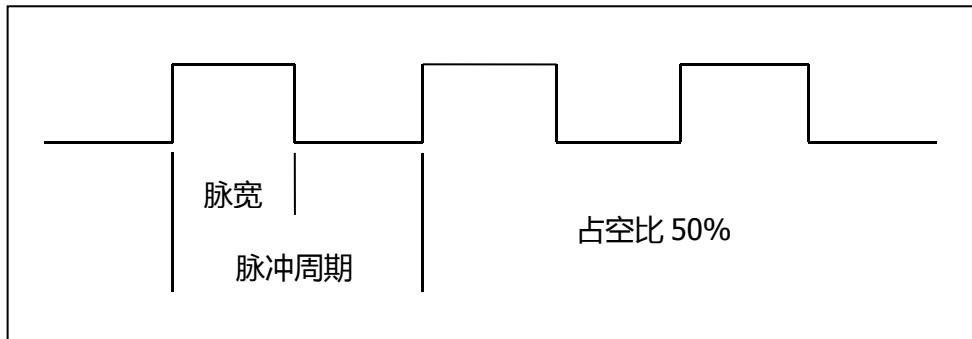


图 2.4.2 PWM 各种参数示意图

PWM 中文名叫脉冲宽度调制，因此 PWM 是通过调节脉宽的长短来实现的。但正如上文所说，因为脉冲周期是不定的，脉宽并不能直观地体现脉冲的特性。但占空比是一个百分比，能更直观反映脉冲的特性。所以我们讨论 PWM 时，都是讨论占空比而不是脉宽。

4.2 软件实现

我们新建一个名为“PWM”的工程，在工程下创建 main.c 文件，写入下列代码：

```
#include "stc12.h"

#define uint unsigned int

void Delay_ms(uint z);
void PWM(uint x);

sbit led = P0^0;

void main()
```

```

{
    PWM(1); //PWM 频率为 50Hz , 占空比为 95%
}

//PWM 产生函数
void PWM(uint x)
{
    led = 1;
    Delay_ms(20 - x);
    led = 0;
    Delay_ms(x);
}

//毫秒延时函数 , 误差 2us
void Delay_ms(uint z)
{
    uint x,y;

    for(x = z;x > 0;x--)
        for(y = 921;y > 0;y--);
}

```

在这几节的程序中，我们都多次用到 Delay_ms() 函数。其实，对于一些经常用到的函数，我们并不需要每次都重新码一遍。我们只需充分发挥 C 语言移植性好的特点，复制以前的代码粘贴到本工程中即可继续使用。

在主函数中，程序不断循环执行 PWM() 函数，我们就来着重看看 PWM() 函数。

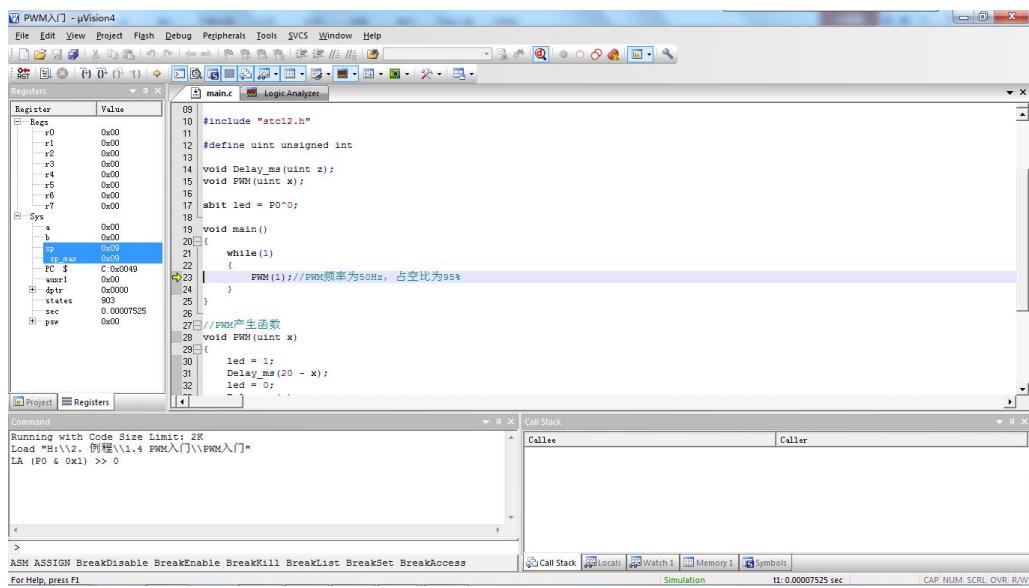
每次执行 PWM(1) 函数时，都是先把 P0.0 即 led 置高延时 19 毫秒，然后把 P0.0 拉低延时 1 毫秒。于是我们就得出脉冲周期为 20ms 即频率为 50Hz，占空比 95% 的 PWM 信号。

在一个脉冲周期内，当 P0.0 为高电平时即高脉冲期间，led 处于熄灭状态；当 P0.0 为低电平时，led 处于点亮状态。所以现在 20ms 内，有 19ms 的时间 led 是熄灭的，另外 1ms 的时间 led 是点亮的。因此此时 led 比较暗。可见，只要我们调整 PWM 脉冲的占空比，就能控制 led 灯的亮度了！

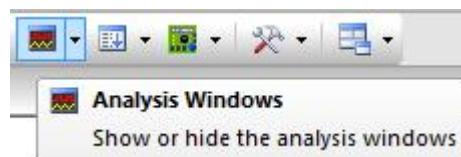
4.3 仿真验证

在前面的学习中，我们就已经知道 Keil 能在软件层面上进行硬件层面的仿真。为了让大家感受仿真调试的实用性和重要性，这里我们就通过仿真的方法看看 PWM 信号是长什么样子的。

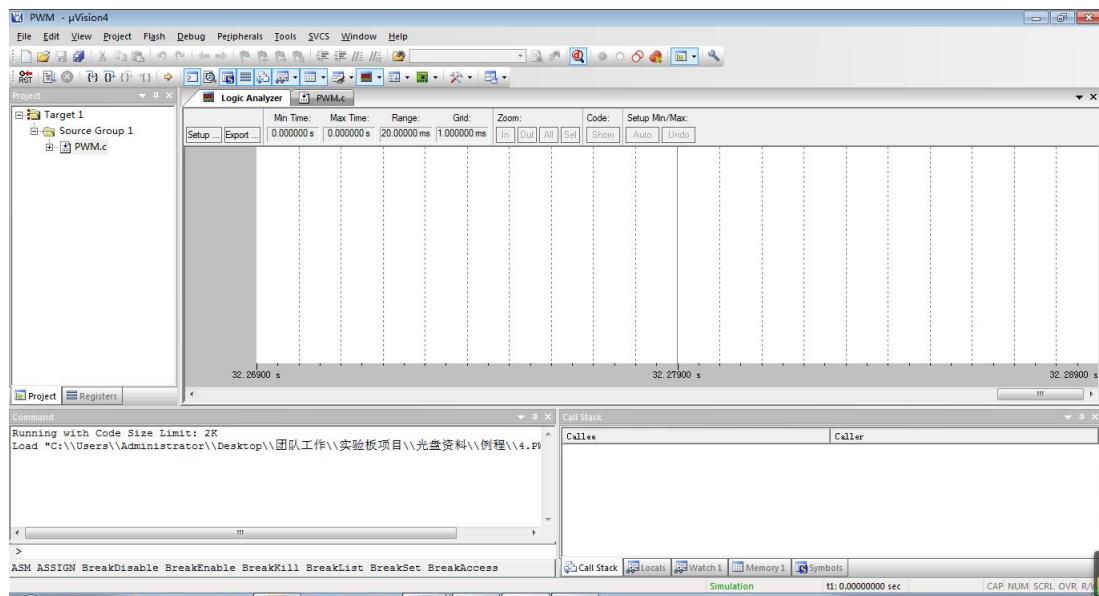
首先我们打开仿真界面。



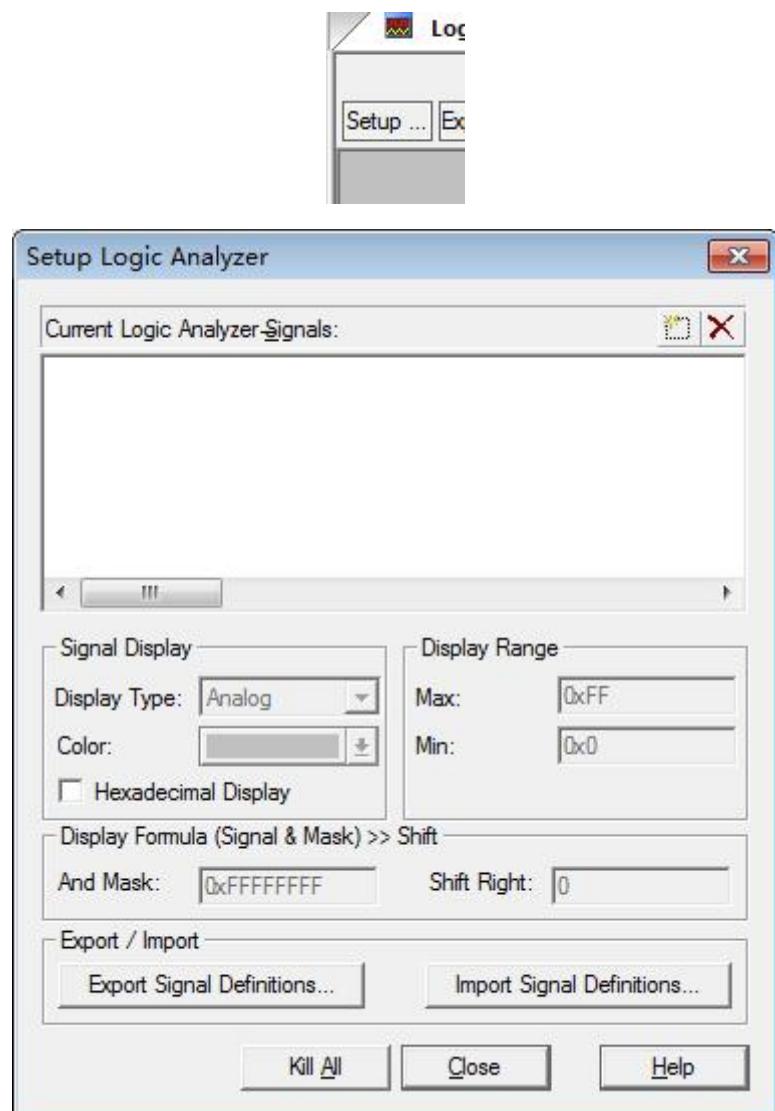
打开分析框。



如下图。



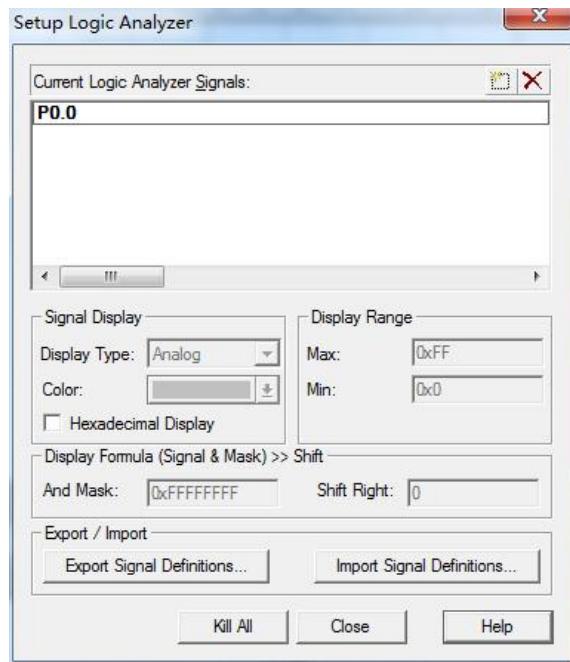
然后我们设置要观察的 P0.0。点击 Setup。



左击新建按钮。



输入 P0.0 , 回车确定。

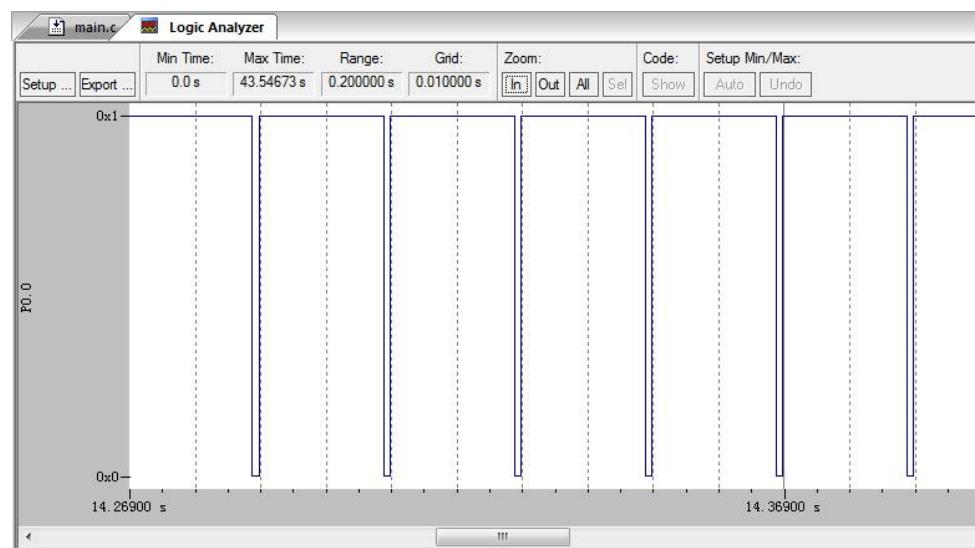


关闭对话框。点击运行程序 RUN , 运行程序。

我们发现分析框中显示的速度太快了，不方便观察。于是我们可以左击在速度 Zoom 栏中的 Out 把速度降下来。



调节到合适速度的后，我们就能看到我们模拟出来的 PWM 信号了！

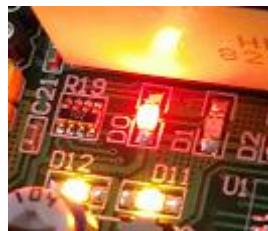


4.4 下载验证

我们把工程编译生成的 hex 文件烧录进实验板，观察 LED 的亮度。



然后我们回到程序中，将 PWM()的型参改成 19。再次烧录进单片机，观察 LED 的亮度变化。



可见，当 PWM()型参为 19 时，LED 的亮度明显比型参为 1 时亮。

4.5 思考总结

在这节中，我们用改变占空比的办法来调节 LED 亮度。占空比是数字量，而 LED 的亮度却是模拟量。有没有发现，其实这是和第三节的模数转换相对的数模转换！但这只是 PWM 的一次牛刀小试。

本节中，我们用 for 延时函数模拟出的 PWM 信号是非常简陋的。毕竟我们前面说过，for 延时函数在精度要求不高的时候可以使用，怎么能满足我们精益求精的精神呢？

其实，一般的 51 单片机都可以用定时器（关于定时器，我们会在进阶篇第二节详细学习到）产生高精度的 PWM 信号。而我们实验板采用的增强版 51 单片机 STC12 更是内置了可以方便制造出 PWM 脉冲信号的 PCA 模块。我们会在进阶篇第三节了解到 PCA 的功能与使用。

进阶篇

经过入门篇的了解，不知道大家有没有稍微体会到单片机的乐趣呢？接下来，我们就由浅入深，重点看看一下单片机最常用的四个功能——按键、定时器、PWM 和串口。在单片机技术中，很多外设的控制都离不开这四个功能。所以可以说，只有你能熟练地掌握这四个功能，你才算真正入门了单片机。

一、按键——程序由我掌控

在这之前的 LED 控制中，我们都是通过软件的方法，先编程，再烧录，然后单片机自己执行。因此这种情况下，单片机程序的执行过程中，你是不能控制的程序的执行过程的。难道你想改变程序的执行过程，你就要先修改程序再重新烧录？为了解决这个问题，我们可以通过各种外设来控制单片机程序的执行过程，其中按键就是最典型的例子。

1.1 硬件准备

打开光盘资料中的实验板原理图，找到按键模块：

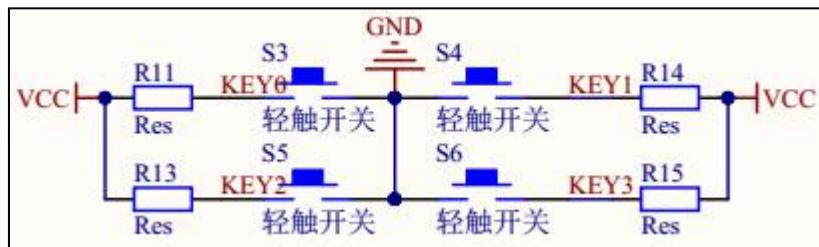


图 3.1.1 实验板按键电路连接

我们实验板配置了四个按键:KEY0、KEY1、KEY2、KEY3。四个按键一端分别接 P3.0、P3.2、P3.1、P3.3。而这四个按键引脚又分别串联一个上拉电阻后接 VCC。按键的另一端统一接地。

我们再看看按键正背面实物图：

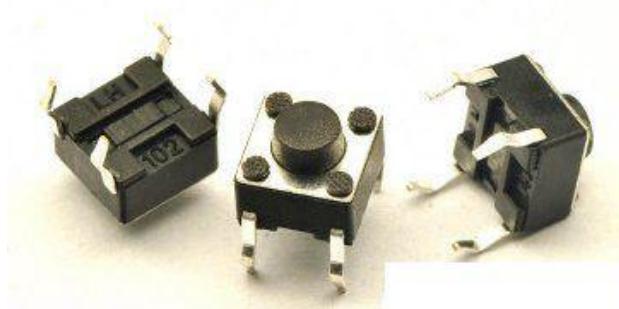


图 3.1.2 轻触按键正背面实物图

实验板采用的按键是上图这种轻触按键，它带有四个引脚，按下松手后会自动复位。在轻触按键底面我们可以观察到四个引脚之间有一道凹槽。在按下按键之前，凹槽两侧的两个引脚是导通的。而当按键按下时，凹槽两侧的引脚会被铁片导通。这样四个引脚就变成相互导通的。

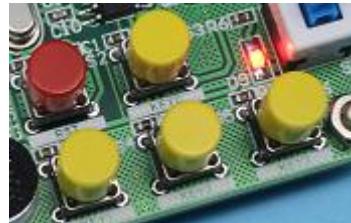


图 3.1.3 实验板上的按键

为了方便识别，不用每一次都看一下下方凹槽方向，一般我们使用轻触开关时，只使用对角的两个引脚。这样就能保证按下前两端时断路的。

1.2 知识点讲解

其实按键检测的实现非常简单。我们在将电压比较器时已经提到，单片机的 I/O 口复位后是高电平的，即默认状态下为输入模式，引脚的电平高低随外界变化而变化。

通过原理图我们知道，当按键没按下前，单片机的 I/O 口先串联一个上拉电阻后接到 VCC 上，因此此时对应的 I/O 口为高电平。

所谓上拉，是将一个不确定的电平信号通过一个上拉电阻钳位在高电平，同时电阻也起到的限流的作用。同理，下拉就是将一个不确定的电平信号通过一个下拉电阻钳位在低电平。区分上拉和下拉电阻的方法很简单：接 VCC 的是上拉电阻；接 GND 的是下拉电阻。

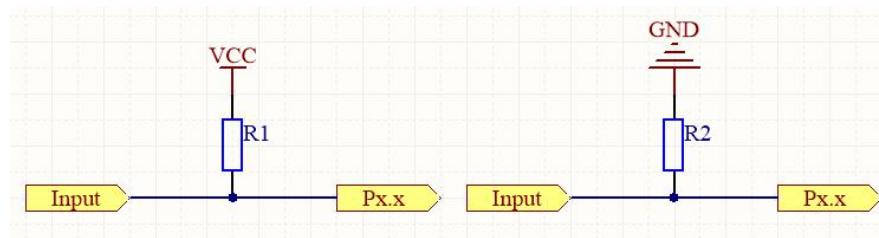


图 3.1.4 上拉电路（左）&下拉电路（右）

看上图两个电路。左图是上拉电路，通过接 VCC 的 R1，外部输入的 Input 信号不管是高还是低，都会被拉高，所以 Px.x 一直检测到高电平；反之，右图 Input 信号一直被拉低，Px.x 一直检测到低电平。

回到电路图。我们让 I/O 口接上拉电阻后接 VCC 是保证按键被按下前，I/O 一直为电平。

若按键按下，该按键所连 I/O 口将接地，会变成低电平（在电平中，这叫下降沿，相反由低电平变为高电平为上升沿）。因此，只要我们通过编程检测当 I/O 口的电平情况，就能判断按键是否被按下了。另外，按下按键后，上拉电阻也起到了避免 VCC 和 GND 短接，保护电路的作用。

但仅仅检测一个按键是不够的，毕竟在实际应用中，我们往往不会只用到一个按键。因此我们就必须通过编程来分辨到底是哪个按键被按下了。根据实际需要的不同，检测按键的方法有两种：扫描法和中断法。为了让大家对按键检测有一个全面的认识，我们就分别建立两个工程，看看扫描法和中断法是怎样实现按键检测的。

1.3 软件实现

我们先来看看扫描法检测按键：

我们新建一个名为“按键-扫描法”的工程，在工程下创建 main.c 文件，写入下列代码：

```
#include "stc12.h"
#include "intrins.h"

#define uint unsigned int
#define LED P0

void Delay_ms(uint z);

sbit key0 = P3^0;
sbit key2 = P3^1;

void main()
{
    LED = 0XFE;//点亮 LED0

    while(1)
    {
        if(key0 == 0)//检测按键 key0
        {
```

```

Delay_ms(10);
if(key0 == 0)//消抖
{
    LED = _cror_(LED,1);//LED 灯右移一位

    while(!key0);//松手检测
}
}

if(key2 == 0)//检测按键 key2
{
    Delay_ms(10);
    if(key2 == 0)//消抖
    {
        LED = _crol_(LED,1);//LED 灯左移一位

        while(!key2);//松手检测
    }
}
}

void Delay_ms(uint z)
{
    uint x,y;

    for(x = z;x > 0;x--)
        for(y = 921;y > 0;y--);
}

```

程序的逻辑非常简单，点亮一盏 LED 后，在 while 循环里不断检测 key0 和 key2 是否按下。当按键 key0 按下时，LED 左移一位；当按键 key1 按下时，LED 右移一位。细心的你有没有发现，当我们每次用 if 语句检测到按键被按下时，都会先延时 10ms，然后再次用 if 语句检查按键是否被按下，这是为什么呢？

其实，在理想情况下，当按键按下时，连接的 I/O 口的电平会由高变低。但实际情况中，当按键按下时，连接的 I/O 口的电平由高变低后，会有短时间的抖动，如图：

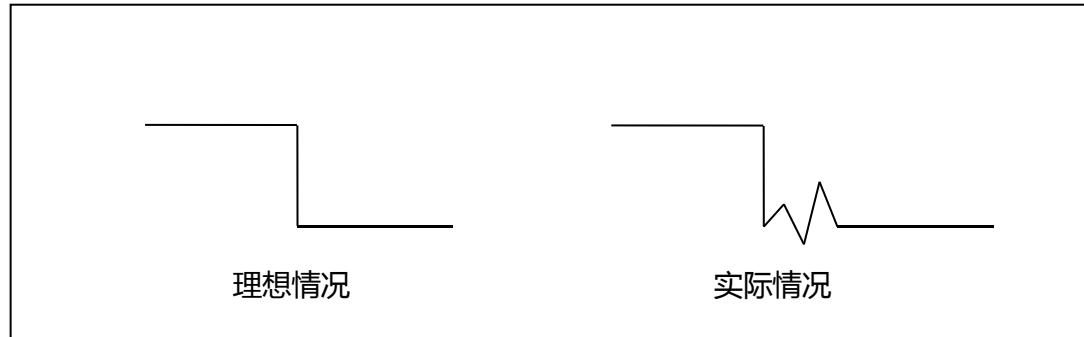


图 3.1.3 抖动示意图

我们在准备篇第三节的 TTL 电平中说过，高电平和低电平有个范围值：对于 51 单片机的输入信号，大于+2V 为高电平，低于+0.8V 为低电平。所以如果在抖动中，刚好有个抖动的电压值达到+2V，在判断瞬间单片机依然认为是高电平，那按键判断岂不是没意义了吗？幸好，电平的抖动持续的时间不会超过 10ms。因此我们在第一次判断后隔 10ms 秒再判断一次，就相当于双重保险，消除抖动带来的影响，简称消抖。

在执行完 LED 的左右移后，我们在后面加入一个松手检测。因为按下时，按键连接的 I/O 口为低电平。如果按键一直被按着，就通过 while 语句让程序一直处于死循环。直到按键被松开，按键连接的 I/O 口恢复高电平才能跳出 while 循环，达到按一下按键，LED 动一次的效果。

以上就是扫描法按键检测的全部内容。接下来，我们看看中断法是如何实现按键检测的：

我们新建一个名为“按键-中断法”的工程，在工程下创建 main.c 文件，写入下列代码：

```
#include "stc12.h"
#include "intrins.h"

#define uint unsigned int
#define LED P0

void Con_Init(void);

void main()
{
    P0 = 0XFE;
    Con_Init();

    while(1);
}

//外部中断初始化函数
void Con_Init(void)
{
    //设置外部中断 0
    IT0 = 1;//设置外部中断 0 下降沿捕获
    EX0 = 1;//打开外部中断 0 中断允许

    //设置外部中断 1
```

```

IT1 = 1;// 设置外部中断 1 下降沿捕获
EX1 = 1;// 打开外部中断 1 中断允许

EA = 1;//打开总中断
}

//外部中断 0 服务函数
void Key0_Handler(void) interrupt 0
{
    LED = _cror_(LED,1);
}

//外部中断 1 服务函数
void Key1_Handler(void) interrupt 2
{
    LED = _crol_(LED,1);
}

```

相对于扫描法，中断法按键检测的程序比较长，知识点也比较多。但大家不要因为代码一多就心存畏惧，其实只需理清思路，这些代码对于大家来说肯定是小菜一碟！接下来我们就逐一分析中断法按键检测。

在 C 语言中，中断的概念就是程序执行到某个时刻时，不执行接下来的代码而跑去执行指定的代码。待指定的代码执行完后，返回原来程序停止执行的地方，继续执行原来的程序。为了方便大家理解，我们举一个生活上的例子：如果你正在写作业，忽然间有人打电话过来，那你就要中断手中的作业去听电话，等听完电话后回来继续写作业。再举个很简单的例子，我们一直用到的 Delsy_ms() 函数就是一个很好的中断例子。当我们执行 Delay_ms() 函数时，先从主函数中跳入 Delay_ms() 函数执行其中的函数，等 Delay_ms() 函数执行完后再回去主函数中继续执行 Delay_ms() 函数后面的函数。

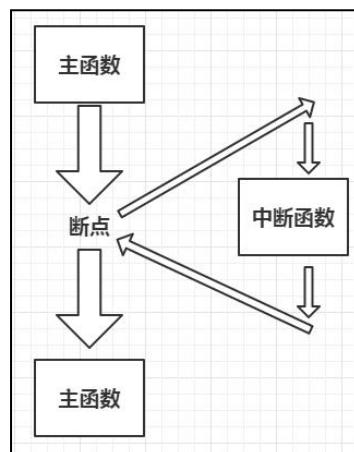


图 3.1.5 中断示意图

这种方法只是软件上的中断，而在单片机的实际运用中，往往需要通过一些外部信号来控制单片机内程序的执行（换句话说，就是通过外部信号产生各种中断），例如我们这节讲的按键。这种由外部硬件产生的中断，称为**外部中断**。当引脚检测到相应的中断源时，程序就会跳去执行中断服务函数。

我们打开光盘中的实验板原理图，找到按键 KEY1 和 KEY3 连接单片机的两个 I/O 口：P3.2 和 P3.3。咦？它们名字旁边居然还有别称——INT0 和 INT1：

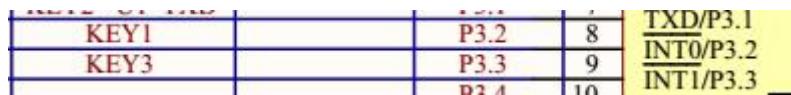


图 3.1.5 按键与外部中断连接

这说明这单片机的这两个引脚不仅仅用作 I/O 口，还用作外部中断的输入口。为什么有些引脚那么“花心”，要一心二用呢？原来，因为单片机的引脚数是固定的，为了让单片机能有更多的更能，一些引脚可以有多个用途。一些更高级的单片机更是可以通过引脚映射，把一个引脚当另外一个引脚来用！

在这里，我们可以看到 P3.2、P3.3 不仅仅是 I/O 口，更是外部中断的输入口。但是怎样告知单片机，你是把引脚用于 I/O 口还是外部中断的输入口呢？为了解决这个问题，我们需要设置两个特殊功能寄存器：中断允许寄存器 IE 和定时器/计数器控制寄存器 TCON。

和 I/O 口一样，IE 和 TCON 都是 8 位的特殊功能寄存器。关于两个寄存器的位定义，大家可以打开光盘中->4. 芯片资料->1. STC12C5A60S2 中文参考手册。IE 在 P209，TCON 在 P213。与本节相关的位定义如下图：

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	EA	ELVD	EADC	ES	ET1	EX1	ET0	EX0

表 3.1.1 IE 相关位定义

EA：总中断允许控制位。置 1 打开总中断，置 0 关闭总中断；

EXx：外部中断 x 中断允许位。置 1 允许外部中断 x 触发中断，置 0 则不云允许该外部中断 x 触发中断。

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

表 3.1.2 TCON 相关位定义

ITx : 外部中断 x 中断方式控制位。置 1 为下降沿触发，置 0 为低电平触发。

查看参考手册是玩单片机的必学技能，而且是必需加满技能点的技能！授人以鱼不如授人以渔。单片机种类琳琅满目，每种单片机的内部结构和操作方式都不尽相同。你不可能逐一学习每一款单片机的使用。而如果你学会查看参考手册，即使有一款陌生的芯片，你都能快速驾驭它！

因为我们在 “stc12.h” 中定义好了 IE 和 TCON，所以 TCON 和 IE 可以直接位操作而无需声明。这里我们要设置 TCON 的 ITx 位（小写 “x” 为数字，值视具体情况而定，下同。这里 “x” 为 0 或 1，分别对应外部中断 0 和外部中断 1），IE 的 EXx 位和 EA 位。

TCON 的 ITx 位是设置中断触发方式：1 为下降沿触发，只要 INTx 遇到下降沿就触发相应中断，0 为低电平触发，如果 INTx 为低电平则持续触发相应中断。

IE 的 EXx 位是外部中断允许位：0 为关闭中断允许，1 为开启中断允许；EA 位是总中断允许位：0 为关闭全部中断，1 为开启全部中断。外部中断开关和总中断开关双开关的方法，大大增强中断的灵活性——你可以只关闭其中一个外部中断，又可以把全部中断一下子全部关闭。等以后随着外设的增加，大家就会慢慢体会到其中的实用性。

程序中，我们把外部中断的设置都封装成一个初始化函数。另外我们上面说过，单片机程序会在 main() 函数里不断循环，为了防止重复初始化，会在初始化后嵌套一个 while(1) 死循环，把要重复执行的程序放在 while 循环里面循环执行。

外部中断初始化后，还要有相应的服务函数，即中断产生后，单片机要执行的程序。**中断服务函数无需声明，但在函数名后应加上 “interrupt x”**！单片机内部可以产生的中断不仅仅只有外部中断。其它诸如定时器、串口等功能都会产生中断。为了避免编译器傻乎乎搞不清中断服务函数对应着哪一个中断，“Interrupt x” 就相当于一个标识符。它告诉编译器你写的中断服务函数对应着哪个中断。interrupt 后面跟着的数字 x 并不是随便定的，它都是由芯片设计者定好的。外部中断 0 对应着 interrupt 0，外部中断 1 对应着 interrupt 2，其他中断对应的序号 x 如下表：

中断编号	中断方式
------	------

0	外部中断 0
1	定时器 0
2	外部中断 1
3	定时器 1
4	串口 1
5	ADC 功能
6	低电压
7	PCA 模块
8	串口 2
9	SPI 功能

表 3.1.3 中断序号对应中断方式

1.4 下载验证

分别两个把工程编译生成的 hex 文件下载进单片机，分别按 key0 和 key1，观察 LED 的变化。

1.5 思考总结

在本节我们分别讲了扫描法和中断法来检测按键。不知道你有没有思考过，什么时候用扫描法，什么时候用中断法呢？

我们先回顾一下扫描法按键检测。扫描法按键检测中，为了实时检测 I/O 口的情况，我们在 while 循环里面要不断检测 I/O 口的情况。现在我们只用到按键检测，但我们不会总只做按键检测功能而不实现其他功能吧？在实际应用中，按键往往回搭配其他模块使用。如果其他模块又需要在 while 循环内进行逻辑判断，那整体的逻辑就变得非常臃肿了。

而中断法检测就能很好地解决这个问题。因为我们设置好中断相关的寄存器后，单片机就在后台不断检测是否产生中断。所以我们看到中断法中 while 循环里是不用执行

任何代码的。因此在我们按下按键前，while 循环里面可以执行其他代码。只有当我们按下按键时才去判断按键相关的逻辑。但 STC12 的外部中断只有两个。换言之中断法最多只能支持两个按键。但扫描法却对要检测的按键数量没有限制。

总结起来，两者各有优劣。到底要哪种方法就需要我们因地制宜了。

二、定时器——单片机里的小闹钟

在入门篇第二节中我们提到，为了得到高精度的计时，靠 for 延时函数是远远不够的。而且通过 Delay_ms() 函数只能做到延时而不能用于计时或者定时。

假如我们要做一个定时器，让单片机按照我们的意愿每隔一段时间就执行一些操作。那我们要怎么实现呢？这时我们就要借助单片机内部的定时器/计数器来进行到高精度的计时了。

2.1 知识点讲解

我们日常生活中用十进制计数，而单片机却是用二进制处理数据。单片机的时钟制也和我们日常生活中的 60 进制也大有不同。我们日常生活中，60 秒为 1 分钟，60 分钟为 1 小时。秒是最小的计时单位。而在单片机中，最小的计时单位并不是秒，而是时钟周期。那时钟周期是什么，又是怎样产生的呢？

这还得由我们在入门篇第二节提到的晶振说起。晶振全称晶体振荡器（Crystal Oscillator），它是单片机的“心脏”，我们可以在实验板上直接找到这个银色盒的小家伙：

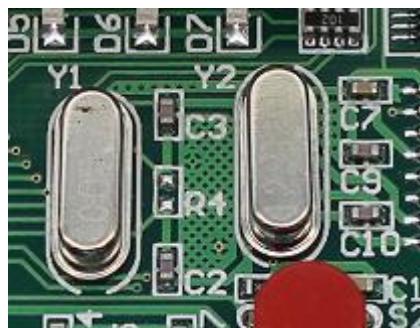


图 3.2.1 实验板上的晶振

之所以称晶振是单片机的心脏，是因为晶振通电后会不断产生机械震荡。震荡经过单片机内硬件处理后产生脉冲，单片机就在这个脉冲的指引下工作。这就像人的心脏一样，每一次搏动都为身体输送血液。其中，晶振每次震荡所用的时间就是时钟周期，也称为系统时钟（SYSclk），震荡频率称为时钟频率。时钟周期是单片机执行一个最基本动作所需要的时间。这类似一个节拍器，晶振每打一个节拍，单片机就执行

一个最基本动作。它是单片机内部所有工作的时间基准，就相当于我们日常生活中时间对我们工作的规范作用一样。基于石英晶体的物理特性，普通晶振的精度能达到百万分之五！这就是定时器能有高精度的物理基础。

但时钟周期只是单片机时间系统中最小的计时单位。好像我们日常生活中的最小的计时单位秒一样，我们不可能都用秒作为我们活动的时间尺度。否则我们今天的行程就变成 28800 秒起床、43200 秒吃午饭……日常生活中，为了方便计时我们将 60 秒定为 1 分钟，60 分钟定为 1 小时。类似的，在单片机中也是如此。对于传统 51 单片机，它将 12 个时钟周期定为 1 个机器周期。机器周期是单片机执行一个最简单指令（单机器周期指令）所用的时间。这种 12 个时钟周期为 1 个机器周期，1 个机器周期执行一条最简单指令的单片机我们称为 12T 单片机。但我们现在使用的 stc12 单片机是 1T 单片机：对于 stc12 来说，1 个时钟周期就是 1 个机器周期，所以相对于传统 51 单片机，stc12 执行指令的速度大大提升：最低 3 倍，最高 24 倍。

我们实验板采用的晶振的频率为 12MHz，时钟周期的频率为 $1/12 \times 10^{-6}$ 秒。stc12 为 1T 单片机，1 个时钟周期为 1 个机器周期，所以 1 个机器周期就是 $1/12 \times 10^{-6}$ 秒。但对于定时器/计数器，stc12 默认情况下依然是 12T 模式，也就是仍相当于（注意这里用词）12 个时钟周期为 1 个机器周期，所以一个机器周期为 $1/12 \times 10^{-6}$ 秒即 1us。当然你也可以通过设置辅助寄存器 AUXR 的第 6 位 T0x12 和第 7 位 T1x12 分别将定时器 0 和定时器 1 设置工作在 1T 模式下。AUXR 寄存器的位定义如下表：

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	T0x12	T1x12	UART_M0x6	BRTR	S2SMOD	BRTx 12	EXTRAM	S1BRS

表 3.2.1 AUXR 相关位定义

T0x12：定时器 0 速度控制位。置 1 定时器 0 工作在 1T 模式，置 0 定时器 0 工作在 12T 模式；

T1x12：定时器 1 速度控制位。置 1 定时器 1 工作在 1T 模式，置 0 定时器 1 工作在 12T 模式。

在对单片机的时间规则和定时器的工作模式有一定了解后，我们再看看单片机的定时器/计数器的工作原理。

实验板上的 stc12 内置两个定时器/计数器 T0 和 T1（详见中文参考手册 P241）。之所以叫“定时器/计数器”，因为它是以计数器为基础的定时器。因为我们用它的定时功能甚多于计数功能，所以更多时候我们简称它为“定时器”。

定时器 T1 和 T0 都是 16 位的计数器，各由两个 8 位的寄存器组成。当定时器启动后，当接收到一个计数脉冲时计数器就自动加一。计数器就像一个水桶，启动定时器就是给这个水桶不断加水。当水桶装满溢出，也就相当于计数器加满溢出时，T1 或 T0 将自动清零并向单片机发出中断请求。

单片机复位后，AUXR 寄存器的第 6 位 T0x12 和第 7 位 T1x12 默认为 0。这样周期为 $1/12 \times 10^{-6}$ 秒的机器周期脉冲在给定时器前先进行一次 12 分频，让它慢 12 倍后再送给定时器充当计数脉冲。这样就相当于 12 个时钟周期等于 1 个机器周期。换言之定时器的计数脉冲周期为 1us，即定时器每 1us 加 1。如果我们通过软件将 AUXR 寄存器的第 6 位 T0x12 或第 7 位 T1x12 置 1，则相应定时器将工作在 1T 模式：周期为 $1/12 \times 10^{-6}$ 秒的机器周期脉冲会直接给相应的定时器，相应定时器的计数脉冲周期也就变成 $1/12 \times 10^{-6}$ 秒，定时器每 $1/12\text{us}$ 加 1，计数速度比默认的 12T 模式快 12 倍。

接下来，我们就通过编程，看看通过编程如何使用单片机的定时器。

2.2 软件实现

我们新建一个名为“定时器”的工程，在工程下创建 main.c 文件，写入下列代码：

```
#include "stc12.h"

#define uint unsigned int

void Timer_Init();

sbit beep = P0^3;

uint a;

void main()
{
    beep = 1;

    Timer_Init();

    while(1)
    {
        if(a == 20)//定时器每 20 次，即 1s 进入一次该函数
        {
            a = 0;//清零，为下一次计时做准备

            beep = ~beep;
        }
    }
}
```

```

}

//定时器初始化函数
void Timer_Init()
{
    TMOD = 0X01;
    TH0 = (65536 - 50000) / 256;
    TL0 = (65536 - 50000) % 256;

    TR0 = 1;
    ET0 = 1;
    EA = 1;
}

//定时器 0 中断服务函数
void Timer0_Handler() interrupt 1
{
    a++;
}

```

使用定时器 T0 和 T1 我们需要操作中断允许寄存器 IE、定时器/计时器工作模式寄存器 TMOD、定时器/计时器控制寄存器 TCON、计时器高 8 位 THx 和计时器低 8 位 TLx。

我们先看看 1 个新的特殊功能寄存器：定时器/计时器工作模式寄存器 TMOD。它在参考手册 P243，相关位定义如下表：

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	GATE	C/T	M1	M0	GATE	C/T	M1	M0

表 3.2.2 TMOD 相关位定义

它是 1 个 8 位寄存器，低 4 位中的 M1、M0 用来设置定时器 0 的工作方式，高 4 位中的 M1、M0 用来设置定时器 1 的工作方式。定时器 0 有 4 种工作方式，而定时器 1 只有 3 种工作方式，具体工作方式和相应设置如下表：

M1	M0	定时器工作方式
0	0	13 位定时器/计数器，只用到 THx 全部 8 位和 TLx 低 5 位
0	1	16 位定时器/计数器，THx 和 TLx 全用
1	0	8 位自动重装载定时器，溢出时 THx 中存放的值自动装入 TLx

1	1	对于定时器 0，此设置无效，停止计数；对于定时器 1 为双 8 位定时器：TL0 作为一个 8 位定时器/计数器，由定时器 0 的控制位控制；TH0 仅作为一个定时器，由定时器 1 的控制位控制
---	---	---

表 3.2.3 M1、M0 与定时器工作方式的关系

这里我们都用到 T0 的工作方式 1——16 位定时器/计数器，即 THx 和 TLx 全用。于是我们给 TMOD 的低 4 位赋二进制 1，即 00000001，也就是十六进制 0X01。

在设置好定时器的工作方式后，我们就要给计数器填入初值。我们采用计数器全部 16 位，它的容量为 2^{16} ，即 65536。假如我们将计数器从零开始计数，计数器每 10^{-6} 秒加一，定时器一次最多只能定时 65536×10^{-6} 秒，约 0.066 秒。0.066 秒一眨眼的功夫，根本没多大用处嘛。于是我们试着给 16 位计数器填入 65536-50000。想一下，计数器要加到 65536 时才溢出，如果我们向计数器填入 65536-50000，它是不是要加 50000 次才溢出，每次溢出用的时间为 50000×10^{-6} 秒，即 0.05 秒。以上述例程为例，每次进入定时器中断服务函数 Timer0_Handler()，变量 time 都会加 1。换言之只要我们打开定时器，每 0.05 秒 time 加 1。然后我们再在定时器中断服务函数函数中判断 time 的值。如果我们要隔 1 秒将蜂鸣器的工作状态翻转一次，也就是说定时器要溢出 20 次，我们就判断 time 是否为 20 然后翻转一次蜂鸣器的工作状态。注意！判断 0 为 20 进入 if 函数后，第一步要做的就是把 time 清零。否则 time 继续从 20 往上加，就再也进不了 if 判断了。

但是 16 位计数器是由两个 8 位计数器——低 8 位计数器 TLx 和高 8 位计数器 TH0 组成的。为了把 65536-50000 的高 8 位和低 8 位提取出来，我们用到了一个小小的方法。把 65536-50000 整除 (/) 2^8 ，即 256 得高 8 位，把 65536-50000 求余 (%) 256 得低 8 位。我们以 11001010 为例：

11001010 十进制为 202。 $202/2^4$ 得 12 即 1100， $202\%2^4$ 得 10 即 1010。1100 和 1010 拼起来不正是 11001010 吗。

THx 和 TLx 除了在模式 2 下外每次溢出后都会自动清零，所以每次定时器中断函数都要重新填入初值！

万事俱备，我们就要准备启动定时器了！我们在准备篇第三节中讲到 TCON 和 IE 是可以位寻址的，因此我们可以直接操作它们的某个位而无需声明。

启动定时器需要设置定时器/计时器控制寄存器 TCON 的 TRx 位：置 1 启动，置 0 关闭。

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

表 3.2.4 TCON 相关位定义

TRx : 定时器 x 运行控制位。置 1 启动定时器 x , 置 0 关闭定时器 x。

但是，仅仅启动了定时器，定时器溢出后给单片机发送中断请求后，如果中断开关没打开，那单片机怎么会受理中断请求呢？中断允许寄存器 IE 已经在我们上一节的外部中断中提及。因为计数器溢出时会向单片机发出中断请求，所以要分别给 IE 的定时器控制中断位 ETx 和总中断控制位 EA 赋 1，打开定时器中断和总中断。这样当计数器溢出时，向单片机发出中断请求后，单片机才会执行中断服务函数。

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	EA	ELVD	EADC	ES	ET1	EX1	ET0	EX0

表 3.2.5 IE 相关位定义

EA : 总中断允许控制位。置 1 打开总中断，置 0 关闭总中断；

ETx : 定时器 x 溢出中断允许位。置 1 当定时器 x 溢出时能触发中断，置 0 当定时器 xy 溢出时不会触发中断。

2.3 下载验证

把工程编译生成的 hex 文件下载进单片机，结合程序和现象分析蜂鸣器的发声规律以及是怎样实现的。

2.4 思考总结

在使用定时器时最容易犯的错误之一是没有在定时器中断服务函数中给计数器重新赋值。

定时器处理模式 2——8 位自动重装载定时器外，其他三个模式下，计数器溢出后就会清零。如果不重新填上初值的话，第二次起的计时就会从零开始。那每次都是计数 65536 次，即 0.066 秒后进入中断服务函数。这样一来定时器就失去意义了。

至于 8 位自动重装载模式，我们只用到低 8 位计数器。在配置定时器时我们先给低 8 位计数器和高 8 位计数器填入相同的值。当低 8 位计数器溢出时，高 8 位计数器的值会自动装入低 8 位计数器中，完成自动重装载。

另外一个容易出错的地方是，定时器中断服务函数中对 time 进行判断后没有清零。如果不及时清零 time，那就永远只能执行一次 if 判断了。

三、PWM 进阶——控制技术中的万金油 2

还记得在入门篇第四节中，我们利用 for 延时函数模拟出简单的 PWM 脉冲信号。但这种方法实际应用起来很不方便。例如有些外设，你需要不断给它一个稳定的 PWM 信号来驱使它工作。这时如果你要操作外设又要执行其他工作时，在程序的设计上既要不断循环执行 PWM 函数，又要执行其他代码，在编程上就会举步维艰。还有一种办法，那就是上一节的思考拓展中提到的用到定时器产生 PWM 信号。因为定时器是在后台工作的，因此 PWM 信号的产生不会和其他代码的执行发生冲突。但是定时器的资源是很珍贵的！因为定时器的功能强大，很多外设都需要用到定时器。但定时器数量是有限的：只有 T0 和 T1，因此我们要把定时器用在最需要的地方。那有没有办法不用定时器，就能在后台自动产生 PWM 信号呢？那我们就要借助 PCA 模块了！

3.1 知识点讲解

PCA (Programmable Counter Array)，中文全称可编程计数器阵列。它可以用于软件定时器、外部脉冲的捕捉、高速输出以及我们本节要讲的脉宽调制 (PWM) 输出。STC12 芯片在 STC89C51 的基础上新增了两路 PCA。关于 PCA 的详细资料，在中文参考手册 P356。下面我们就着重讲解如何用 PCA 模块实现 PWM 信号的输出。

PCA 含有一个特殊的 16 位定时器，且有两个 16 位捕获/比较模块（两路，一路一个）与之相连，如下图：

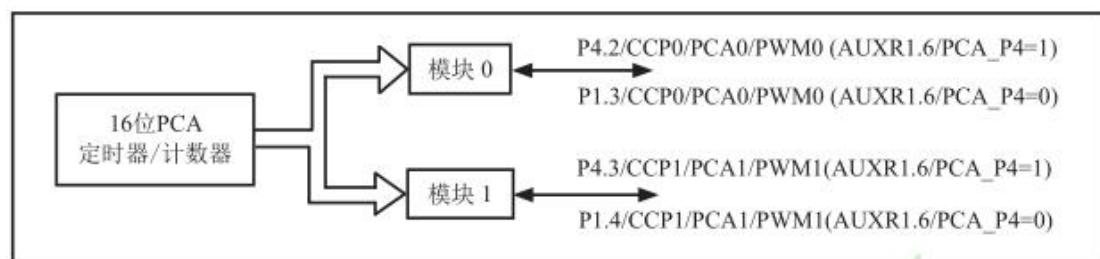


图 3.3.1 PCA 结构简图

两个模块都可以分别通过设置 PCA 比较/捕获寄存器 CCAPM0 (对应模块 0) 和 CCAPM1 (对应模块 1) 来设置成 4 种工作模式：上升/下降沿捕获、软件定时器、高速输出或 PWM 输出。其中，模块 0 的输出与单片机的 P1.3/PWM0 连接 (可切换成 4.2)，二模块 1 的输出与单片机 P1.4/PWM1 连接 (可切换成 4.3)。

16位PCA定时器/计数器是2个模块的公共时间基准，是两个模块得以工作的“心脏”，它的逻辑图如下：

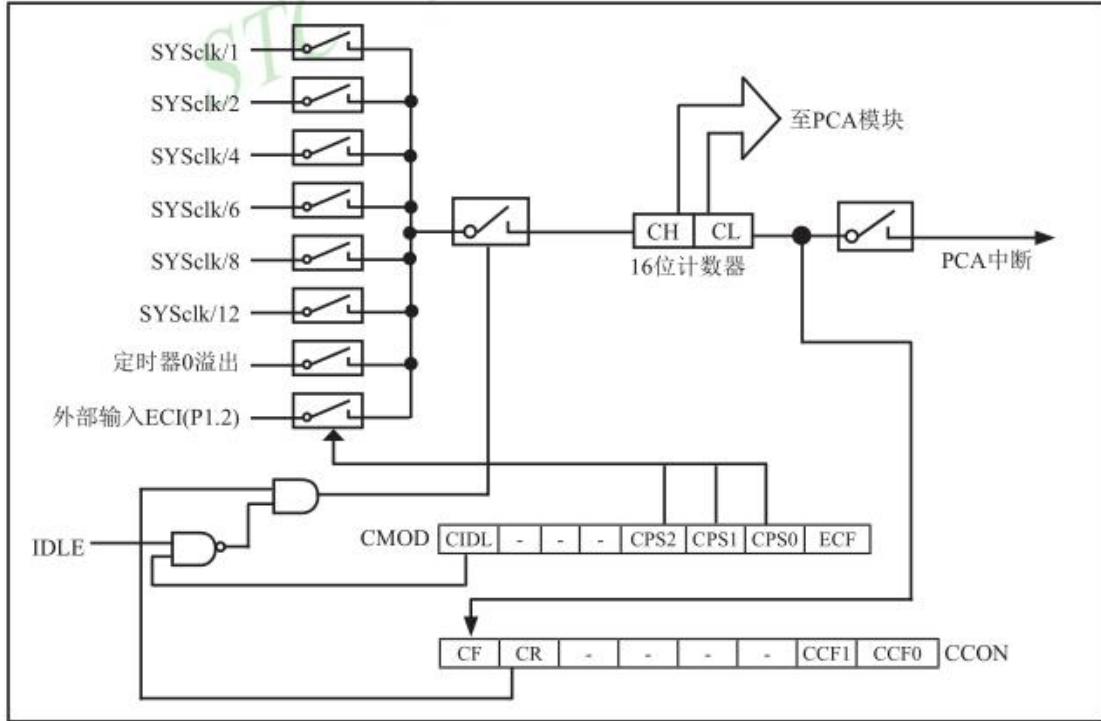


图 3.3.2 PCA 计数器逻辑图

还记得单片机定时器由两个8位计数器：THx 和 TLx 组成吗？16位PCA定时器/计数器同样由两个8位的计数器，不过为了区分，我们叫它们CH和CL。PCA的计数器和定时器的计数器一样，接受一个计数脉冲时就加一。但与定时器的计数频率只能是系统时钟频率或系统时钟频率的12分频不同，PCA定时器的计数频率可以通过设置工作模式寄存器CMOD的CPS0位、CPS1位和CPS2位这3位来设置成：1/12系统时钟频率、1/8系统时钟频率、1/6系统时钟频率、1/4系统时钟频率、1/2系统时钟频率、系统时钟频率、定时器0溢出频率或ECI脚的输入。和定时器模式2一样，两个计数器只有CL工作，当CL溢出时将清零然后CH中的值将自动赋给CL。

我们再看看 PWM 模式下模块 0、模块 1 的内部结构：

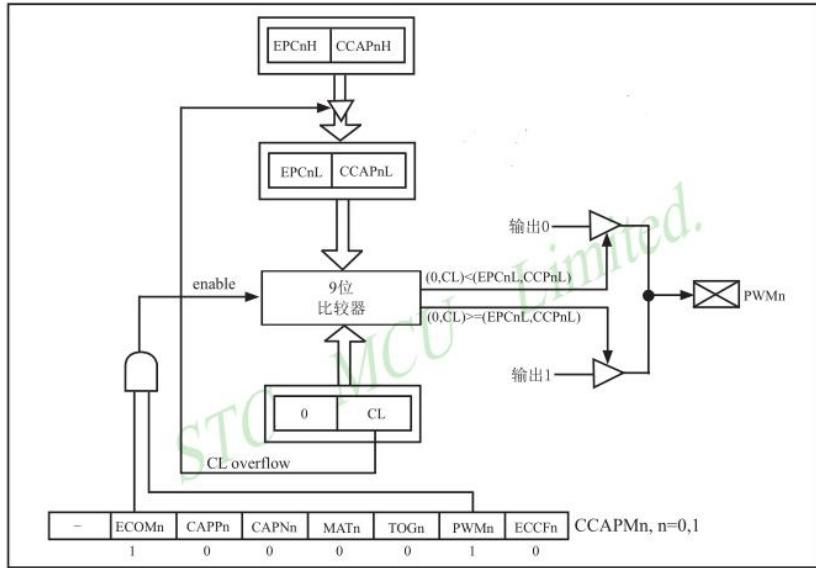


图 3.3.3 PCA 比较器逻辑图

我们知道 PWM 信号是由高电平和低电平组成的。而 PCA 模块在 PWM 模式下何时输出高电平何时输出低电平，是由[EPCxL,CCAPxL] (EPCxL 位与 CCAPxL 寄存器 8 位组成的 9 位) 与[0,CL] (0 与 CL 寄存器 8 位组成的 9 位) 之间的大小关系决定的：当 CL 加一时，就会判断[0,CL]与[EPCxL,CCAPxL]之间的大小关系：**当[0,CL]小于[EPCxL,CCAPxL]时输出低，当[0,CL]大于或等于[EPCxL,CCAPxL]时输出高。**当 CL 溢出清零后，[EPCxH,CCAPxH]会自动填充到[EPCxL,CCAPxL]中，实现无干扰更新 PWM 输出。所以实际使用中比较的是[EPCxL,CCAPxL]和[0,CL]之间的大小关系。可见，只要改变[EPCxL,CCAPxL]的值，就能改变 PWM 信号的占空比。

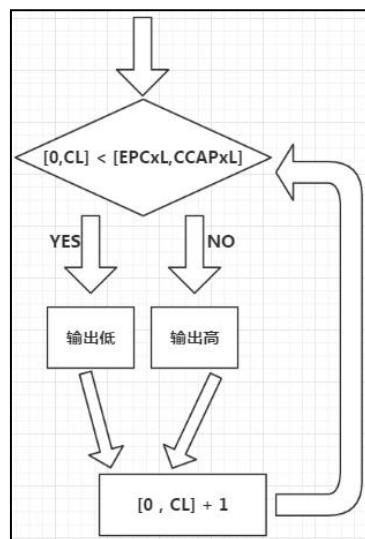


图 3.3.4 PCA 比较器判断图

至于 PWM 信号的频率，其实我刚刚已经提到了。计数器 CL 每次加一就比较一次，溢出后会自动清零，再重新下一次的计数。这样一来，CL 的溢出频率不就是 PWM 信号的频率吗？又因为 CL 是一个 8 位寄存器，从 0 加到溢出需要加 256 次，因此：

$$\text{PWM 频率} = \text{计数源频率}/256$$

但这样就又有一个问题了。我们上面提到，模块 0 和模块 1 是公用 1 个时间基准的，所以模块 0 和模块 1 输出的 PWM 信号频率是相同的。但它们的占空比却可以不一样，因为两个模块分别对应了[EPC0L,CCAP0L]和[EPC1L,CCAP1L]。

下面我们就通过程序，看看如何用 PCA 模块输出 PWM 信号。

3.2 软件实现

我们新建一个名为“PWM 进阶”的工程，在工程下创建 main.c 文件，写入下列代码：

```
#include "stc12.h"

void PCA_Init();

void main()
{
    PCA_Init();

    while(1);
}

//PCA 初始化函数
void PCA_Init()
{
    CCON = 0X00;//初始化 CCON
                //关闭 PCA 计数器

    //清零计数器
    CL = 0;
    CH = 0;

    CMOD = 0X0E;//设置 PCA 工作模式
                //空闲模式下 PCA 计数器继续工作
                //PCA 计数器计数频率为系统时钟/8
                //关闭 PCA 计数器溢出中断
```

```

CCAP0H = CCAP0L = 0XF3;//PWM0 输出频率为 5.8KHz , 占空比为 5%的 PWM 信
号
CCAPM0 = 0X42;//模块 0 为 PWM 输出模式
    //不翻转 , 在 P1.3 输出
    //没有 PCA 中断

CR = 1;//启动 PCA 计数器
}

```

进入主函数后，我们先把 CCON、CL、CH 清零初始化。这是一个编程很好的习惯。因为你不知道在你进行这步操作前，这些寄存器原先都装了什么值。养成这个好习惯，你会在编写一些大型程序中避免一些关键却很难发现的错误。

我们配置 CCON 只需设置它第 6 位：PCA 计数器控制位 CR。CR 位置 1 启动 PCA 计数器，置 0 关闭 PCA 计数器。我们现在要配置 PCA 定时器的相关设置，当然要确保它处于关闭状态，等我们设置好后再开启它。关于 CCON 的位定义，详见中文参考手册 P358。与本节相关的位定义如下：

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	CF	CR	-	-	-	-	CCF1	CCF0

表 3.3.1 CCON 相关位定义

CR：PCA 运行控制。置 1 启动 PCA 计数器，置 0 关闭 PCA 计数器。

接下来我们就要配置 PCA 定时器的工作模式。

CMOD 的第 7 位是设置单片机处于空闲状况下，PCA 计数器是否继续工作，我们置 0 设置成继续工作。所谓空闲模式，是指单片机进入休眠状态，除 CPU 停止工作外其他硬件处于活动状态；CMOD 的第 1、2、3 位是设置 PCA 计数器计数源的，我们这里设置成系统时钟的 1/8，又因为 PWM 频率为 PCA 计数源频率/256，因此产生的 PWM 信号的频率为 5.8KHz；PCA 计数器在 PWM 模式下是没有溢出中断的，因此 CMOD 的第 0 位置 0。关于 CMOD 的位定义，详见中文参考手册 P357。与本节相关的位定义如下：

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	CIDL	-	-	-	CPS2	CPS1	CPS0	ECF

表 3.3.2 CMOD 相关位定义

CIDL : 空闲模时 PCA 工作控制位。置 1 当单片机进入空闲模式时 PCA 不工作 , 置 0 当单片机进入空闲模式时 PCA 将继续工作 ;

CPS2、CPS1、CPS0 : PCA 计数脉冲源控制位。三个位的值共同决定 PCA 计数脉冲源 , 对应关系如下表 :

CPS2	CPS1	CPS0	PCA 计数脉冲源
0	0	0	系统时钟 / 12 , SYSclk / 12
0	0	1	系统时钟 / 2 , SYSclk / 2
0	1	0	定时器 0 溢出脉冲
0	1	1	ECI / P1.2 (或 P4.1) 脚输入的外部时钟
1	0	0	系统时钟 , SYSclk
1	0	1	系统时钟 / 4 , SYSclk / 4
1	1	0	系统时钟 / 6 , SYSclk / 6
1	1	1	系统时钟 / 8 , SYSclk / 8

表 3.3.3 PCA 计数脉冲源的选择

ECF : PCA 计数溢出中断允许位。置 1 允许 PCA 计数溢出中断 , 置 0 关闭 PCA 计数溢出中断。

然后我们就要给 CCAPxH 和 CCAPxL 填值了。CCAPxL 是 8 位寄存器 , 它的容量是 2^8 即 256 。我们现在要输出占空比为 50% 的 PWM 信号 , 那我们就要给 CCAPxL 填 128 。 CL 从零开始累加 , 一开始 CL 是小于 CCAPxL 的 , 因此输出的是低电平 ; 当 CL 加到大于或等于 CCAPxL 时 , 输出高电平。所以如果给 CCAPxL 填入 256 的一半 128 , 那每个 CL 计数器周期内 , 输出高电平和低电平的时间就一样了 , 即占空比为 50% 。而 128 转换为十六进制就是 0X80 。

咦 ? 等等 , 我们不是要比较 [0,CL] 与 [EPCxL,CCAPxL] 之间的大小关系吗 ? 那为什么我们现在只判断 CL 和 CCAPxL 的大小关系呢 ? 原来第 PCA_PWMx 的第 0 位 EPCxL 是用来固定输出高和低时用到的 :

当 EPCxL = 0 且 CCAPxL = 00H 时 , PWM 固定输出高

当 EPCxL = 1 且 CCAPxL = FFH 时，PWM 固定输出低

所以除非我们给 PWM 固定输出高电平或低电平，我们实际上只用到了 CCAPxL 和 CL 作比较。

至于给 CCAPxH 和 CCAPxL 赋相同的值的原因，我们在上面已经说过。当 CL 溢出时，CCAPxH 的值会自动填入 CCAPxL 中，保证无干扰更新 PWM 输出。

从上面我们就已经可以大致得出 CCAPxL 的大小与 PWM 占空比之间一个重要关系：
CCPAxL 越大，输出的 PWM 占空比越小；反过来 CCPAxL 越小，输出的 PWM 占空比越大！所以大家使用 PCA 产生 PWM 时不要弄反了。

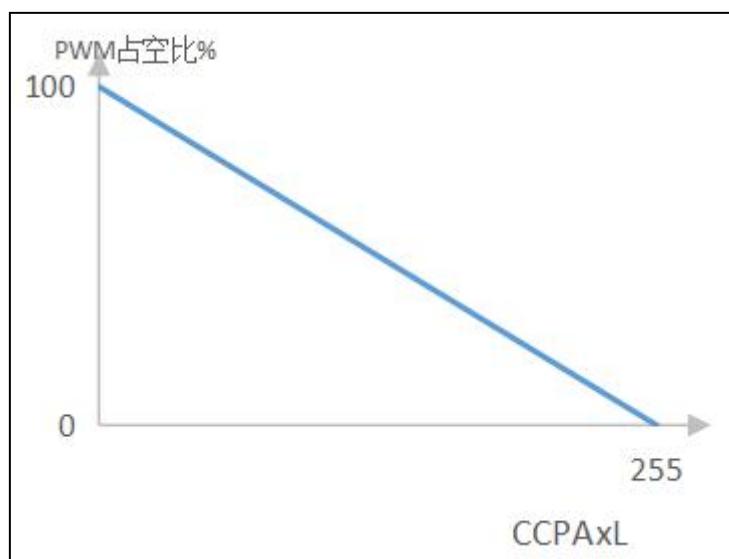


图 3.3.5 PWM 占空比与 CCAPxL 的关系

最后我们要配置比较/捕获寄存器 CCAPMx。第 1 位 PWMx 控制 PCA 模块的 PWM 模式，置 1 把模块 x（分别为模块 0 和模块 1）设置成 PWM 模式；第 6 位 ECOMx，置 1 打开模块的比较器功能，这样 CL 和 CCAPxL 才能开始比较。关于 CCAPMx，可以查看中文参考手册 P358。与本节相关的位定义如下：

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	-	ECOMx	CAPP1x	CAPN1x	MATx	TOGx	PWMx	ECCFx

表 3.3.4 CCAPMx 相关位定义

ECOMx：比较器控制位。置 1 打开比较器，置 0 关闭比较器；

PWMx：PWM 模式控制位。置 1 将模块 x 设置成 PWM 输出模式

配置好相关寄存器后，我们就将 PCA 控制寄存器 CCON 的 PCA 计数器控制位 CR 打开，启动 PCA 模块，对应的输出端口就可以产生 PWM 信号了。

3.3 下载验证

把工程编译生成的 hex 文件下载进单片机。但 PCA 的输出端 P1.3、P1.4、P4.2 和 P4.3 接的外设我们都还没接触到。所以我们用洞洞板和面板板扩展的方式观察 PWM 的产生情况。

洞洞板和面板板是电子设计中必不可少的工具。通过洞洞板或面板板我们可以迅速根据一个电路图搭建一个电路用于各种实验。

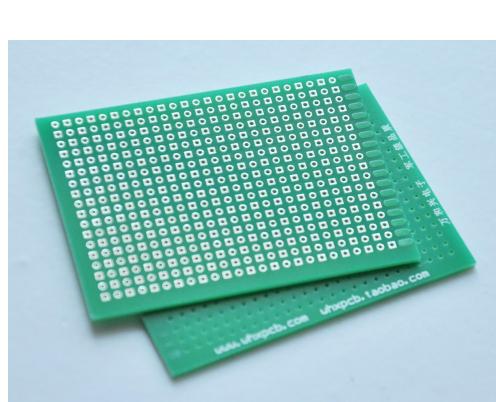


图 3.3.6 洞洞板



图 3.3.7 面板板

洞洞板叫洞洞板，就因为上有一个个规则排布的小洞。一个面上这些小洞上都有银色或黄色的焊盘。我们设计好电路后，就将相应的元器件布局好后插在洞洞板上。然后我们在有焊盘的一侧用焊锡把各个元件按电路图连接好。

而面板里面已将步好了导线。以图 2.3.2 中的面板板为例。面板板中间有一道分隔区。分隔区两侧每行的孔都是导通的，且各行之间和分隔区之间是不导通的。我们使用面板板时只需将元件按电路连接要求插在面板板上即可搭建出一个电路。



图 3.3.8 面包板背面铁片排布

洞洞板和面包板各有优劣。因为元件的位置可以自己选择，所以洞洞板搭建出来的电路清晰明了。但搭建和修改需要焊锡，故过程繁琐；而面板版搭建和修改都很方便，只需将元件插上去。但面包板因为它内部的导线是已经固定的，所以你搭建的电路并不一定能直观反应出电路的连接情况。

我们就分别用洞洞板和面包板搭建一个简易的 LED 电路，用于观察 PCA 产生 PWM 的情况。简易 LED 电路如下图：

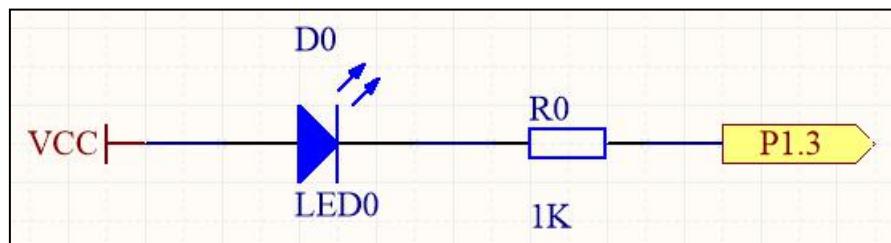


图 3.3.9 LED 电路图

关于焊锡，大家可以上网看一下相关教学。使用起来一定要避免烫伤和火灾。

用洞洞板和面包板搭建的电路成品如下图：

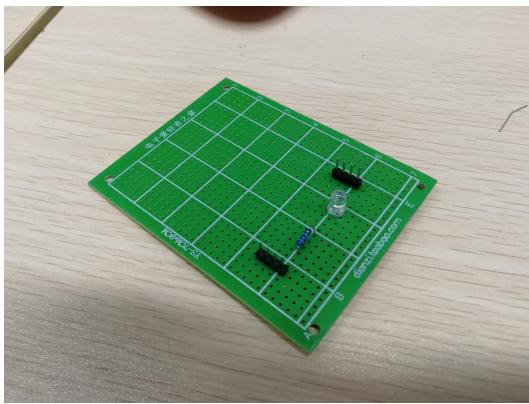


图 3.3.10 洞洞板搭建出的成品正面

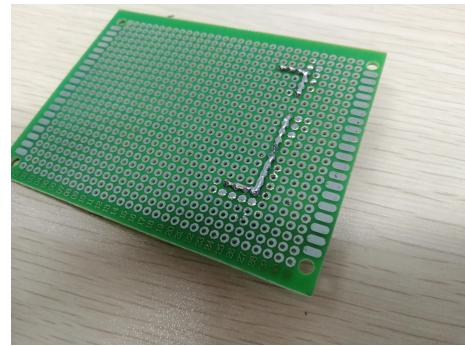


图 3.3.11 洞洞板搭建出的成品背面

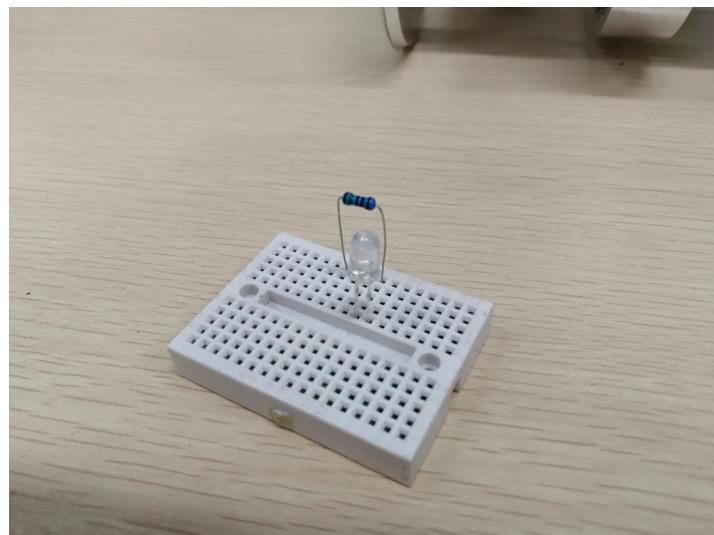


图 3.3.12 面包板搭建出的成品

通过杜邦线，我们将 LED 的负极接实验板的外接引脚 P1.3，LED 串联电阻后接实验板外接电源的正极。观察 PWM 占空比的变化对 LED 亮度的影响。

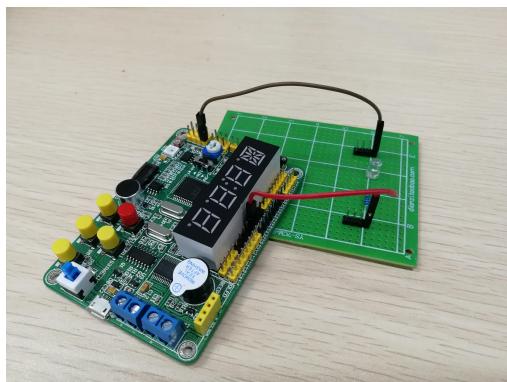


图 3.3.13 连接好后的洞洞板

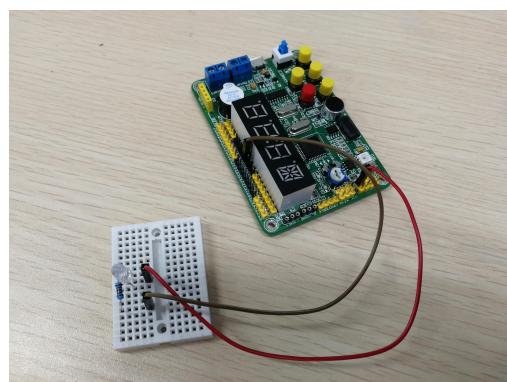


图 3.3.14 连接好后的面包板

3.4 思考总结

鉴于 PCA 的配置有点复杂，这里我们再次理清一下配置思路：

- ①清零 CCON，关闭 PCA 计数器；
- ②清零 PCA 计数器；
- ③配置 CMOD，设置 PCA 工作模式；
- ④配置 CCAPxL 和 CCAPxH，设置 PWM 占空比；
- ⑤配置 CCAPMx，设置 PWM 输出和输出引脚；
- ⑥启动 PCA 计数器。

对比入门篇的 PWM 入门，有没有发现我们用 PCA 产生的 PWM 控制 LED 亮度变化更明显。这是频率决定的。在 PWM 入门中，我们产生的 PWM 频率只有 50Hz，而我们现在产生的 PWM 频率达到 5.8KHz。当 PWM 频率较低时，LED 的刷新速度也较慢。但随着频率变高，LED 的刷新速度也就越快，这样 LED 的亮度控制就能更加细腻。

另外和 PWM 占空比的控制精度也有关系。在 PWM 入门中，PWM()函数的型参加 1，产生 PWM 的占空比就加 5%。但用 PCA 产生 PWM，因为 PWM 的占空比和 CCAPxL 有关。PWM 占空比从 0% 到 100% 对应 CCAPxL 的 0 到 255。这样 CCAPxL 每加 1，PWM 的占空比就改变 0.39%。显然用 PCA 产生的 PWM 在占空比的控制上精度更高，相应能控制 LED 亮度的精度也更高。

四、串口通讯——来和单片机悄悄话

有没有发现，在之前的篇幅中，我们与单片机的交流都是单向的——要不就仅仅将信息反馈给单片机，例如按键控制 LED；要不就仅仅单片机把信息反馈给我们，例如 PWM 输出控制。那有没有办法让单片机实现与外界环境的双向通讯呢？如果可以实现，那意义非常重大。想想看，你和别人交流，如果一个人只说，另一个人只听，那样子唱独角戏，你们两个人根本谈不成任何事情。如果实现与单片机的双向通讯，你不仅仅可以控制单片机，同时单片机还能及时把一些信息反馈给你，这在单片机间通讯、程序调试等很多地方都很有实用意义。在这节中，我们就通过单片机的串口功能，完成计算机与单片机的双向串口通讯。

4.1 知识点讲解

我们实验板使用的单片机 STC12 具有两个采用 UART (Universal Asynchronous Receiver/Transmitter) 工作方式的全双工串行通信接口：串口 1 和串口 2。每个串口都由两个数据缓冲器、1 个位移寄存器、1 个串行控制寄存器和 1 个波特率发生器等组成。关于串口的详细资料，可查看中文参考手册 P280。

通过软件配置，每个串口都有 4 种工作模式，其中两种方式的波特率可变，另两种波特率是固定的。作为一种双向通讯功能，串口 1 的发送引脚是 TxD/P3.1，接收引脚是 RxD/P3.0。而串口 2 的发送引脚和输出引脚可以通过设置相关寄存器在 P1 和 P4 之间任意切换（这里说发送和接收是站在单片机的角度看的，这点要搞清楚）。在本节，我们着重讲解串口 1 模式 1：8 位 UART，波特率可变。

等等，串口是什么？波特率又是什么鬼？上面提到的术语有点多，我们要先逐一了解一下。因为这些术语会决定了我们接下来能不能一起好好地玩耍。

首先什么是全双工串行通信，这里包含了两个概念：

串口其实也叫串行口，通过串行口的通讯也叫串行通讯。串行通讯是和并行通讯相对的，我们从名字上就可以知道：串行通讯的数据是以位 (bit) 为单位一个接一个发送接收的，数据就像在单车道公路上传输；而并行通讯数据是多位的一起发送接收的，数据就像传输在多车道的公路上。可见，并行通讯虽然传输速度快，但需要更多的数据线。而串行通讯与反之，虽然传输速度稍慢，但需要的数据线也相对较少。所以一般短距离的通讯可以选择并行通讯，而远距离的通讯一般会选择串行通讯。

串行数据是一节一节传输的。传输时由 1 位起始位 “0”、8 位数据和 1 位终止位 “1” 组成。当没传输时，电平为 1，称为标记。而传输完成后的 0 称为空格。

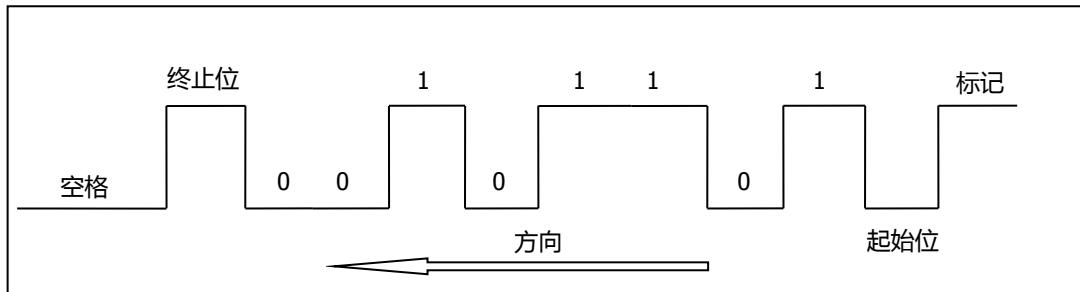


图 3.4.1 串口通讯数据格式

而全双工是描述通讯时发送和接收的时间关系：全双工是发送和接收同时进行；半双工是发送和接收分开进行；单工是只能发送或只能接收。

接着，波特率 (Baud Rate) 可以说是串口通讯的灵魂所在。继续以公路为例（这不就应合了数据通讯像是数据在一条“公路”上奔跑吗？）。正常情况下，一条不长的公路它两端的限速是一致的，这样才能保证公路的正常通行。不然一头限速 80 公里，另一端突然限速 20 公里，岂不是会造成混乱？那我们现在通过串口然单片机与计算机实现通讯，那两端数据的传输速度也应当保证一致。不然一方发送很快而另一方接受很慢，那样不就发生大混乱？波特率就是描述数据传输快慢的量度，它的单位是 bps (Bit Per Second)，即每秒传输数据的位数。我们通过软件设定单片机串口通讯的波特率，然后在计算机端的串口调试助手用相同的比特率。这样就能保证单片机与计算机实现稳定通讯了。

在给单片机设置波特率时，要涉及到一些计算。串口 4 种模式的比特率计算有差别：

模式 0：

模式 0 的波特率是固定的，但和辅助寄存器 AUXR 的第 5 位 UART_M0x6 位有关。

$$\text{当 } \text{UART_M0x6} \text{ 位置 1 时，波特率} = \frac{\text{SYSclk}}{12}$$

$$\text{当 } \text{UART_M0x6} \text{ 位置 0 时，波特率} = \frac{\text{SYSclk}}{2}$$

我们之前已经提到，SYSclk 是系统时钟频率，对于我们的实验板就是晶振频率 12MHz，下同。可见，当 UART_M0x6 位置 0 时，数据传输速度是置 1 时的 6 倍。

模式 2：

模式 2 下，波特率也是固定的，它不仅与 SYSclk 有关，还与电源控制寄存器 PCON 的第 7 位 SMOD 位有关。波特率 = $\frac{2^{SMOD}}{64} * SYSclk$

当 SMOD 位置 1 时，波特率 = $\frac{1}{32} * SYSclk$

当 SMOD 位置 0 时，波特率 = $\frac{1}{64} * SYSclk$

可见，当 SMOD 位置 1 时，波特率是置 0 时的两倍。

模式 1 和模式 3：

模式 1 和模式 3 下的串口通讯的波特率是可以由我们自己定的。

波特率 = $\frac{2^{SMOD}}{32} * \text{定时器1溢出率}$

其中，定时器 1 的溢出率就是定时器 1 溢出的频率，它不仅和定时器 1 所装的值有关，还与辅助寄存器 AUXR 的第 6 位 T1x12 位有关。

当 T1x12 置 0 时，定时器 1 溢出率 = $\frac{SYSclk}{12 * (256 - TH1)}$

当 T1x12 置 1 时，定时器 1 溢出率 = $\frac{SYSclk}{(256 - TH1)}$

如今我们用串口 1 模式 1，所以配置时离不开定时器 1。串口在传输数据时，为了让它的传输更加稳定，我们不得不借助高精度的定时器。这里我们只用到定时器 1 的模式 2——8 位自动重装载定时器。当我们选定一种波特率时，代入上述公式，就可以轻松求得我们要给定时器 1 低 8 位 TL1 填入的值了。

4.2 软件实现

51 系列单片机实现串口通讯，主要有两种办法：中断法和扫描法。这个与按键检测有着异曲同工之妙。我们就分别用中断法和扫描法，看看如何通过软件实现单片机和计算机间的串口通讯。

我们先来看看扫描法：

```
#include"stc12.h"

void USART_init();//串口初始化函数

void main()
{
    USART_init();//串口初始化

    while(1)
    {
        if(RI == 1)//检测串口 1 接收中断标志位
        {
            RI = 0;//清零串口 1 接收中断标志位
            P0 = SBUF;
        }
    }

//串口初始化函数
void USART_init()
{
    SM0 = 0;//设置串口 1 工作方式为 8 位 UART，波特率可变
    SM1 = 1;
    REN = 1;//打开串口接收允许

    TMOD = 0X20;//设置定时器 1 工作模式 2
    TH1 = 0xFD;//给 TH1、TL1 装入初值，波特率为 9600
    TL1 = 0xFD;

    TR1 = 1;//启动定时器 1
}
```

程序一开始，我们先要对串口相关设置进行初始化。

我们只用到了定时器 1 的低 8 位，因此我们选择定时器 1 工作模式 2。定时器 1 在工作模式 2 下，只在低 8 位 TH1 进行累加。当 TH1 溢出时，将高 8 位 TH1 中的值填

入 TL1 中，实现自动填装，继续下一次的计时。所以我们不需要像工作模式 1 一样要在定时器中断服务函数中给计数器重新装载。因此我们在配置时可以给 TH1 和 TL1 赋相同的值。

一般串口通讯波特率我们选择 9600。在每次单片机上电后，SMOD 位和 T1x12 位都会复位为 0。我们代入公式后，算得约等于 252.745，这是因为我们选用 12MHz 的晶振作时钟源引起的误差。我们把 252.745 约等于 253，即十六进制的 FD。虽然这样会稍微影响串口通讯的同步性能，但大体上不会影响到正常的通讯。一些单片机实验板会选用 11.0592MHz 的晶振作为时间源，这样就能精确计算出 TH1 要装的初值。但如果用 11.0592MHz 晶振，定时器的计数器就不再是 1us 加 1 了。考虑到定时器比串口用途更广，市面上的绝大多数 51 单片机实验板都采用 12MHz。

在配置好定时器 1 后，我们就要配置串口的相关寄存器了。

首先是打开串口接收允许，它由串口 1 控制寄存器 SCON 的第 4 位 REN 控制，置 1 开启，置 0 关闭。

接着是串口工作模式设置。我们需要设置的同样是 SCON，不过是它的第 6、7 位 SM1 和 SM0。两位二进制，一共可以有 4 种组合，刚好对应串口 4 种工作模式。SCON 可以直接位寻址，不用声明。关于 SCON 详细位定义可以查看中文参考手册 P281。与本节相关的位定义如下表：

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	SM0	SM1	SM2	REN	TB8	RB8	TI	RI

表 3.4.1 SCON 相关位定义

SM0、SM1：决定串口 1 的工作模式：

SM0	SM1	串口 1 工作模式
0	0	模式 0
0	1	模式 1
1	0	模式 2
1	1	模式 3

表 3.4.2 串口 1 工作模式选择

REN : 串口接受控制位。置 1 允许串口接收数据，置 0 禁止串口接收数据；

TI : 串口发送中断请求标志位。当串口发送数据结束时置 1，若清零需要软件清零；

RI : 串口接收中断请求标志位。当串口接收数据结束时置 1，若清零需要软件清零。

在初始化最后，我们要打开定时器 1，开始检测串口是否有接收。

RI 是 SCON 的第 0 位，是串口 1 接收中断标志位。当串口接收到数据结束时置 1。所以在主函数的 while 循环中，我们一直检测 RI 是否为 1。一旦 RI 为 1，说明接收结束，跳入 if 语句。**在跳入 if 语句后要将软件 RI 清零！**否则 RI 将一直为 1，怎样都会进入 if 语句。所以清零是为下一次的中断检测做准备。

接下来我们看到一个我们之前没介绍的寄存器：串行数据缓冲器 SBUF。SBUF 是一个很特殊的寄存器，因为它其实是两个独立的寄存器，但却有同一个名字，在单片机中共用一个地址。SBUF 由一个 8 位接收缓冲器和一个 8 位发送缓冲器组成。接收缓冲器只能读出不能写入，而发送缓冲器反过来只能写入不能读出。这就是为什么它们可以在单片机内共用一个地址。操作 SBUF 也很特别：当 SBUF 在等号右边时，我们操作的是接收缓冲器。当 SBUF 在等号左边时，我们操作的是发送缓冲器。其实这很形象直观：当 SBUF 位于等号右侧时，说明我们要将里面的值赋给某个变量，正好对应着接收缓冲器；而当 SBUF 位于等号左侧时，说明我们要将某个值赋给它，正好对应发送缓冲器。在上面例程中，我们从接收缓冲器读出我们发送的数据，从而控制 LED。

我们再来看看中断法是如何实现串口通讯：

```
#include "stc12.h"

#define uchar unsigned char

void USART_init();//串口初始化函数

void main()
{
    USART_init();//串口初始化

    while(1);
}

//串口初始化函数
void USART_init()
```

```

{
    SM0 = 0;//设置串口 1 工作模式 1
    SM1 = 1;
    REN = 1;//串口允许接收

    TMOD = 0X20;//设置定时器 1 工作模式 2
    TH1 = 0xFD;//波特率 9600
    TL1 = 0xFD;

    ES = 1;//打开串口中断允许
    EA = 1;//打开总中断允许
    TR1 = 1;//启动定时器 1
}

//串口中断服务函数
void USART_handler() interrupt 4
{
    uchar a;

    RI = 0;//清除串口接收标志位

    a = SBUF;
    P1 = a;
    SBUF = a;

    while(!TI);//检测串口发送标志位，当发送完成时 TI 置 1
    TI = 0;//清除串口发送标志位
}

```

在中断法中，串口初始化和扫描法略有不同。因为我们要用到中断，那中断相关寄存器的配置就必不可少了。除了将总中断 EA 置 1 外，我们还有将中断允许寄存器 IE 的第 4 位串口中断允许位 ES 置 1。

interrupt 4 是串口 1 对应的中断服务函数，当单片机接收到数据终止位时，RI 置 1 并跳入串口中断服务函数。进入服务函数后，我们先将 RI 清零。然后我们把接收缓冲器中接收到数据赋给 a，然后把 a 的值分别赋给 P1 和发送缓冲器。这里可以看到，当 SBUF 在等号左侧时，操作的是发送缓冲器。

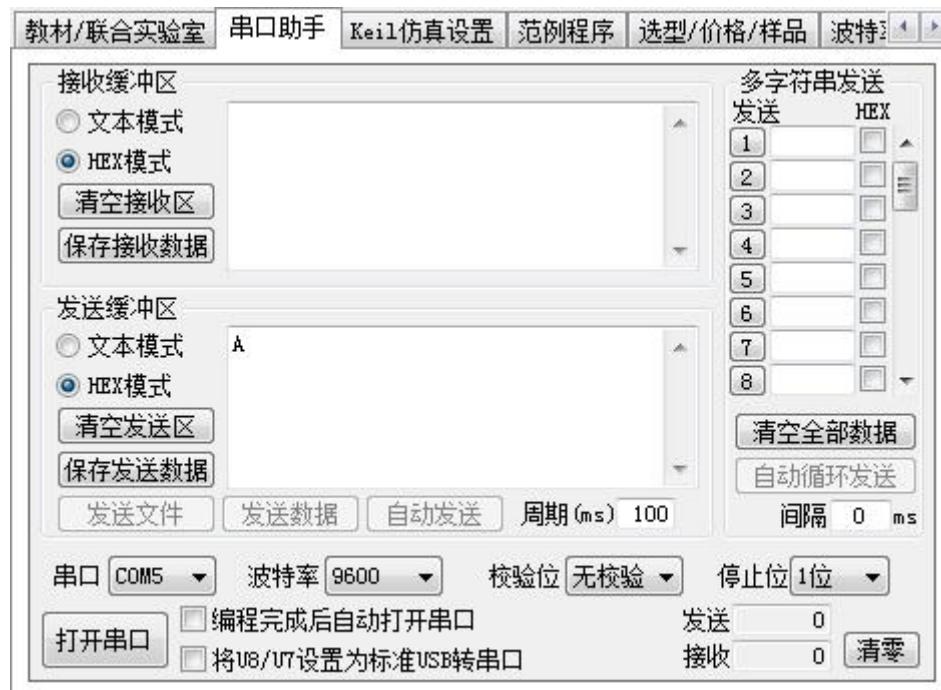
但发送需要时间，你不可能让人家发送到一半就跳出中断干其他事情去了。和接收数据完成后 RI 会硬件置 1 一样，发送数据在发送终止位后，SCON 的第 1 位发送中断请求位 TI 会硬件置 1。利用这点，我们就通过判断 TI 来判断是否发送完数据。

4.3 下载验证

我们把中断法的程序编译成 hex 文件，烧录进单片机中。在保持单片机和计算机连接下，我们通过串口调试助手来实现单片机和计算的串口通讯。

打开光盘中的软件->烧录 STC 单片机软件 stc-isp->stc-isp-15xx-v6.82.exe

在右上方的项目栏中我们选择“串口助手”，进入串口调试界面。



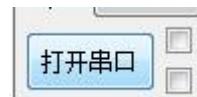
然后我们要选择对应的串口。所谓对应的串口，就是我们单片机下载是的串口。



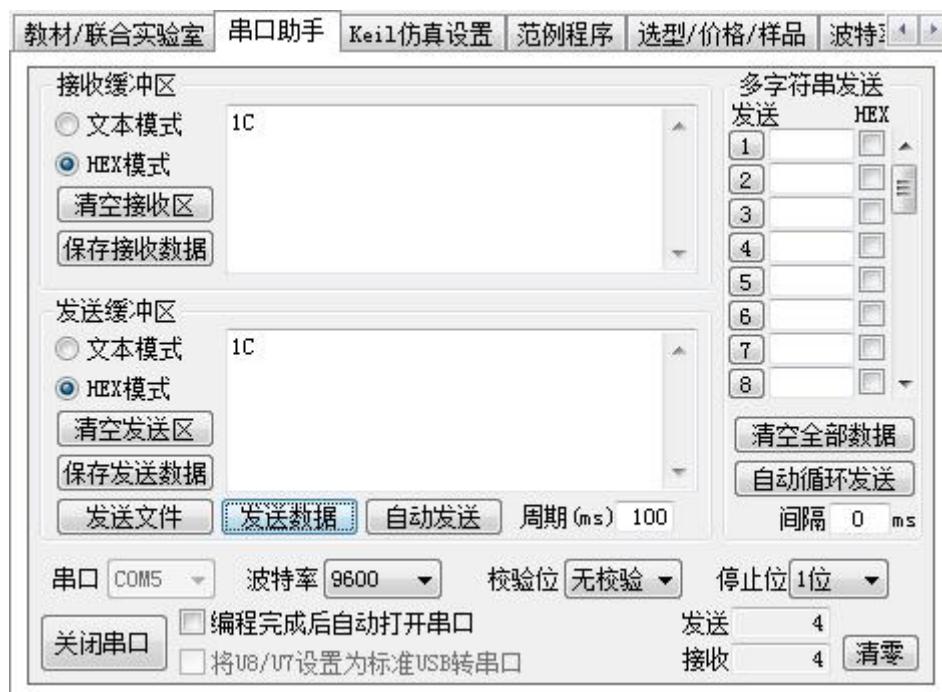
设置串口后，我们还要设置波特率。我们单片机方面波特率为 9600，那计算机方面波特率也要设置成 9600。



完成设置后，我们就打开串口。值得注意的是，在打开串口后，单片机是不能烧录程序的。如果你想烧录其他的程序，那你先要关闭串口。



然后我们试着给串口发送数据。我们发 1 个 1C，观察串口发送回来的数据。



1C 二进制是 00011100，观察 LED 灯的亮灭情况和 00011100 的关系。

4.4 思考总结

不管用扫描法还是中断法，串口相关寄存器的配置大家都要熟记：

①配置 SCON 相关位。SM0、SM1 设置串口工作模式。如果串口要接受数据则要将 REN 位置 1；

②使用模式 0 要配置辅助寄存器 AUXR 的第 5 位 UART_M0x6 位。UART_M0x6 位置 0 比置 1 波特率高 6 倍。复位后 UART_M0x6 位默认为 0；

③其他模式下要配置电源控制寄存器 PCON 的第 7 位 SMOD 位。SMOD 位置 1 比置 0 波特率高 2 倍。复位后 SMOD 位默认为 0；

④使用模式 1 和模式 3 时还要配置辅助寄存器 AUXR 的第 6 位 T1x12 位。T1x12 位置 1 比置 0 波特率高 12 倍。复位后 T1x12 位默认为 0；

⑤如果用中断法，则要将 ES 置 1 打开串口中断，将 EA 置 1 打开总中断；

⑥如果是模式 1 和模式 3，启动定时器，开始串口通讯。

应用篇

大家经历了进阶篇的学习，应该对单片机的几个重要功能都有全新且深刻的理解。但别忘了我们学单片机并不是仅仅摆弄几个按键或几个定时器就能完事的。

我们要真正学以致用，把我们上面学到的知识拓展到更多的领域，让单片机做更多有趣的事情。接下来我们就通过单片机，来操作各种各样的外设，看看单片机有多么的神通广大！

一、数码管静态显示——LED 变身高富帅

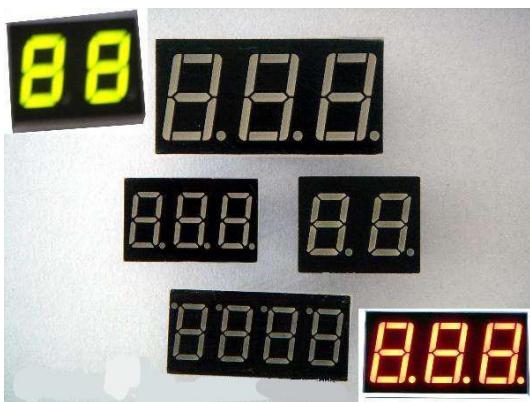


图 4.1.1 各种各样的数码管



图 4.1.2 实验板上的数码管

上图是市面上各种各样的数码管和我们实验板上使用的数码管。其实大家对数码管一点都不陌生，因为它里面都是由一盏一盏的 LED 组成的。数码管方便使用，是单片机技术中不可或缺的显示模块。

1.1 硬件准备

虽然和流水灯一样同时操作多盏 LED 灯，但不同的是，这次我们要把 LED 灯组合成特定的形状，这样以来我们就可以显示数字和字母！我们实验板上的 4 位数码管中有 3 位是 8 段数码管，1 位是米字数码管。米字数码管在 8 段数码管的基础上把中间部分改成 8 段 LED，进而可以显示更多字母。我们要将数码管上的 8 段或者 14 段 LED 组合成为自己想要的图形，那肯定先要给他们排个号、编个顺序。我们就看看 8 段数码管和米字数码管中每段 LED 对应的字母：

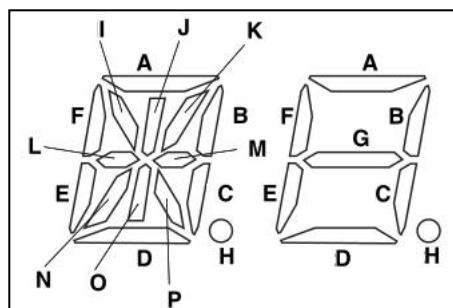


图 4.1.3 8 段数码管和米字数码管编号

然后我们在看看我们打开光盘中的实验板原理图，找到实验板上的 4 位数码管的引脚图：

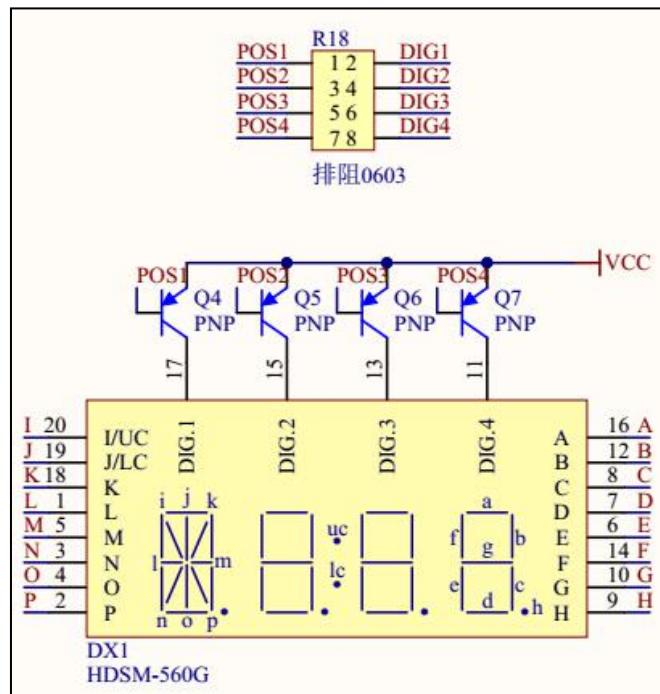


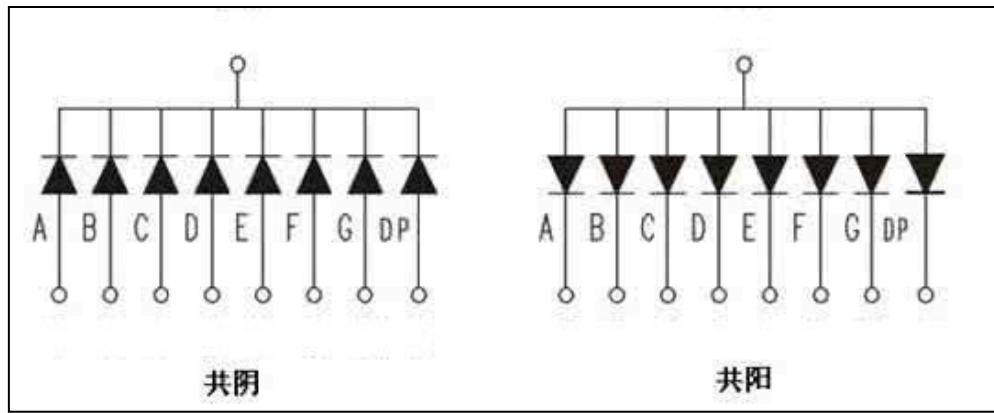
图 4.1.4 实验板数码管电路连接

我们用的数码管一共有 20 个引脚。其中第 11、13、15 和 17 引脚各接一个 PNP 三极管的集电极。四个三极管的发射极同时接 VCC，基极分别接单片机 P4.4、P4.5、P4.1 和 P4.0。这里三极管的作用和蜂鸣器接的三极管作用一样是充当开关作用。数码管第 16、12、8、7、6、14、10 和 9 引脚分别对应 A 到 H 段 LED，第 20、19、18、1、5、3、4 和 2 引脚分别对应 I 到 P 段 LED。

数码管引脚编号和芯片引脚编号大同小异。我们正对数码管（正对时米字数码管位于左侧），左下角第一个引脚位为 1 引脚，逆时针一圈数过去依次为 1 到 20 引脚。

1.2 知识点讲解

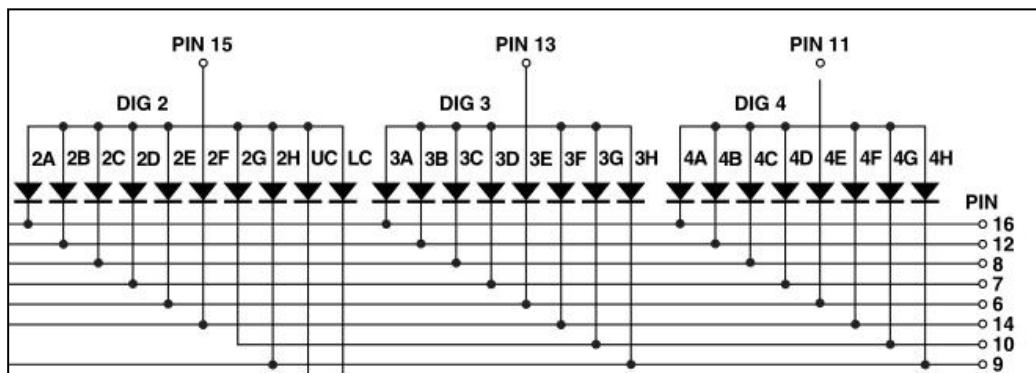
当你拿到一个数码管，你第一时间要了解他的内部结构。数码管根据内部 LED 的电路连接情况，分为共阳极数码管和共阴极数码管。内部 LED 的正极（阳极）共 VCC 的数码管叫共阳极数码管，内 LED 的负极（阴极）共地的数码管叫共阴极数码管。共阳极数码管和共阴极数码管的内部结构如下图（注：DP 段也就是 H 段）：



4.1.5 共阴极和共阳极数码管简图

我们实验板使用的是共阳极数码管，所以我们重点说一下共阳极数码管。共阳极数码管中 LED 的正极均接 VCC。如果我们要点亮某盏 LED，先给共阳段一个适当的电压，然后将该 LED 的负极拉低。所以对于共阳极数码管，LED 的负极作为控制端。其实这和我们的线性 LED 点阵控制 8 盏 LED 没什么区别，只是数码管将 8 盏按一定相对位置摆放，让它能显示各种数字和字母。其实共阴极数码管和共阳极数码管原理是一样的。只是共阴极数码管内 LED 的负极共地，控制端为各个 LED 的正极——你想点亮某盏 LED 你就将该 LED 正极拉低。因此，对于共阴极和共阳极 8 段数码管，如果你想显示相同的数字或字母，那两者给控制端的数据互为反码。

我们实验板上的数码管有位米字数码管和 3 位 8 段数码管，共由 20 只引脚控制。但这和其他单纯的 4 位 8 端数码管相比没多少实质上的区别。其中 8 位数码管内部 LED 连接情况如下图：



4.1.6 8 段数码管内 LED 电路连接

3 位共阳极 8 段数码管的 3 个公共端分别为第 15、13 和 11 引脚。这三个引脚分别接 PNP 三极管的集电极。因为三极管的发射极都接 VCC，所以当三极管的基极拉低

时，电流就从三极管发射极流向集电极。只有这样，我们拉低三极管的基极时，电流才能流进数码管，LED 才能被点亮。可见第 15、13 和 11 引脚控制着从左到右 3 位数码管的开启与关闭，所以我们称他们为位选。另外左边的 8 段数码管由两盏 LED：UC 和 LC，它们分别是数码管上中间分号的两个点。在下一节中我们才用到它，这里我们可以暂时无视它。

至于数码管内 LED 的控制，我们先从 8 段数码管说起。数码管第 16、12、8、7、6、14、10 和 9 引脚分别对应控制 A 到 H 段 LED。如果我们想点亮相应的 LED，只要把相应的引脚拉低即可。因为这些引脚都控制着内部各段 LED 的亮灭，所以我们称他们为段选。从原理图我们看到，我们已经将这 8 个引脚按顺序从低到高先接排阻后接到 74HC573 锁存器 1 的输出端。锁存器 1 相应的 8 个输入端接到单片机的 P2 口。

锁存器，是一种能把信号暂存以维持某种电平状态的电子器件。这里我们用到的 74HC573 锁存器是一款 8 位锁存器。当给使能端 OE 置 1 时，锁存器输出端 (Q) 会随输入端 (D) 的变化而变化。换言之，使能端置 1 后锁存器就相当于是“透明”的，你可以忽略锁存器的存在。但当给使能端置 0 时，锁存器输出端会保持在使能端置 0 前一瞬间输入端的状态。使能端置 0 后不管输入端如何变化，输出端都保持不变。

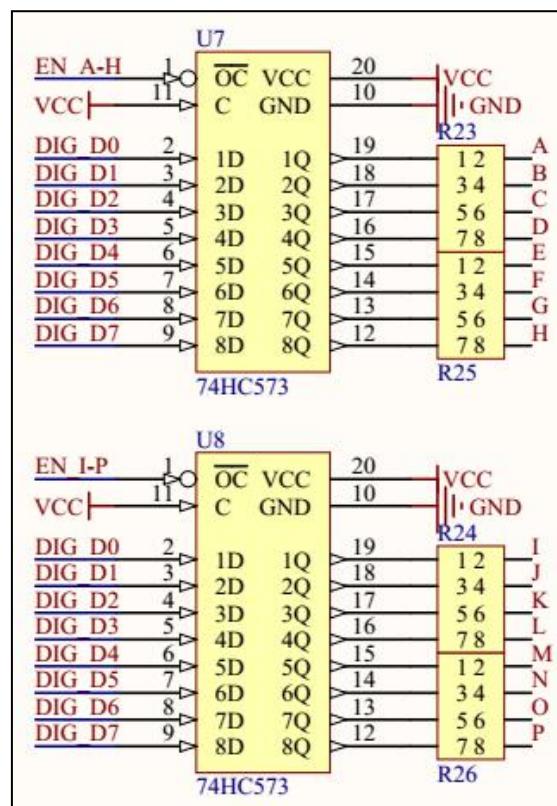


图 4.1.7 数码管锁存器

其实真正用到锁存器是下一节。为什么要用到锁存器？为什么两个锁存器的输入端都一样？这里只做个预告。因为我们可以不用锁存器，所以我们将锁存器使能端常置1，让锁存器保持透明。

因为锁存器是透明的，我们通过单片机P2口控制8段LED的亮灭就可以在数码管上显示你想要的数字或字母。给P2口送的数据和数码管显示的图案存在一一对应关系，我们称这个数据为段选码。为了方便大家，我已经将显示十六进制0~F所需的段选码列成表格，方便大家查找：

字符	段选码
0	0XC0
1	0XF9
2	0XA4
3	0XB0
4	0X99
5	0X92
6	0X82
7	0XF8
8	0X80
9	0X90
A	0X88
B	0X83
C	0XC6
D	0XA1
E	0X86
F	0X8E

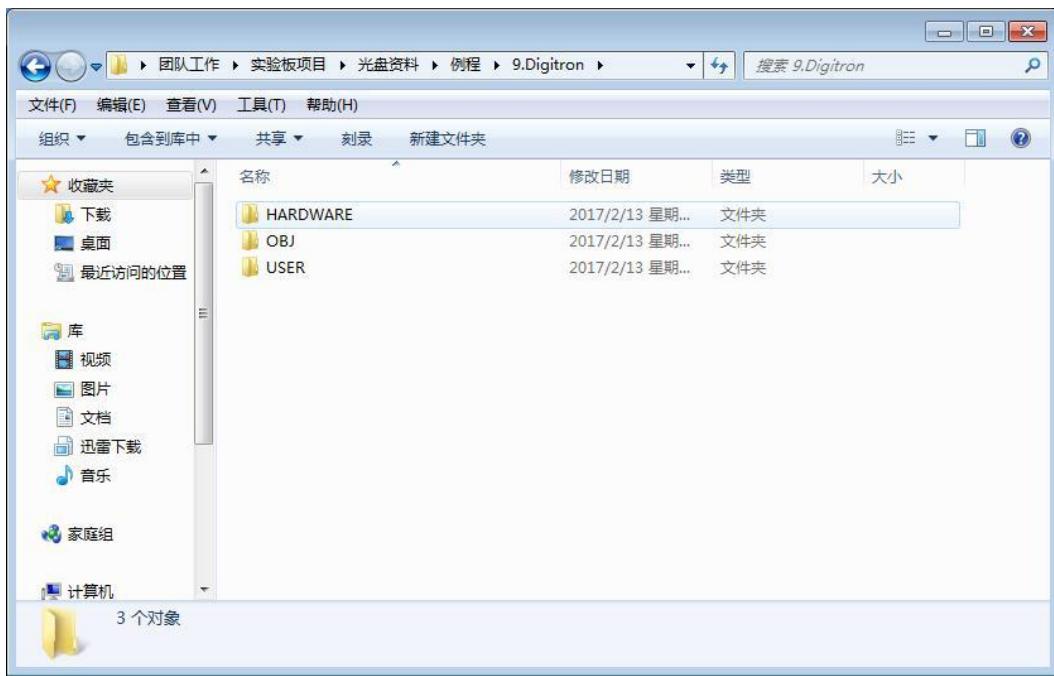
表 4.1.1 8 段数码管 16 进制段选码

上面我们说过，我们实验板采用的是共阳极数码管。所以如果使用共阴极数码管，则以上字符的段选码要变成反码。

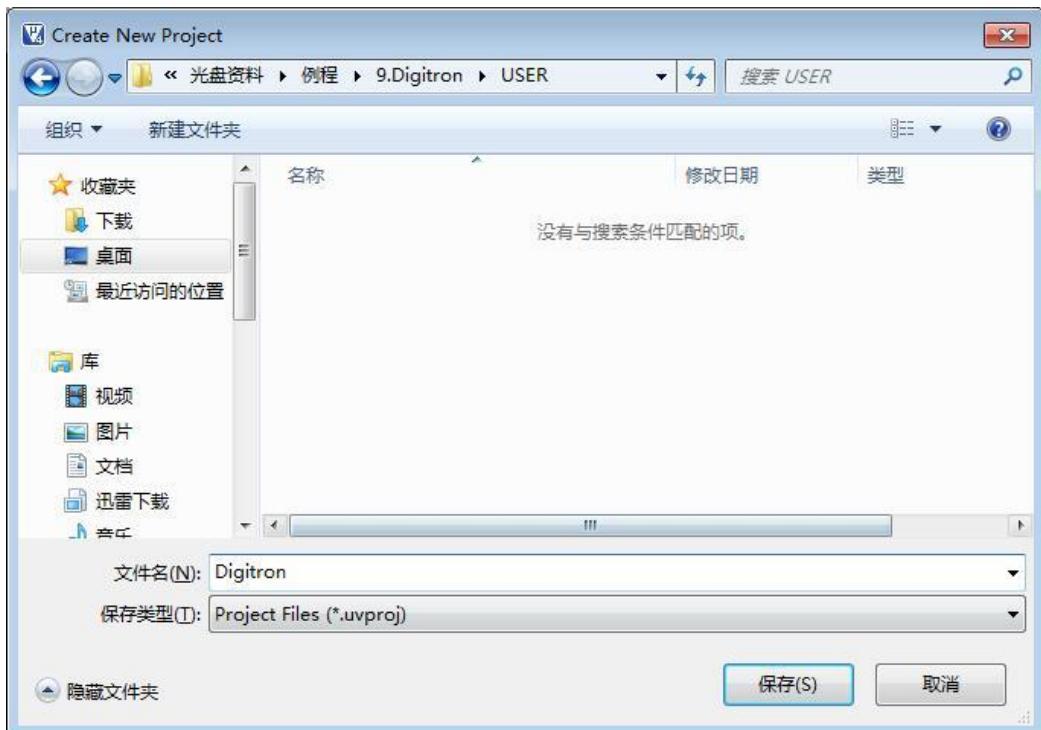
1.3 软件实现

对于以前的工程，因为实在太简单了，在工程建立的时候我们都是很随便。但到了应用篇，我们往往会调用很多单片机的功能和很多外设。因此需要在工程中加以规划，否则所有函数都放在一个C文件中，那就非常杂乱无章。所以我们这里重新叫大家建立一个更科学的单片机工程。

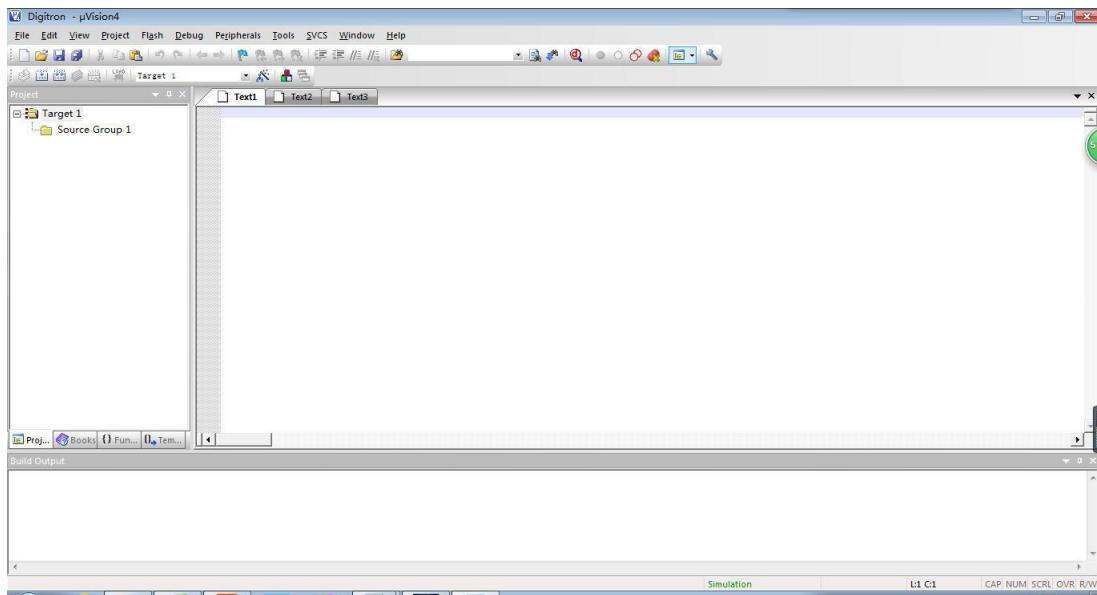
首先我们先建立一个文件夹，文件名字与工程相关，我们就叫数码管（静态显示）。然后在该文件夹下建立两个子文件，分别命名为USER和HARDWARE。其中USER用来存放工程相关文件，HARDWARE用来存放相关头文件。



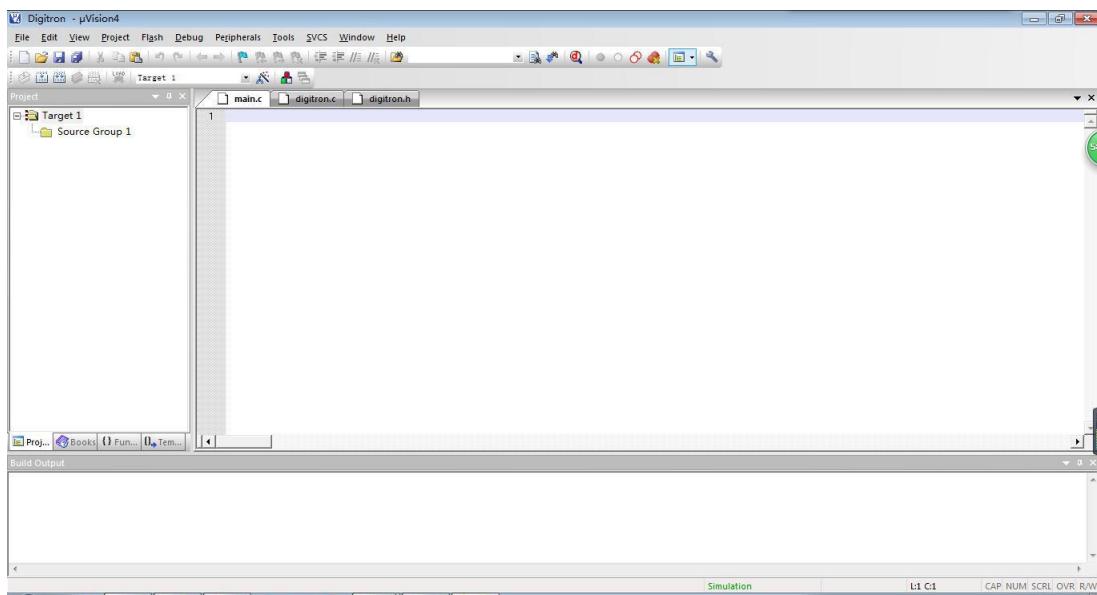
然后我们打开 Keil4，建立工程的方法和我们之前讲的一样，不过我们的工程要放在 USER 中，我们取名为 Digitron。



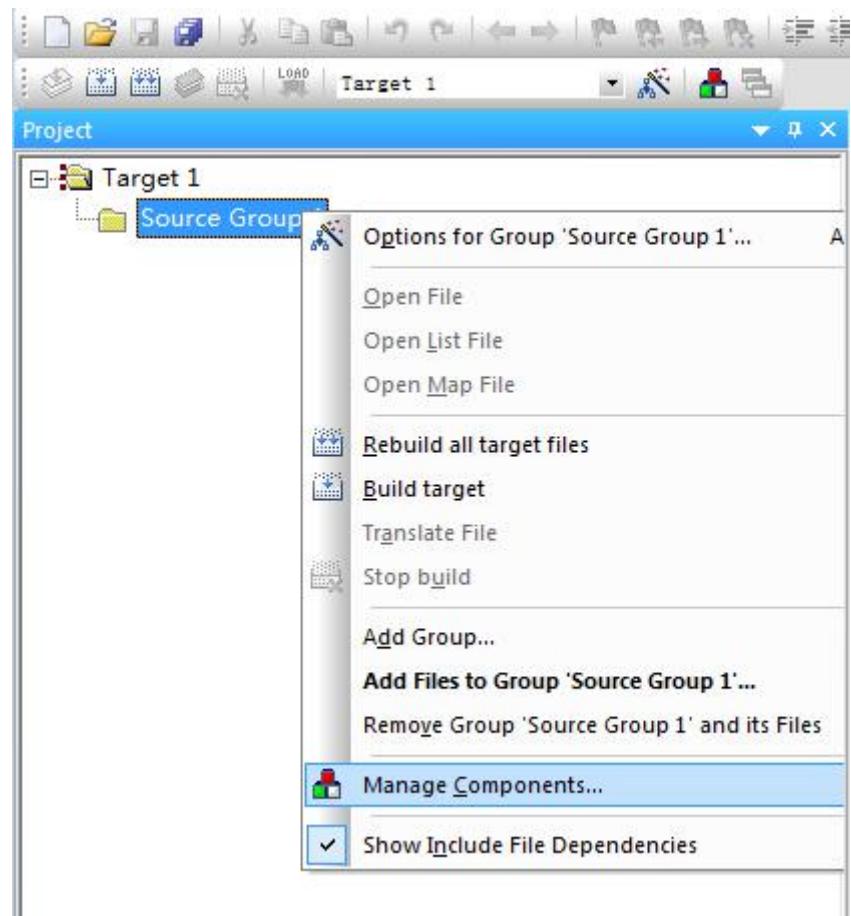
然后我们就如常建立工程。建立好工程后，我们新建三个文件。



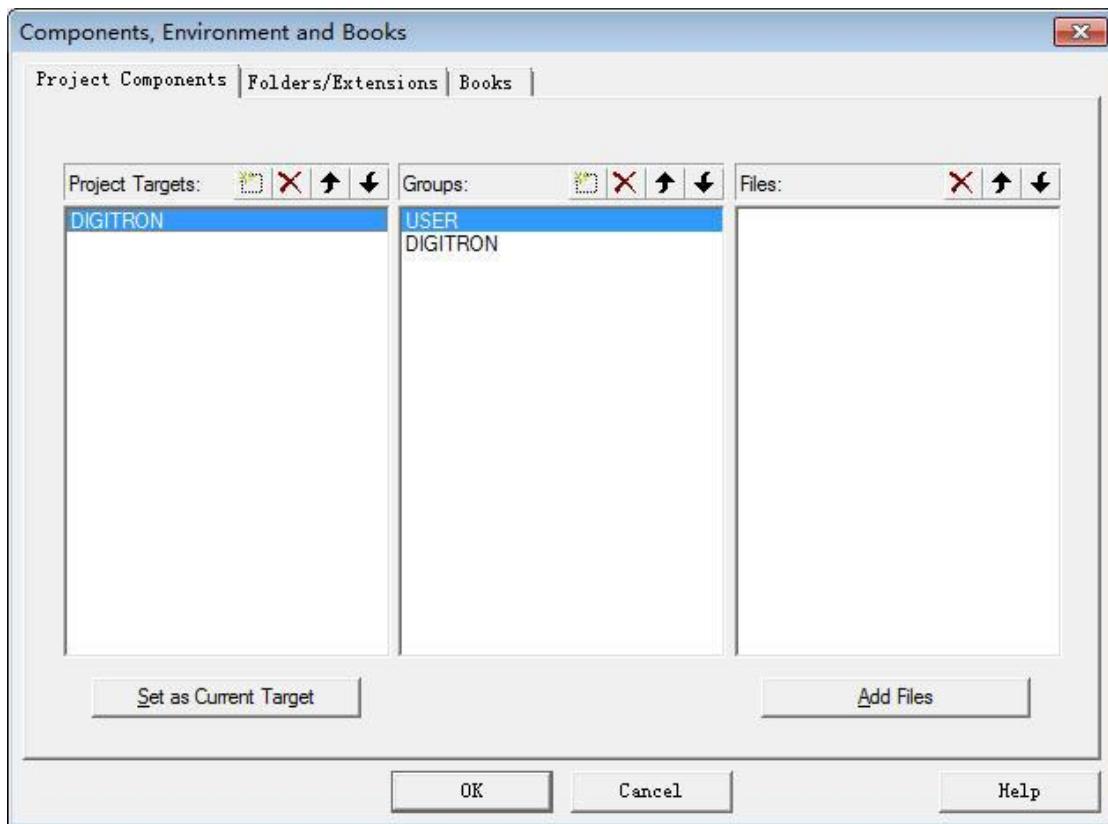
我们把第 1 个文件保存到 USER 中，命名为 main.c；第二个文件保存时，我们在 HARDWARE 文件夹下右键新建 1 个新的文件夹，名为 DIGITRON，把第二个文件保存其中，命名为 digitron.c；第三个文件同样保存在 HARDWARE 的 DIGITRON 下，命名为 digitron.h。注意文件后缀！结果如下图。



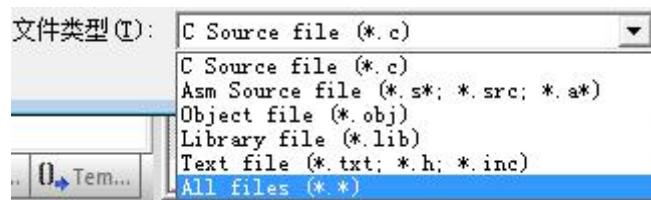
然后我们就要分组。右键 Sourc Group 1，选择 Manage Components。

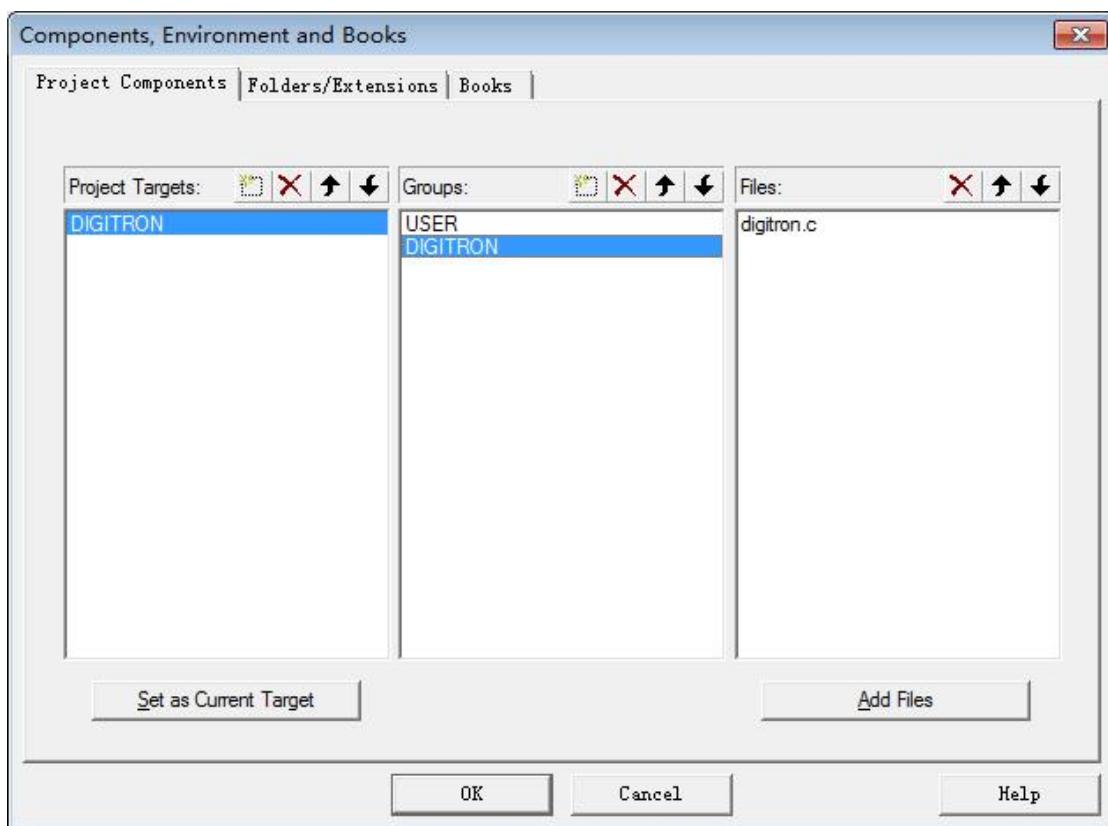
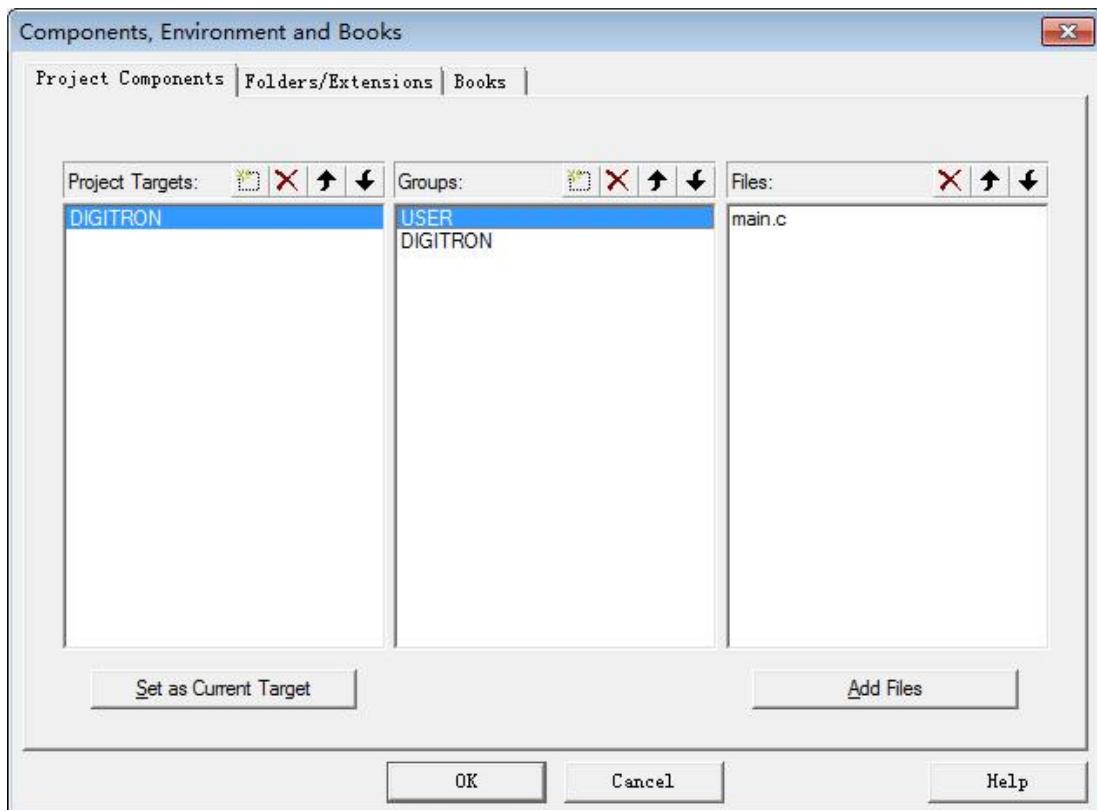


组管理界面中，我们把左侧中的 Target 删去，新建 1 个组，名字与工程名相同，我们就叫 DIGITRON。然后把中间那栏中的 Source Group 删去，新建两个组，分别叫 USER 和 DIGITRON。

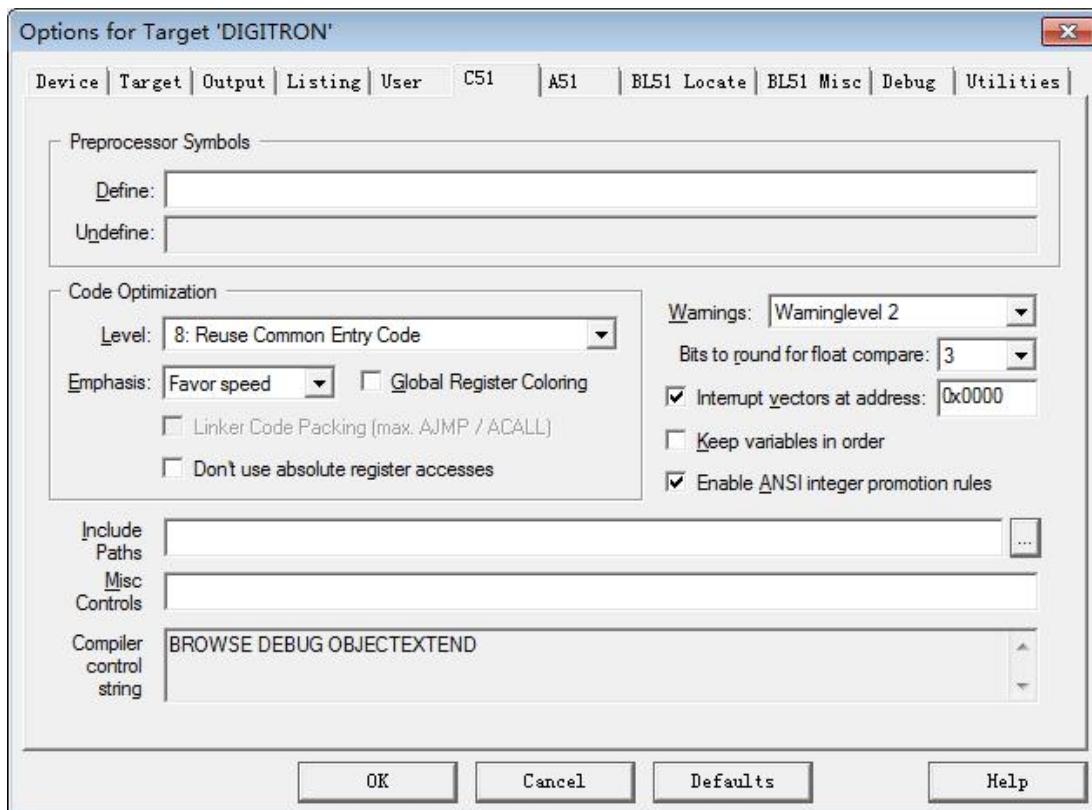


然后选中中间栏中的 USER , 点击右下方的 Add Files。找到 USER 下的 main.c , 添加进去。接着我们选中中间栏中的 DIGITRON , 把 HARDWARE 中 DIGITRON 下的 digitron.c 添加进去。

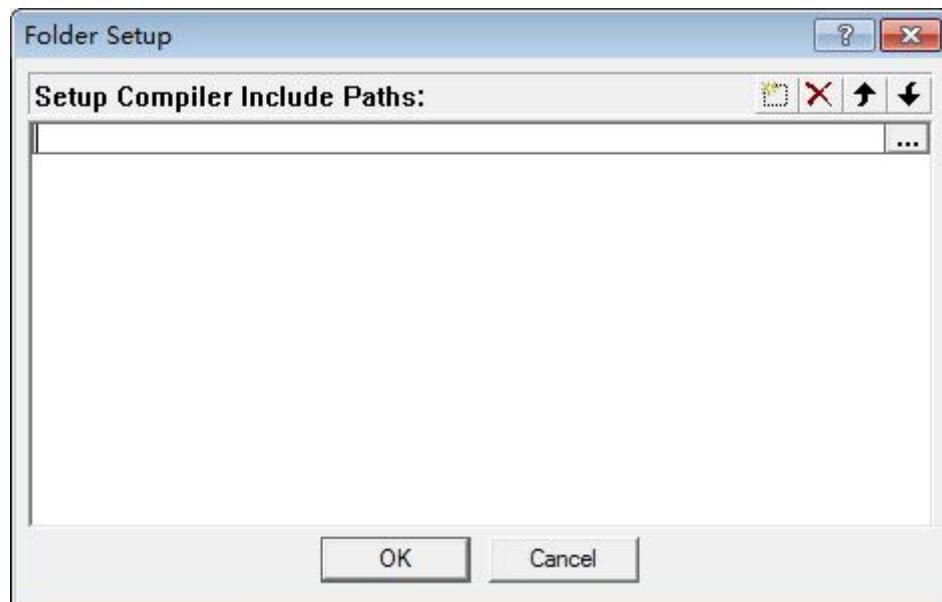




因为我们用到 1 个自己建立的头文件 digitron.h，因此我们要把我们自己建立的头文件的位置告诉给编译器。打开目标选项，选择 C51 一栏。



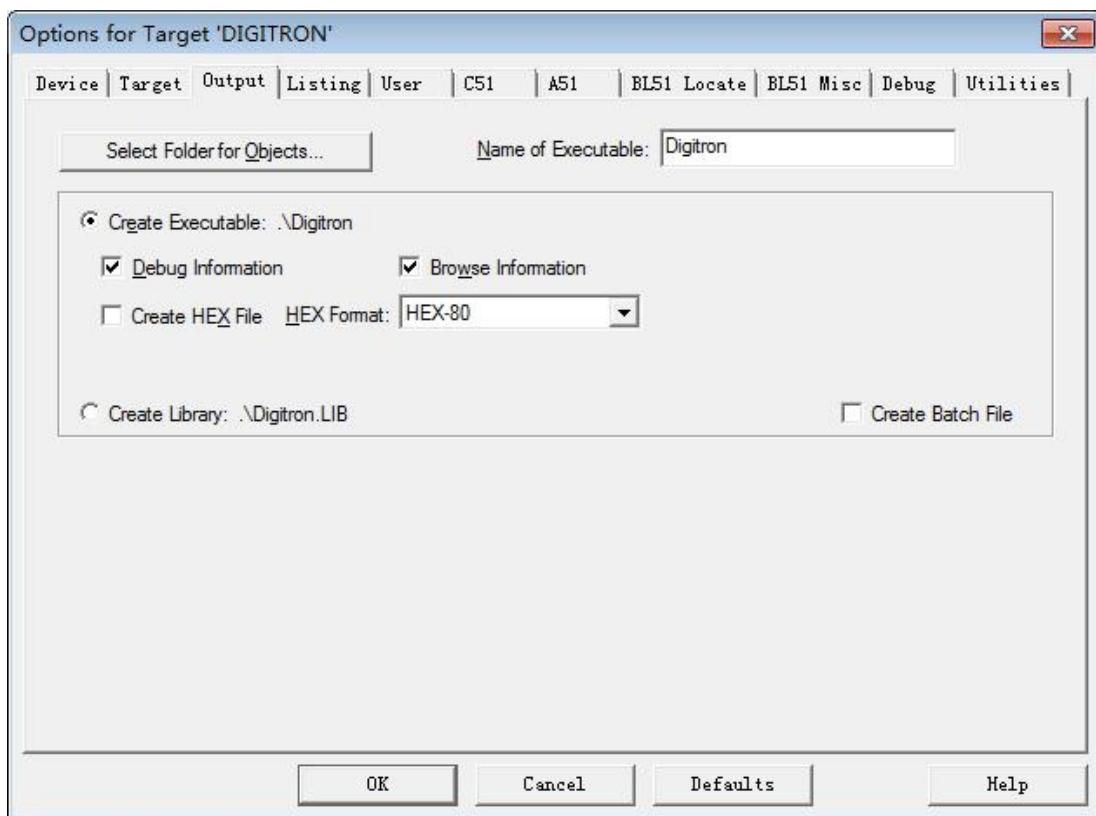
选择 Include Paths 栏右侧带省略号的小框框。按新建按钮，再次选择右侧带省略号的小框框。

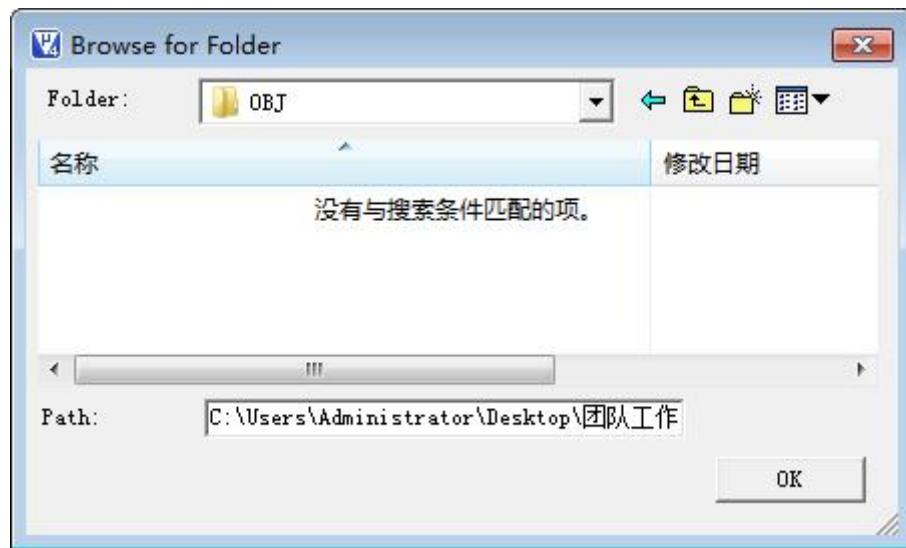


把位置定位在头文件所在的文件夹。



按确定添加完毕后，我们选择上方 Output 栏。在界面中选择 Select Folder for Objects。这一步我们是选择生成 hex 文件的位置，我们选择我们之前建立的文件夹 BOJ。





当然不要忘了勾选生成 hex 文件。到这里，一个新的工程就建立完成了。虽然比起以前的方法，这种方法显得很繁琐，却能给工程建立一个清晰明了的框架。我们会在以后的编程中慢慢体会。

工程建立好后，我们就开始编程了。

在 digitron.c 中写入下列程序：

```
#include "stc12.h"
#include "digitron.h"

/*
-----  
编号-显示 编号-显示 编号-显示 编号-显示  
0 - 0      4 - 4   8 - 8   12 - C  
1 - 1      5 - 5   9 - 9   13 - D  
2 - 2      6 - 6   10 - A   14 - E  
3 - 3      7 - 7   11 - B   15 - F
-----*/  
  
uchar code table[] =  
{  
0XC0,0XF9,0XA4,0XB0,  
0X99,0X92,0X82,0XF8,  
0X80,0X90,0X88,0X83,  
0XC6,0XA1,0X86,0X8E  
};  
  
void Digi_Init(void)  
{  
    P4SW = 0X30;  
}  
  
//数码管静态显示函数
```

```

//前三个型参为从左到右 3 个八段数码管 3 个位选，1 有效
//最后一个型参为显示的数字或字母，取值范围为 0~15
void Digi_Show(uchar a, uchar b, uchar c, uchar num)
{
    DIG1 = 1;
    DIG2 = !a;
    DIG3 = !b;
    DIG4 = !c;

    EN1 = 1;
    P2 = table[num];
    EN1 = 0;
}

//数码管静态显示演示函数
void Digi_Test(void)
{
    uint i;

    for(i = 0; i < 16; i++)
    {
        Digi_Show(1, 1, 1, i);

        Delay_ms(1000);
    }
}

//毫秒延时函数，误差 2us
void Delay_ms(uint z)
{
    uint x, y;

    for(x = z; x > 0; x--)
        for(y = 921; y > 0; y--);
}

```

接着在 digitron.h 中写入下列程序：

```

#ifndef _DIGITRON_H_
#define _DIGITRON_H_

#define uint unsigned int
#define uchar unsigned char

sbit DIG1 = P4^0;//4 个位选
sbit DIG2 = P4^1;
sbit DIG3 = P4^4;
sbit DIG4 = P4^5;
sbit EN1 = P0^5;//锁存器使能端

```

```
void Digi_Init(void);
void Digi_Show(uchar a, uchar b, uchar c, uchar num);
void Digi_Test(void);
void Delay_ms(uint z);

#endif
```

最后在 main.c 中写入下列程序：

```
#include "stc12.h"
#include "digitron.h"

void main()
{
    Digi_Init();

    while(1)
    {
        Digi_Test();
    }
}
```

我们不再像入门篇和进阶篇一样把所有函数都放在 main.c 中，而是把数码管相关的程序放在 digitron.c 中。这样一来，我们就把功能模块化了。需要某种功能我们在其他文件中写好然后在 main.c 中调用。main.c 就像一个机器人的身体，而其他类似于 digitron.c 的功能函数就像机器人的手臂和头部可以自由地和 main.c 组装在一起。在往后一些用到多个功能的编程中我们就能体会到这种模块化使单片机编程就像拼积木一样简单。

在 digitron.c 中，我们将 0~F 的段选码放在一个数组中。在声明数组时我们用到了一个新的关键字 code。使用 code 声明的数组将放在单片机的只读存储器内。

单片机内的存储空间主要由两部分构成：只读存储器（ROM，Read Only Memory）和随机存取存储器（RAM，Random Access Memory）。只读存储器顾名思义是只能读取的，它用来存放我们单片机的程序。如果要改变只读存储器的数据只能通过烧录程序来刷新，在单片机运行程序时或复位后只读存储器内的数据是不会被改变的。而随机存取存储器是用来存放变量和程序执行过程中产生的数据，单片机上电复位后随机存取存储器里面的数据都会被刷新。

将段选码数组放在 ROM 里有三个原因：

①将数组放入 ROM 中可以防止写程序时不小心修改数组。如果修改已经用 code 声明的数组编译器将会报错；

②节省 RAM 空间。stc12 的 RAM 只有 1280 字节而 ROM 有 128K 字节。如果数组加大时，应将数组放进 ROM 中；

③程序调用 ROM 中的数组比调用 RAM 中的更快。

再看看 digitron.h。digitron.h 是用来声明 digitron.c，所以我们要记住 digitron.h 中.h 文件的基本格式。.h 文件中的程序都要写在#ifndef _XXX_H_ #define _XXX_H_ 和#endif 之间。这是用来防止重复声明。只有这样，多个.c 文件才可以重复调用这个.h 文件了。因为我们不知道其它.c 文件里的程序会不会重复调用 digitron.h，如果不声明，则会报错。

然后我们习惯将.h 文件声明的.c 文件中用到的宏定义和特殊功能寄存器位变量放在该.h 文件里。这样将一些琐碎的代码放在.h 文件中使得.c 文件的程序更简——在你要修改一些变量或宏定义时，直接找到头文件，在头文件里面修改就行。然后最重要的是将.c 文件中写的函数在.h 文件中都声明一次，否则编译会报错。

回归程序本身。从左到右四位数码管对应的四个位选，分别为 DIG1、DIG2、DIG3 和 DIG4。它们分别由单片机的 P4.0、P4.1、P4.4 和 P4.5 控制。当置低时，才能点亮数码管。

但 P4.4 和 P4.5 有点特别。我们之前说过，单片机的引脚是有限的，但为了使单片机在引脚有限的情况下能实现更多的功能，某些引脚会有复用功能。刚刚说 P4.4 和 P4.5 有点特别，说的正是它们有复用功能。而控制它们到底用于什么功能，就要看寄存器 P4SW 如何设置了：

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义		LVD_P 4.6	ALE_P4. .5	NA_P4. 4				

表.. P4SW 相关位定义

ALE_P4.5：置 0，P4.5 为 ALE 信号脚，只有在用 MOVX 指令访问片外扩展器件时才有信号输出；

置 1，P4.5 为普通 IO 口。

NA_P4.4：置 0，P4.4 为外部低压检测脚，可使用查询方式或设置成中断来检测；

置 1，P4.4 为普通 IO 口。

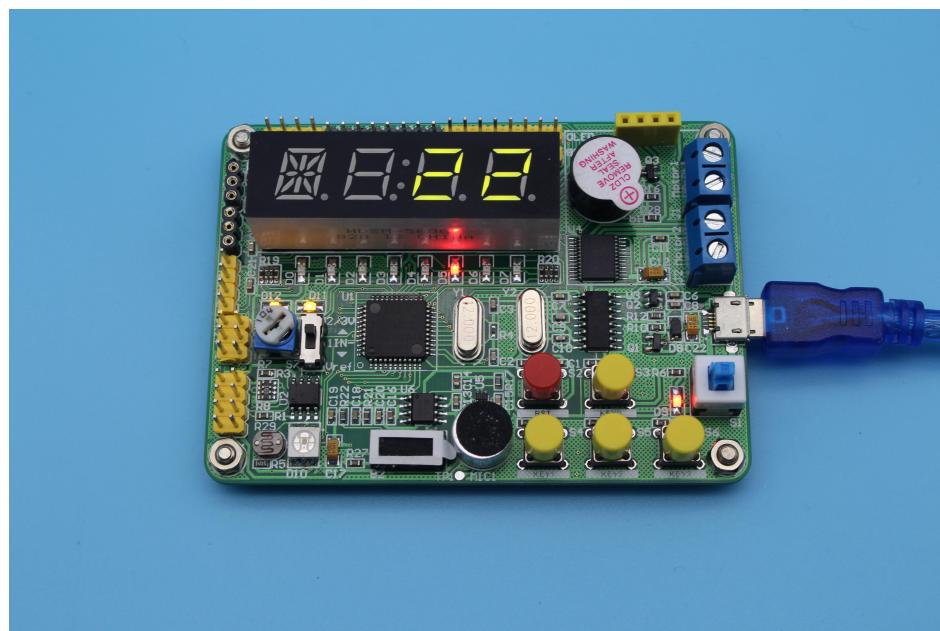
P4.4 和 P4.5 的复用功能我们不用管能干什么，我们只需将它们用作普通 IO 口，可以置高置低就行。所以我们要将这两位置 1，给 P4SW 赋 0X30 作为数码管的初始化。

数码管初始化完成后，我们就可以操作数码管了。操作数码管第一步是设置位选。我们把目光移到函数 Digi_Show()。Digi_Show()前 3 个型参 a、b、c 都是控制从左到右 3 个 8 段数码管位选的。将 a、b、c 赋给 DIG2、DIG3、DIG4，就能控制数码管的位选。我们知道，当 DIGx 置 0 为打开位选，置 1 为关闭位选。但这和我们日常的 1 为有，0 为没有相冲突。为了方便程序的阅读和使用，在将 a、b、c 赋给 DIG2、DIG3、DIG4 前，我们先将 a、b、c 取反。这样一来，我们使用 Digi_Show() 函数设置位选时，置 1 为打开位选，0 为关闭位选。

在设置好位选后，我们就要设置段选来显示数字或字母了。送段选码时，我们先将锁存器使能端置 1，使锁存器输出和输入保持一致。当我们送完段选码后，我们再失能锁存器，让锁存器输出端保持不变，即保持数码管上的显示不变。这样一来，如果你将控制段选的 P2 用于其他用途，也不会影响数码管的显示！

1.4 下载验证

把工程编译生成的 hex 文件烧录进单片机中。观察数码管的显示情况。



修改 Digi_Show() 函数的三个型参，观察数码管的显示变化。

1.5 思考总结

在使用数码管前，千万千万不要记住先将 PS4W 寄存器的第 4 和第 5 位置高，将 P4.4 和 P4.5 设置成普通 IO 口模式。否则数码管后两位将无法使用。

通过标题我们知道，我们将这种控制数码管的方式称为静态显示。静态显示的弊端是很明显的：3 位数码管它只能同时显示相同的数字或字母。3 位数码管同时只能显示相同的图案，那它和 1 位数码管没有实质上的区别。数码管想要实用，至少可以我们可以控制它显示一个 4 位的任意数字。这就要用到我们下一节要讲的动态显示了。

还有你有没有发现，因为三极管的存在，数码管位选 1 是熄灭，0 是点亮。这有悖于我们平时的思维。所以我们写的 Digi_Show() 函数接受控制位选的型参，先取反再送给位选端。这样一来，我们运用 Digi_Show() 函数控制位选时就变成 1 点亮，0 熄灭。这只是一个小小的修改，却给使用函数的人带来不一样的使用体验！

二、数码管动态显示——LED 变身白富美

2.1 知识点讲解

在上一节中我们用静态显示操作数码管。但用静态显示的话，3位数码管是显示相同的字母或数字，根本不实用。为此，我们在这节来看看如何通过动态显示使4位数码管显示四个不同的数字。

在静态显示中，我们只点亮某几位数码管，而动态显示却没那么简单。动态显示顾名思义，我们要显示的数码管是动态改变的。假如我们要在4位数码管上分别显示1、2、3和4：首先我们用位选先点只亮最左边一位数码管，让它显示我们想要的1，然后熄灭第一位数码管然后只点亮第二位数码管并让它显示2，以此类推第三第四位数码管并不断重复。

如果点亮某位数码管和熄灭上一位数码管之间的时间间隔较长时，我们能观察到数码管是从左到右按顺序依次显示1、2、3和4。如果我们将间隔缩短，让数码管之间的切换速度快到到人的肉眼分辨不出数码管之间的切换，那4位数码管看起来就分别同时显示1、2、3和4了。

在这节动态显示中，我们会用到米字数码管。相对于8端数码管，米字数码管没有8端数码管中的G号LED，另外又新增了8段LED。我们再回顾一下实验板上的数码管的内部LED编号情况和连接情况：

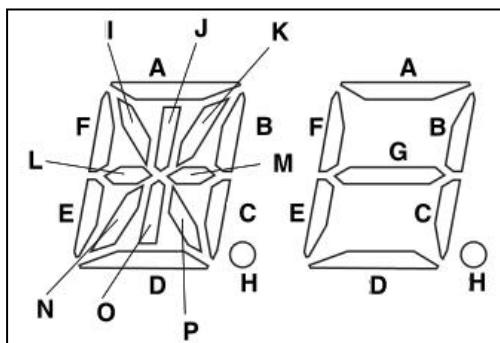


图 4.2.1 8段数码管和米字数码管编号

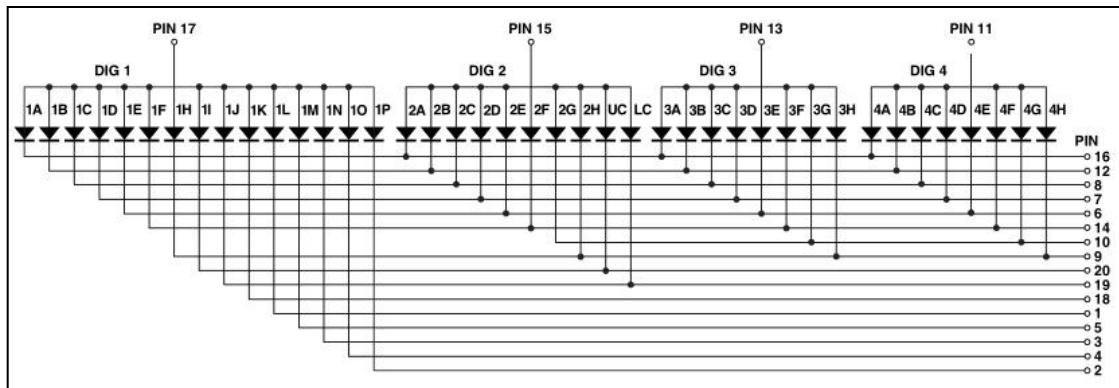


图 4.2.2 数码管内部 LED 连接情况

我们将新增的 LED 编号为 I~P，像 A~H 数码管一样按顺序从低到高先接排阻后再接到锁存器 2 的输出端。这样我们就可以通过控制锁存器 2 像控制原 8 段数码管一样控制这新增的 8 段 LED。

控制米字数码我们可以用两组 I/O 口分别直接控制米字数码管外面 7 盏 LED 和里面米字的 8 盏 LED，为此我们用到了两个锁存器：锁存器 1 控制米字数码管外面 7 盏 LED，锁存器 2 控制里面米字的 8 盏 LED。这是为什么呢？你有没有发现，两个锁存器的输入端都是 P2 口。我们利用锁存器能锁存输出信号的特性，先开锁存器 1，发送控制外围 LED 的段选码，关闭锁存器 1。然后打开锁存器 2，发送控制内部 LED 的段选码，再关闭锁存器。我们分别通过开启和关闭锁存器 1 和锁存器 2 的方法，就能只用一组 I/O 口就能够控制米字数码管了。这样一来就能充分利用到单片机的引脚资源。

2.2 软件实现

建立工程，命名为“数码管-动态显示”。在 HARDWARE 文件中新建文件夹 DIGITRON，文件下新建 digitron.c 和 digitron.h。在工程所在的 USER 中新建文件 main.c。

在 digitron.c 中写入以下代码：

```
#include "stc12.h"
#include "digitron.h"

/*
-----  

编号-显示  编号-显示  编号-显示  

0 - 0      12 - C    24 - O  

1 - 1      13 - D    25 - P  

2 - 2      14 - E    26 - Q  

3 - 3      15 - F    27 - R
```

```

4 - 4      16 - G    28 - S
5 - 5      17 - H    29 - T
6 - 6      18 - I    30 - U
7 - 7      19 - J    31 - V
8 - 8      20 - K    32 - W
9 - 9      21 - L    33 - X
10 - A     22 - M    34 - Y
11 - B     23 - N    35 - Z
-----*/
//8 段数码管段选码
uchar code table[] =
{
0XC0,0XF9,0XA4,0XB0,
0X99,0X92,0X82,0XF8,
0X80,0X90,0X88,0X83,
0XC6,0XA1,0X86,0X8E
};

//米字数码管段选 1
uchar code table1[] =
{
0XC0,0XF9,0XE4,0X39,0XD9,0XD2,
0X3D,0X07,0XC0,0XD0,0XC8,0XC3,
0XC6,0XE1,0XC6,0XCE,0XC2,0XC9,
0XF6,0FFF,0FFF,0XC7,0XC9,0XC9,
0XC0,0XCC,0XC0,0XCC,0XD2,0XFE,
0XC1,0FFF,0XC9,0FFF,0FFF,0XF6
};

//米字数码管段选 2
uchar code table2[] =
{
0FFF,0FFF,0XE7,0XE7,0XE7,0XE7,
0XE7,0FFF,0XE7,0XE7,0XE7,0XE7,
0FFF,0XE7,0XE7,0XE7,0XDF,0XE7,
0XBD,0FFF,0X39,0FFF,0XF9,0X7E,
0FFF,0XE7,0XF,0X67,0XE7,0XBD,
0FFF,0FFF,0X5F,0X5C,0XBA,0XDB
};

void Digi_Init(void)
{
    P4SW = 0X30;
}

//数码管动态按位显示函数
//a、b、c、d 分别为从左到右 4 个数码管的 4 个位选，1 有效

```

```
//num1、num2、num3、num4 分别为四个数码管的段选，其中 num1 取值范围为  
0~35 , num2、num3、num4 取值范围分别为 0~15  
void Digi_Show(uchar a,uchar num1,uchar b,uchar num2,uchar c,uchar num3,uchar  
d,uchar num4)  
{  
    DIG1 = !a;  
    DIG2 = 1;  
    DIG3 = 1;  
    DIG4 = 1;  
  
    EN1 = 1;  
    P2 = table1[num1];  
    EN1 = 0;  
  
    Delay_ms(2);  
  
    EN2 = 1;  
    P2 = table2[num1];  
    EN2 = 0;  
  
    Delay_ms(2);  
  
    DIG1 = 1;  
    DIG2 = !b;  
    DIG3 = 1;  
    DIG4 = 1;  
  
    EN1 = 1;  
    P2 = table[num2];  
    EN1 = 0;  
  
    Delay_ms(2);  
  
    DIG1 = 1;  
    DIG2 = 1;  
    DIG3 = !c;  
    DIG4 = 1;  
  
    EN1 = 1;  
    P2 = table[num3];  
    EN1 = 0;  
  
    Delay_ms(2);  
  
    DIG1 = 1;  
    DIG2 = 1;  
    DIG3 = 1;  
    DIG4 = !d;  
  
    EN1 = 1;  
    P2 = table[num4];
```

```

    EN1 = 0;
    Delay_ms(2);
}

//数码管整数显示函数
//给函数一个变量，能在数码管上显示出来
//变量取值范围：0~9999
void Digi_Num(uint num)
{
    uchar a,b,c,d;//从 num 分离出来的千位、百位、十位和个位

    if((num > 0) || (num < 9999))
    {
        a = num / 1000;
        b = num % 1000 / 100;
        c = num % 1000 % 100 / 10;
        d = num % 1000 % 100 % 10;

        Digi_Show(1,a,1,b,1,c,1,d);
    }
    else
        Digi_Show(1,9,1,9,1,9,1,9);
}

//毫秒延时函数，误差 2us
void Delay_ms(uint z)
{
    uint x,y;

    for(x = z;x > 0;x--)
        for(y = 921;y > 0;y--);
}

```

在 digitron.h 中写入以下代码：

```

#ifndef _DIGITRON_H_
#define _DIGITRON_H_

#define uint unsigned int
#define uchar unsigned char

sbit DIG1 = P4^0;//4 个位选
sbit DIG2 = P4^1;
sbit DIG3 = P4^4;
sbit DIG4 = P4^5;
sbit EN1 = P0^5;//两个锁存器使能端
sbit EN2 = P0^4;

```

```
void Digi_Init(void);
void Digi_Show(uchar a, uchar num1, uchar b, uchar num2, uchar c, uchar num3, uchar d, uchar num4);
void Digi_Num(uint num);
void Delay_ms(uint z);

#endif
```

在 main.c 中写入以下代码

```
#include "stc12.h"
#include "digitron.h"

void main()
{
    Digi_Init();

    while(1)
    {
        Digi_Num(1234);
    }
}
```

在 digitron.c 中，我们通过动态扫描写了一个能单独控制各个位显示的 Digi_Show() 函数。

扫描法，我们是通过控制位选来实现的。在发送各个段选码前，我们依次将各个位选单独置 0。这样数码管每个位就单独显示我们要显示的数字。在每个位显示的间隔，我们都还有一个延时函数。这个延时函数是用来控制扫描每个位的间隔的。如果时间间隔较大，我们肉眼就能分辨每个位之间的跳跃。所以我们要控制扫描的间隔足够的小，实测 1 到 2 微妙效果最佳。

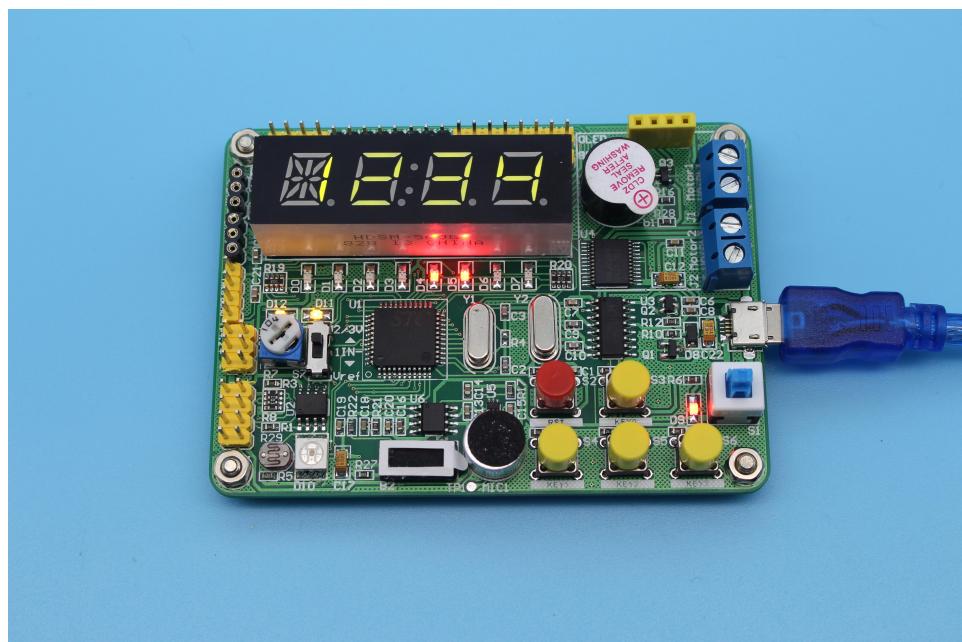
对于米字数码管，段选比 8 段数码管复杂一点点。因为 8 段数码管内部只有 8 盏 LED，一个数组就可以胜任。而米字数码管内共有 15 盏 LED，外围 LED 和内部 LED 是分开控制的。为了方便操作，我们声明了两个数组，分别用来存放外围 LED 和内部 LED 的段选码。

实际应用中，我们经常将数码管作为一种调试的手段：将程序运行中一些变量显示在数码管上，借此来观察程序的运行情况。举个例子：如果我们要通过一个模块来测量环境的温度，处理后的数据我们就需要直接实时显示在数码管上。如果我们用 Digi_Show() 函数来显示温度，那每次显示前我们都要把温度的千、百、十和个位通过数学方法分离出来，再赋给 Digi_Show() 函数。

为了简化程序，我们就写了一个 Digi_Num() 函数。这个函数接受一个 uint 型变量 num，它能先将 num 分离出千位、百位、十位和个位，再用 Digi_Show() 函数显示出来。这样封装的目的是方便我们使用数码管，不用每次都要分离再显示。另外因为数码管是 4 位，而 uint 变量是 16 位二进制变量最大值位 65536，所以我们再分离前先判断它是否超过 9999。如果没超过则正常显示，否则显示 4 个 9。

2.3 下载验证

把工程编译生成的 hex 文件烧录进单片机中。观察数码管的显示情况。



2.4 思考总结

点亮熄灭数码管之间的时间间隔很有讲究。除了要足够短之外，每次延时都要保证间隔相等。试想一下，如果前三位数码管点亮后 2us 后熄灭，而偏偏第四位在点亮后 10us 后才熄灭了。这样子循环下来，在一个显示周期中，第四位数码管显示的时间就比前三位长，导致前三位数码管比第四位数码管更暗。

用数码管扫描显示，我们就能把一些数据显示出来，方便我们计量。但是，数码管显示有个很大的限制——你要不断地去扫描显示。如果在扫描中

三、角度舵机——机电一体化大门 1

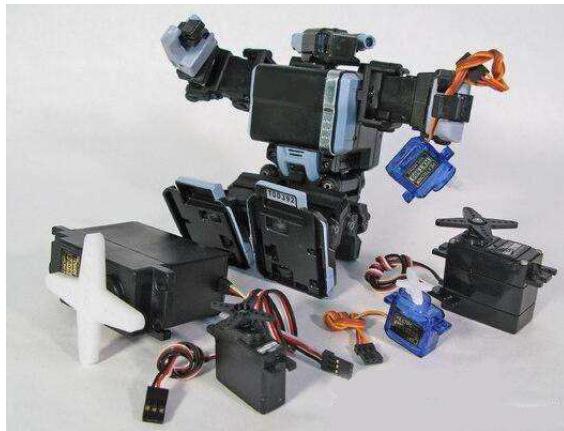


图 4.3.1 机器人及各种舵机



图 4.3.2 小电机

大家小时候接触的机器人玩具动起来时关节是不是总是滋滋滋的响。为什么机器人的手臂能转动？而且转动起来总是滋滋声地响？又例如小时候经常玩的四驱车，大家肯定见过它的“心脏”——小直流电机，也就是我们平时所说的马达。我们玩四驱车都是直接打开开关，车子就飞快跑起来。那马达只能控制开关而不能控制它的转速快慢吗？在接下来两这节中，我们就分别讲讲支撑机器人动作和驱动四驱车飞速前进的“大力士”：角度舵机和直流电机，并用 PWM 信号控制角度舵机和直流电机。

3.1 硬件准备

按工作方式划分，我们经常用的舵机有角度舵机和周转舵机两大类。周转舵机给定一个 PWM 信号后只能正转或反转。能通过控制 PWM 占空比来控制转速，但并不能控制它转多少圈；而角度舵机的转角是和 PWM 的占空比是一一对应的，因此角度舵机的转角是可控的。所以对于机器人的控制，角度舵机的实用性远远高于周转舵机。

我们先看看角度舵机的近照和内部结构图：



图 4.3.3 角度舵机内部结构



图 4.3.4 舵机及各种舵盘

角度舵机里有一套由控制电路、马达、电位计、传动轴和传动结构组成的自动控制装置（图 4.3.3 所示）。所谓的自动控制就是用一个闭环反馈控制电路不断修正输出的偏差。因此我们只要给定舵机信号端一个稳定的 PWM 信号时，它就会保持在一定的角度。另外，这个闭环反馈控制电路也限制了角度舵机的最大旋转角度，一般角度舵机最大旋转角范围为 180° 左右。

舵机的传动轴又短又滑，怎能带动结构一起运动呢。这时我们就要靠各种舵盘来做结构上的支撑和增大扭矩。不同的舵盘可以适用不同的场合，完成不同的任务。

3.2 知识点讲解

角度舵机的控制其实很简单，就是用我们多次学到的 PWM 控制。角度舵机由三条线：红线是 5V 电源线，黑线是地线，白线是信号线（有些舵机是：红线是电源线，棕线是地线，橙线是信号线）控制。**注意舵机正负极不能接反，否则舵机会烧掉！** 舵机自带的线是 3pin 头，只要对准实验板上的插座插上去就保证不会正负接反。

角度舵机内部的控制电路只接收周期为 15~20ms 的 PWM 信号，而 PWM 信号的占空比决定了角度舵机的转动角度：

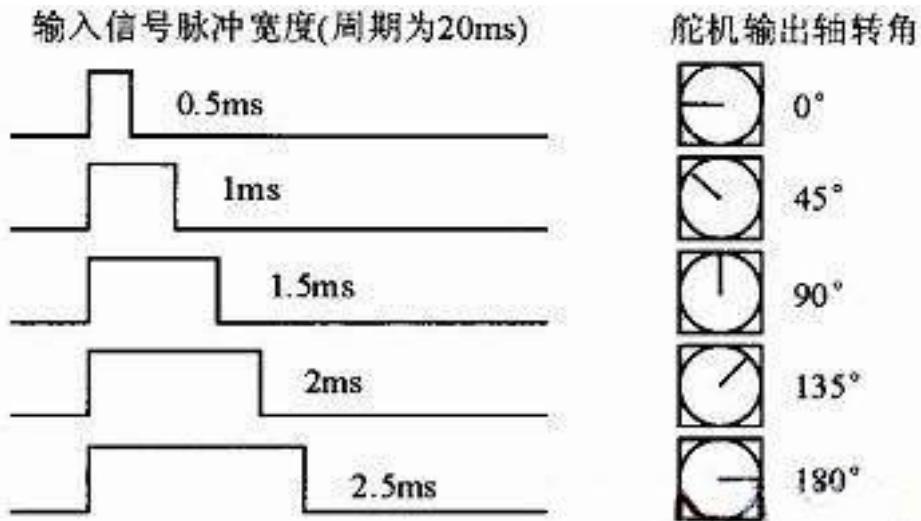


图 4.3.5 信号脉冲宽度和舵机转角的关系

通过图我们可以看到，已占空比为 7.5%（脉宽 1.5ms）为转角中间值。占空比越小，舵机越往左转，但最小值为 2.5%，即最大左转 90°。占空比越大，舵机越往右转，但最大值为 12.5%，即最大右转 90°。当然，这里说的左和右是面朝舵机转动轴时的相对方向。

角度舵机的规格有很多，不同规格的舵机之间在性能上会有一定的区别。选择角度舵机时要参考以下几个参数：

- ① 尺寸、重量和材质：不同的尺寸和重量适用不同的场合。材质主要指舵机内齿轮的材质，一般分塑料齿轮和金属齿轮。金属齿轮的角度舵机比塑料齿轮的舵机更加耐用，但相对价格也贵一点；
- ② 转速：角度舵机的转速是指无负载的情况下转过 60°所需的时间。常见的舵机转速一般在 0.11s~0.21s 之间；
- ③ 扭力：舵机扭力的单位是 Kg·cm，是指舵盘上距离舵机轴中心水平 1cm 处能带动的物体重量。它是用来衡量舵机“力量”大小的量度；
- ④ 电压：不同型号的舵机的工作电压和能承受的最大电压不尽相同。对于同一款舵机，工作在不同电压下它的转速和扭力参数也不相同。因此要仔细查看舵机的使用说明书。

所以，在选择舵机时我们要全面考虑各个因素，选择最适用的舵机。

3.3 软件实现

建立工程，命名为“舵机”。在 HARDWARE 文件中新建文件夹 STEERING，文件下新建 steering.c 和 steering.h。在工程所在的 USER 中新建文件 main.c。

在 steering.c 中写入以下程序：

```
#include "stc12.h"
#include "steering.h"

//舵机初始化函数
void Steering_Init(void)
{
    CCON = 0X00;//初始化 CCON
        //关闭 PCA 计数器
        //清除 PCA 阵列溢出标志位 CF
        //清除模块 0、模块 1 中断标志位 CCF0、CCF1

    AUXR1 = 0X7F;//输出翻转，从 P4.2 和 P4.3 输出

    CL = 0;//清零计数器
    CH = 0;

    CMOD = 0X04;//设置 PCA 工作模式
        //空闲模式下 PCA 计数器继续工作
        //PCA 计数器计数频率取决于定时器 0 溢出率
        //关闭 PCA 计数器溢出中断

    CCAP0H = CCAP0L = 0XE0;//PWM0 输出频率为 50Hz 占空比为 5% 的 PWM 信号,
在 P4.2 输出
    CCAPM0 = 0X42;//模块 0 为 PWM 输出模式
        //没有 PCA 中断

    CCAP1H = CCAP1L = 0XE0;//PWM1 输出频率为 50Hz 占空比为 5% 的 PWM 信号,
在 P4.3 输出
    CCAPM1 = 0X42;//模块 1 为 PWM 输出模式
        //没有 PCA 中断

    CR = 1;//启动 PCA 计数器
```

```

}

//定时器0 初始化函数
void Timer0_Init(void)
{
    TMOD = 0X02;

    TL0 = TH0 = 256 - 78;

    TR0 = 1;
    ET0 = 1;
    EA = 1;
}

//舵机1角度控制函数，180°分25等份每份约7度，即angle1越大转的角度越大，angle1取值范围为0~25
//初始化时舵机的角度为0°
void Steering1(uchar angle1)
{
    CCAP0H = CCAP0L = 0XE0 + angle1;
}

//舵机2角度控制函数，180°分25等份每份约7度，即angle2越大转的角度越大，angle2取值范围为0~25
//初始化时舵机的角度为0°
void Steering2(uchar angle2)
{
    CCAP1H = CCAP1L = 0XE0 + angle2;
}

void Steering_Test(void)
{
    Steering1(0);
    Steering2(12);
}

```

在 steering.h 中写入以下函数：

```

#ifndef _STREEING_H_
#define _STREEING_H_

#define uint unsigned int
#define uchar unsigned char

void Steering_Init(void);
void Timer0_Init(void);
void Steering1(uchar angle1);
void Steering2(uchar angle2);
void Steering_Test(void);

```

```
void Delay_ms(uint z);  
#endif
```

在 main.c 中写入以下函数：

```
#include "stc12.h"  
#include "steering.h"  
  
void main()  
{  
    Steering_Init();  
    Timer0_Init();  
  
    while(1)  
    {  
        Steering_Test();  
    }  
}
```

控制舵机的 PWM 信号频率为 50Hz。PCA 计数器时钟源采用系统时钟的分频的话，频率都在兆级别的，所以 PCA 的计数器时钟源要用定时器 0 的溢出率。我们知道 PCA 输出 PWM 频率=计数器时钟源频率/256，所以我们可以倒算出定时器 0 的溢出率为每 78us 溢出一次。

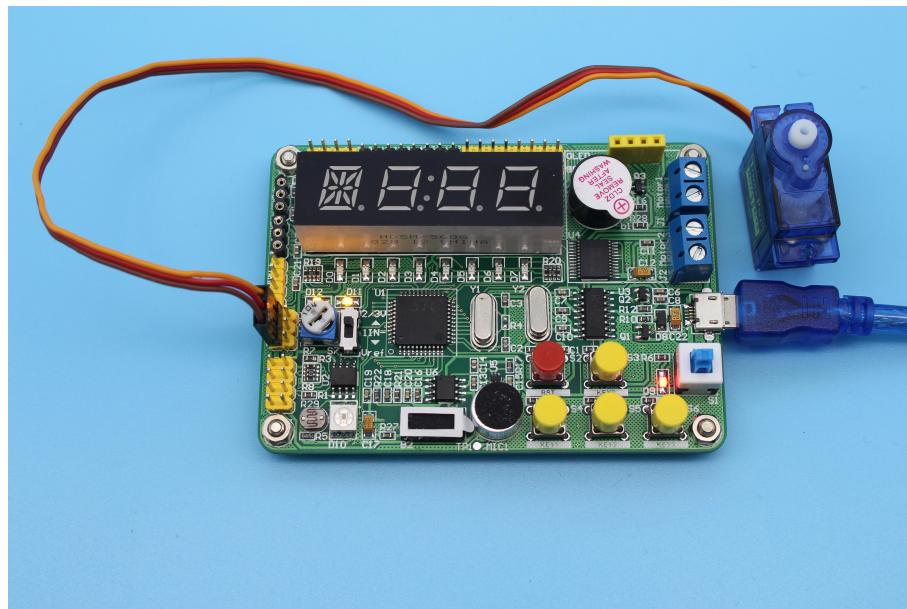
决定好频率后，我们就要控制占空比来调节舵机转角了。控制舵机的占空比范围为 12.5% 到 2.5%，对应着给 CCALxL 的值约为 249 到 224。这样 224 到 249 差 25，对应着舵机转角 0 到 180 度。所以平均下来，每给 CCALxL 加 1，舵机的转角改变约 7 度。

为了方便舵机控制，我们封装了一个舵机控制函数 Steeringx()。它接收的型参取值范围为 0 到 25，对应着 0 到 180 度。

3.4 下载验证

把工程编译生成的 hex 文件烧录进单片机中。

为了直观展现角度舵机的工作状态我们封装了一个舵机测试函数 Steering_Test()。Steering_Test() 函数在两路舵机输出端均有输出。它控制舵机以每秒 7 度的速度转动。当转动满 180 度时返回初始状态，循环转动。



3.5 思考总结

在这节开始时，我们提到了周转舵机。其实我们不讲周转舵机的控制方法，是因为它的控制和角度舵机大同小异。

周转舵机的控制同样用到 50Hz 的 PWM 方波。与角度舵机的占空比对应着转角不同，周转舵机的占空比控制着转动方向和转动速度：占空比越大，正向转速越大；占空比越小，反向转速越大。

那有没有一个中值占空比，来区分正转和反转呢。答案是不存在的。我们看下图，周转舵机侧面我们可以看到一个电位器。通过调节它，我们就可以调节周转舵机的中值占空比。



图 4.3.6 周转舵机的电位器

通常我们的做法是，先给周转舵机一个 50Hz，占空比为 50% 的 PWM 方波。一般情况下，周转舵机已经开始转动了。然后调节电位器，使周转舵机停下来。这里有个小

技巧，当周转舵机快停下来时，降低电位器的转动速度，可以更精准地校正角度舵机。

周转舵机和角度舵机在外表上像是一对双胞胎。所以这个周转舵机才有的电位器就成为我们区分角度舵机和周转舵机的重要标志了。

四、直流电机——机电一体化大门 2

4.1 知识点讲解

和舵机是一门大类一样，直流电机也有很多门类。这里我们只玩最简单的小直流电机，也就是我们日程生活中经常见到的小马达。相对于角度舵机，直流电机的控制就相对简单多了。直流电机只有两个控制脚，直接供电它就会转动。如果反向供电，它也会随之反向转动。

我们用单片机控制直流电机可不像驱动 LED 那样，一端供电，一端接 IO 口控制。一来这样不能控制电机的转动方向，二来单片机的引脚负载电流根本无法驱动直流电机。这时，我们就得用到 H 桥。

H 桥是一种典型的直流电机控制电路，因为它的电路形状酷似字母 H，因此我们叫它 H 桥。下面是一种三极管 H 桥的电路简图：

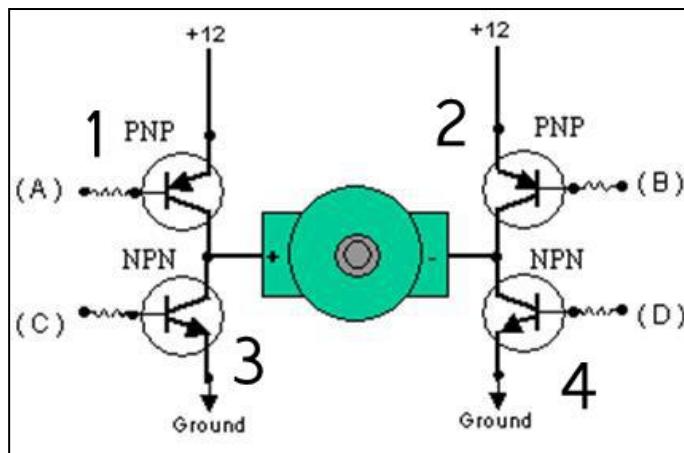


图 4.4.1 H 桥电路简图

绿色是直流电机，直流电机上标有正负。如果电流从正极流向负极则电机正转，电流从负极流向正极则电机反转。4个三极管里有两个 PNP 三极管和两个 NPN 三极管，从左到右从上到下我们给它们编号 1、2、3 和 4。4 个三极管的基极 A、B、C 和 D 为控制端。如果将 A 和 C 置 0，B 和 D 置 1，则 1 号和 4 号三极管将导通而 2 号和 3 号三极管将截止。这样外接的 12V 电源的电流将流过 1 号三极管，流进电机正极，然后从电机负极流出，流过 4 号三极管后接地。这样电机就实现正转了。反过来，如果将 A 和 C 置 1，B 和 D 置 0，则 1 号和 4 号三极管将截止而 2 号和 3 号三极管将导通。电流将从电机负极流向正极，电机反转。

实验班上的电机驱动芯片是 TB6612FNG，它能独立双向控制两个直流电机。它的引脚图如下：

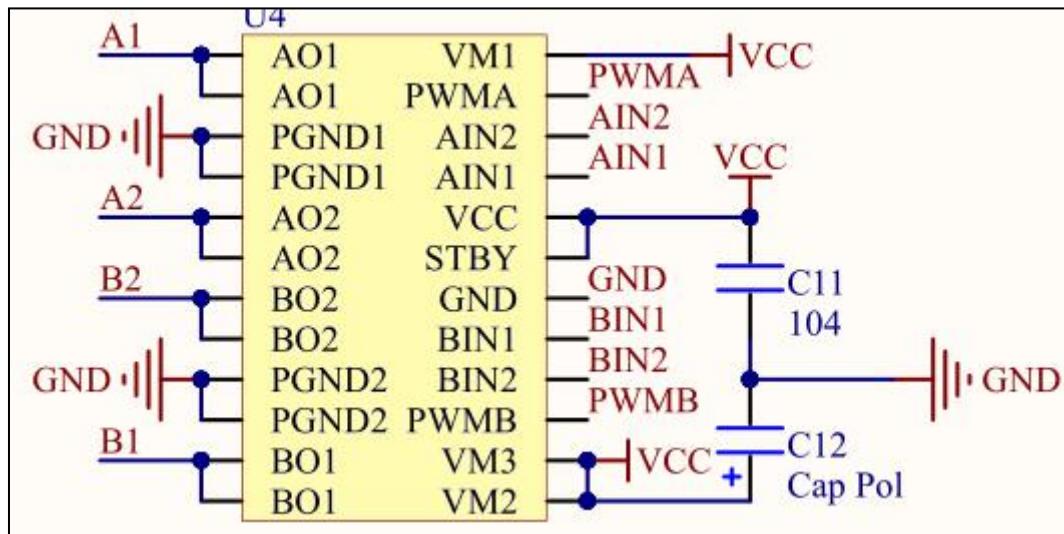


图 4.4.2 电机驱动模块电路连接

AO1 和 AO2、BO1 和 BO2 分别为两路电机控制的输出端，可以分别接两个直流电机。我们规定电流从 AO1 (BO1) 端流入电机，AO2 (BO2) 端流出电机时电机的转向为正转。AIN1 和 AIN2、BIN1 和 BIN2 分别为两路电机的控制端。PWMA、PWMB 为控制两个电机的 PWM 信号输入端。

IN1 和 IN2 组合有 4 种情况，分别对应 4 种电机的运动状态：

IN1	IN2	电机运动状态
0	0	制动
0	1	反转
1	0	正转
1	1	制动

表 4.1.1 IN1 和 IN2 与电机运动状态的关系

在前面的 PWM 控制 LED 的亮度中，当 PWM 中的高电平时 LED 熄灭，低电平时 LED 点亮。我们通过调节 PWM 的占空比就可以调节 LED 的亮度。同样的道理，我们只要通过 PWM 方波的占空比，控制 1 个 PWM 信号周期内电机转和不转的时间来调节电机

的转速。但控制电机转速的 PWM 信号频率不能太高也不能太低。太低了，电机会震动有噪声；太高了，电机就会转不起来。具体频率，我们要根据实际情况和相关手册确定，这里对于小直流电机，我们选用频率为 10KHz 左右的 PWM 信号。

4.2 软件实现

新建工程，命名为“电机”。在 HARDWARE 文件中新建文件夹 MOTOR，文件下新建 motor.c 和 motor.h。在工程所在的 USER 中新建文件 main.c。

在 motor.c 中写入以下程序：

```
#include "stc12.h"
#include "motor.h"

//电机初始化函数
void Motor_Init(void)
{
    CCON = 0X00;//初始化 CCON
        //关闭 PCA 计数器
        //清除 PCA 阵列溢出标志位 CF
        //清除模块 0、模块 1 中断标志位 CCF0、CCF1

    CL = 0;//清零计数器
    CH = 0;

    CMOD = 0XA;//设置 PCA 工作模式
        //空闲模式下 PCA 计数器继续工作
        //PCA 计数器计数频率为系统时钟频率的 1/4
        //关闭 PCA 计数器溢出中断

    CCAP0H = CCAP0L = 0xFF;//PWM0 输出频率为 11KHz 占空比为 0% 的 PWM 信号,
在 P1.3 输出
    CCAPM0 = 0X42;//模块 0 为 PWM 输出模式
        //没有 PCA 中断

    CCAP1H = CCAP1L = 0xFF;//PWM1 输出频率为 11KHz 占空比为 0% 的 PWM 信号,
在 P1.4 输出
    CCAPM1 = 0X42;//模块 1 为 PWM 输出模式
        //没有 PCA 中断
```

```

    CR = 1;//启动 PCA 计数器
}

//电机 1 控制函数
//pos1 控制电机 1 状态 , 0 后退 , 1 停止 , 2 前进
//pwm1 控制电机 1 转速 , pwm1 越大转速越大 , 取值范围为 0~50
void Motor1(uchar pos1,uchar pwm1)
{
    //判断电机状态
    switch(pos1)
    {
        case 0 : AIN1 = 1;
                  AIN2 = 0;
        break;
        case 1 : AIN1 = 1;
                  AIN2 = 1;
        break;
        case 2 : AIN1 = 0;
                  AIN2 = 1;
        break;
        default : AIN1 = 1;
                  AIN2 = 1;
    }
}

//调整电机转速
if(pwm1 >= 0 && pwm1 <= 50)
{
    pwm1 = 250 - pwm1;

    CCAP0H = CCAP0L = pwm1;
}
else
    CCAP0H = CCAP0L = 255;
}

//电机 2 控制函数
//pos2 控制电机 2 状态 , 0 后退 , 1 停止 , 2 前进
//pwm2 控制电机 2 转速 , pwm2 越大转速越大 , 取值范围为 0~50
void Motor2(uchar pos2,uchar pwm2)
{
    //判断电机状态
    switch(pos2)
    {
        case 0 : BIN1 = 1;
                  BIN2 = 0;
        break;
        case 1 : BIN1 = 1;
                  BIN2 = 1;
    }
}

```

```

        break;
    case 2 : BIN1 = 0;
               BIN2 = 1;
        break;
    default : BIN1 = 1;
               BIN2 = 1;
}
}

//调整电机转速
if(pwm2 >= 0 && pwm2 <= 50)
{
    pwm2 = 250 - pwm2;

    CCAP1H = CCAP1L = pwm2;
}
else
    CCAP1H = CCAP1L = 255;
}

//电机驱动函数
//直接控制小车两个电机运动
void Car_Run(uchar pos1,uchar speed1,uchar pos2,uchar speed2)
{
    Motor1(pos1,speed1);
    Motor2(pos2,speed2);
}

```

在 motor.h 中写入以下程序：

```

#ifndef _MOTOR_H
#define _MOTOR_H

#define uint unsigned int
#define uchar unsigned char

#define FORW 2
#define STOP 1
#define BACK 0

sbit AIN1 = P0^0;
sbit AIN2 = P0^1;
sbit BIN1 = P0^2;
sbit BIN2 = P0^3;

void Motor_Init(void);
void Motor1(uchar pos1,uchar pwm1);
void Motor2(uchar pos2,uchar pwm2);
void Car_Run(uchar pos1,uchar speed1,uchar pos2,uchar speed2);

#endif

```

在 main.c 中写入以下程序：

```
#include "stc12.h"
#include "motor.h"

void main()
{
    Motor_Init();

    while(1)
    {
        Car_Run(FORW,20,FORW,20);
    }
}
```

C 文件 motor.c 里主体还是 PCA 模块的配置。我们控制电机的 PWM 频率最好在 10KHz 左右，所以我们选择系统时钟频率的 1/4 作为 PCA 计数源频率。通过公式换算，得到输出 PWM 的频率约 11KHz。

配置好 PCA 后，我们就能通过改变 CCAPxL 的值来改变 PWM 的占空比，从而改变电机的转速。但 PCA 配置是工程的底层，如果我们每次控制 CCAPxL 来控制电机转速，既不方便，也不美观。而且我们还要控制电机的转向。

```
//电机 1 控制函数
//pos1 控制电机 1 状态，0 后退，1 停止，2 前进
//pwm1 控制电机 1 转速，pwm1 越大转速越大，取值范围为 0~50
void Motor1(uchar pos1,uchar pwm1)
{
    //判断电机状态
    switch(pos1)
    {
        case 0 : AIN1 = 1;
                  AIN2 = 0;
        break;
        case 1 : AIN1 = 1;
                  AIN2 = 1;
        break;
        case 2 : AIN1 = 0;
                  AIN2 = 1;
        break;
        default : AIN1 = 1;
                  AIN2 = 1;
    }

    //调整电机转速
    if(pwm1 >= 0 && pwm1 <= 50)
    {
        pwm1 = 250 - pwm1;
```

```
    CCAP0H = CCAP0L = pwm1;
}
else
    CCAP0H = CCAP0L = 255;
}
```

为此，我们封装了一个电机驱动函数 Motorx()。Motorx()函数接受两个变量：第一个变量 posx 控制电机的转向；第二个变量 pwmx 控制 PWM 的占空比，即电机的转速。

对于 posx，我们做了点手脚。我们可以看看 Motorx() 函数对电机转向的控制：通过判断 posx 的值，来设置电机的两个控制端——IN1 和 IN2。所以我们直接 Motorx() 函数第一个参数送 0、1 和 2 就能控制电机转向了。等等！0、1 和 2 哪个是前进？哪个是后退？虽然我们可以在 motor.c 作充分的注释，但在 main.c 中你不可能每次调用都过去看看 0、1 和 2 各代表什么状态吧？

为了解决这个问题，我们用到了一个非常实用的方法——宏定义大法！在头文件 motor.h 中，我们用#define 语句，将 FORW、STOP 和 BACK 宏定义好。所以在调用 Motorx() 函数时就直接用 FORW、STOP 和 BACK，这样就可以直接明了地控制电机的转向了。

转向控制解决了，我们就要解决转速的控制。我们知道，给 CCAPxL 的值越大，PWM 的占空比越小，则电机转得越慢。那如果我们调用 Motorx() 函数时，给的 pwmx 越大，电机转得越慢。这样不管怎样用都觉得很别扭，不符合我们的日常思维。所以我们将 pwmx 进行了一点处理：用 250 减去 pwmx 后再重新赋给 pwmx。对于我们配套的电机，只有 CCAPxL 在 200 到 250 之间能控制，且 CCAPxL 越大，电机转动越慢。所以我们用 250 减去 pwmx 在赋给 CCAPxL，就变成 pwmx 从 0 到 50 控制电机转动，且 pwmx 越大，电机转动越快。再给这样一来我们给的 pwmx 越大，PWM 的占空比就越大，电机也就转得越快。

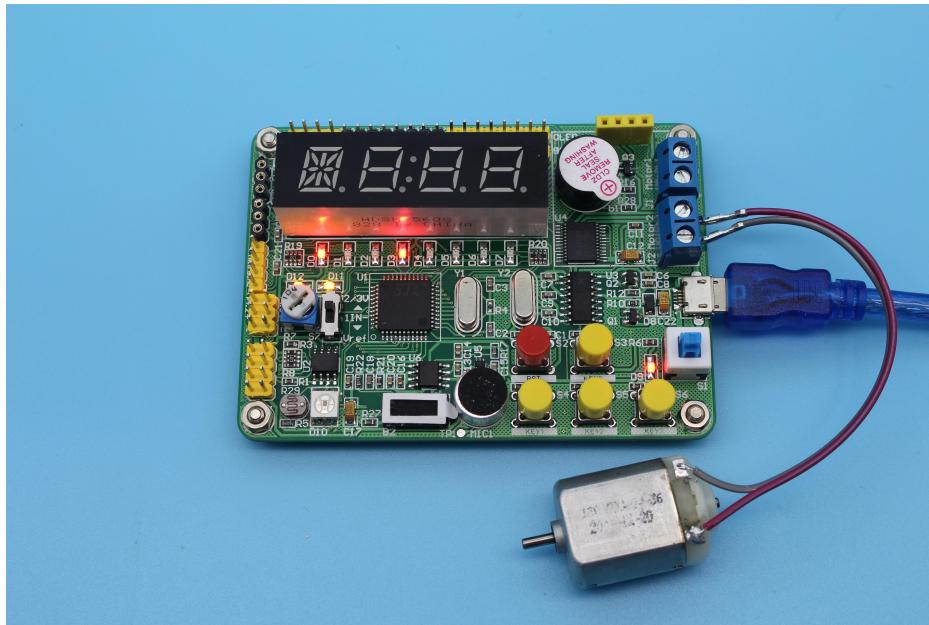
但还有一个小小的问题，那就是电机的最小转速问题。因为电机内部的阻力问题，只有我们给的 PWM 占空比在一定范围之内，电机才会工作。所以在你使用新的电机时，需要测试 PWM 占空比范围，再修改 Motorx() 函数。

我们使用电机大多是为了驱动小车。小车有两个轮子，对应着两个电机。我们分开控制的话，每次就会用两个函数，使用起来不直观也不方便。所以我们封装了函数 Car_Run()。Car_Run() 函数是控制两个电机的函数。它接收 4 个变量，分别控制两个电机的转向和转速。

4.3 下载验证

把工程编译生成的 hex 文件烧录进单片机中。

将两个电机分别接入实验板上两个电机端口。调整函数 Car_Run()中的边变量，观察两个电机的转速，看看哪个快，哪个慢。



4.4 思考总结

作为一个单片机开发者，你要时刻记住你是硬件和软件的主人，而不是被它们所牵着鼻子走。例如之前数码管的位选，1为熄灭，0为点亮；亦或这节中本来 CCAPxL 的值越大，电机转速却越慢。这些都因为硬件上的原因，造成和我们的日常思维有出入。这时我们不能将就着写程序，而是要开动脑筋，想办法处理数据，使我们封装的函数更贴近我们的日常使用习惯，将大大提高我们的编程效率。

和舵机一样，在选用电机前，也要留意电机的各个参数。常用的参数如尺寸、工作电压、扭矩等等。我们要根据需要选择合适的电机。

单单一个电机是无法胜任工作的。在我们大多数的电子设计中，都和小车有关。而电机充当着带动车轮工作的重要角色。可见电机作为小车的动力核心，它要装配在哪里，怎样装配，都影响着小车的运动性能。

五、喇叭——我是贝多芬

再最开始的入门篇第一节中，我们通过控制 IO 口电平来控制蜂鸣器。但那是的蜂鸣器只能发出“哔哔哔”的声音，而不能控制音调。只能“哔哔哔”地响却不能演奏音乐那是多么的无趣啊。那我们用 51 单片机能不能演奏出动耳的音乐呢？这就得用上我们接下来要讲的喇叭了！

5.1 硬件准备

我们实验板上有两种发声元件：一种是我们刚刚提到的蜂鸣器，另一种是喇叭。



图 4.5.1 实验板上的喇叭

这里的蜂鸣器准确点来说，叫有源蜂鸣器——它内部有自带的钼片和振荡电路。当我们给有源蜂鸣器供直流电时，内部的振荡电路能将恒定的直流电转换成一定频率的脉冲信号，产生磁场交变，从而带动钼片震动发音。换言之，有源蜂鸣器的发声频率是固定的，并不能发出各种音调。

但喇叭不同，它内部并没有振荡电路。如果我们要奏响无源蜂鸣器，就要给喇叭一个交变电流来带动喇叭里面鼓膜振动发声。通过单片机控制，我们给喇叭的交变电流的频率是可以改变的。因此，我们可以通过喇叭发出我们想要的音调，从而演奏出我们想要的音乐了。

5.2 知识点讲解

可控频率的交变电流信号，这个重任非 PWM 莫属了。的确，网上一般的用 PWM 控制喇叭的教程是通过直接改变频率的方法来调节音调的。但我们为了配合下一节要讲的咪头，尝试了一种新的调节声调的手段。

首先看看我们的实验板原理图上找到喇叭模块的电路图：

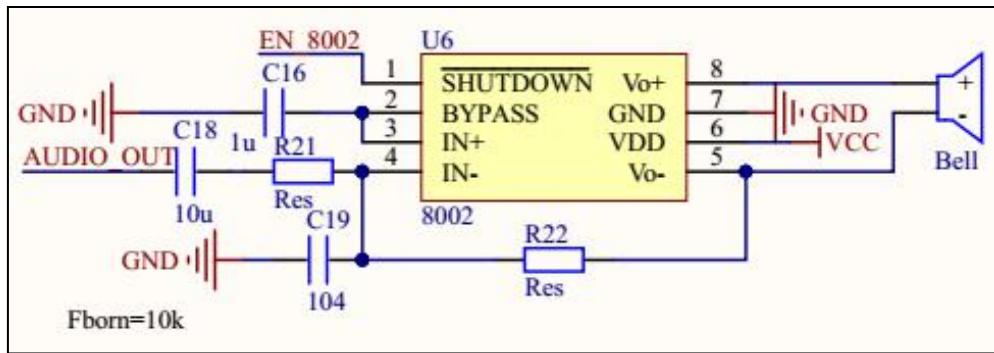


图 4.5.2 喇叭模块电路连接

8002A 是一款高保真音频功率放大芯片。他有两个音频输入脚：IN+和 IN-，分别为左右音频输入端。因为我们只是单声道，所以只用到右音频输入端，而左音频输入段接地。音频信号经过内部功率放大电路后输出到两个接喇叭的输出端：VO+和 VO-。有一点要注意的是，8002A 的使能端是低电平有效的。也就是说，当给 8002A 的使能端低电平时，芯片才处于工作状态。我们也可以仔细观察芯片使能端的与芯片交接处有一个小圆圈，这说明这个端口是低电平有效的。

网上一般的 PWM 控制喇叭教程，是给 8002A 的音频输入端一个稳定的 PWM 信号。通过改变 PWM 信号的频率来控制音调的。但这个方法并不适用我们现在的电路，因为我们这个电路是要将 PWM 信号转换为电压信号的！

原理图中电容 C18 是一个滤波电容，它对我们分析电路影响甚小可以忽略。再分析电阻 R22，音频信号再经过它之前就已经进入 IN-了，因此也可以忽略它。于是我们得到了简化后的音频输入电路图：

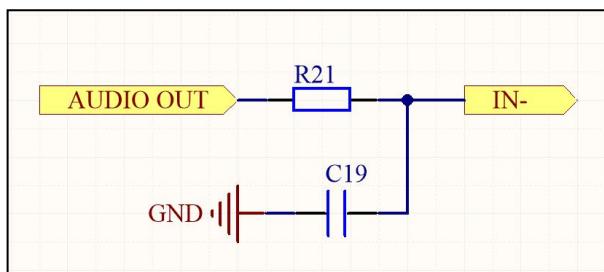


图 4.5.3 简化后的音频输入电路

这时一个由一个电阻和一个电容组成的一阶阻容电路。电容 C19 一极接地，另外一极与 8002A 的音频输入端 IN-相连。因为与地想接的一极电压为 0，所以输入 IN-的电压与电容 C19 非接地一端相等。换言之，我们越给电容 C19 充电，IN-的电压就越大。

但控制电容充电多少何谈容易。对于一阶阻容电路，电容充电时间 $t=R*C$ 。这样一算下来，我们现在的电路给 C19 充满电的时间只要 10us！这时，我们就要请出我们的主角：PWM 了！

如果我们给音频输入电路一个 PWM 信号。一个周期内，当高电平时就相当于给电容 C19 充电，低电平时就相当于给 C19 放电。当这个 PWM 信号频率够高时（这个阈值与一阶阻容电路中电阻和电容的大小有关，实验板上的阈值为 30KHz），C19 非接地端的电压就能稳定在一个特定的电压上了。

根据 $Q = I * t$ 和 $U = Q / C$ ，电流一定时电容充电电压与充电时间成正比。如今我们的 PWM 的频率是一定的，那占空比越大，每个周期内高电平时间就越长，充电时间也就越长。充电时间越长，电容 C19 非接地端电压就越大。如果占空比为 0% 时，C19 非接地端的电压为 0V。如果占空比为 100% 时就相当于将 C19 非接地端直接接 VCC，因此 C19 非接地端电压将为 5V。所以我们可以得出结论：当占空比由 0% 增加到 100% 时，8002A 音频输入端电压从 0V 到 5V 线性增大。

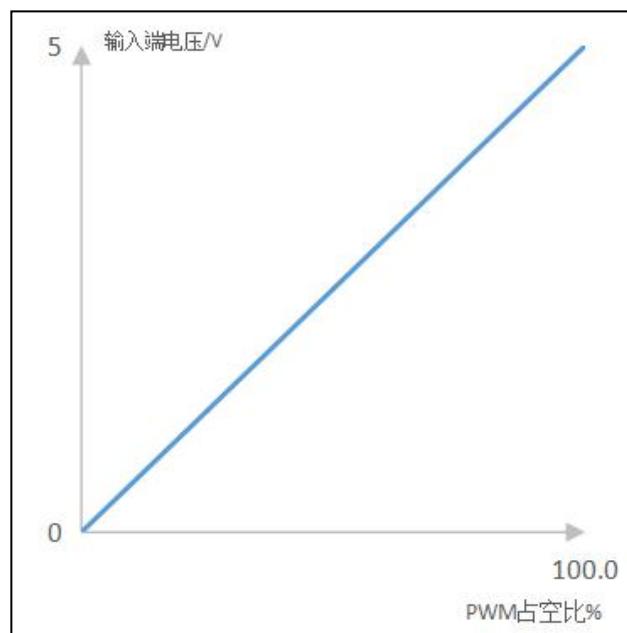


图 4.5.4 PWM 占空比与输入电压的关系

5.3 软件实现

新建工程，命名为“喇叭”。在 HARDWARE 文件中新建文件夹 BORN，文件下新建 born.c、born.h 和 music.h。在工程所在的 USER 中新建文件 main.c。

在 born.c 中写入以下程序：

```

#include "stc12.h"
#include "born.h"

/*-----
53-低音 1  26-中音 1
50-低音 1#  25-中音 1#
47-低音 2  24-中音 2
44-低音 2#  23-中音 2#
42-低音 3  21-中音 3
40-低音 4  20-中音 4
38-低音 4#  19-中音 4#
36-低音 5  17-中音 5
34-低音 5#  16-中音 5#
32-低音 6  15-中音 6
30-低音 6#  14-中音 6#
28-低音 7  13-中音 7
-----*/
//模拟正弦波所需的数组
uchar code table[] =
{
0X80,0X54,0X2D,0X03,0X02,
0X02,0X03,0X2D,0X54,0X80,
0XAA,0XD1,0XEF,0XFC,0XFC,
0XEF,0XD1,0XAA
};

//喇叭初始化函数
void Born_Init(void)
{
    CCON = 0X00;//初始化 CCON
        //关闭 PCA 计数器
        //清除 PCA 阵列溢出标志位 CF
        //清除模块 0、模块 1 中断标志位 CCF0、CCF1

    AUXR1 = 0X7F;//翻转，在 P4.2 输出

    CL = 0;//清零计数器
    CH = 0;

    CMOD = 0X08;//设置 PCA 工作模式
        //空闲模式下 PCA 计数器继续工作
}

```

```

        // PCA 计数器计数频率为 12MHz
        //关闭 PCA 计数器溢出中断

CCAP0H = CCAP0L = 0X80;//PWM0 输出频率为 46K,占空比为 50% 的 PWM 信号

CCAPM0 = 0X42;//模块 0 为 PWM 输出 0 模式
                //没有 PCA 中断

CR = 1;//启动 PCA 计数器
EN = 0;
}

//音调函数
//pit 控制音调，取值范围为上方注释中的值
void Pitch(uint pit)
{
    uint x,y;

    y = BEAT;

    while(y--)
    {
        for(x = 0;x < 19;x++)
        {
            CCAP0H = CCAP0L = table[x];
            Delay_4us(pit);
        }
    }
}

//演奏音乐函数
//型参为音乐数组名字
void Music(uint *musictable)
{
    uint x;

    for(x = 0;musictable[x] != 255;x++)
    {
        Pitch(musictable[x]);
    }
}

//4us 延时函数
void Delay_4us(uint z)
{
    while(z--);
}

```

在 born.h 中写入以下程序：

```
#ifndef _BORN_H_
#define _BORN_H_

#define uint unsigned int
#define uchar unsigned char

#define BEAT 600//节拍常熟，越小越快

sbit EN = P0^5;

void Born_Init(void);
void Pitch(uint pit);
void Music(uint *musictable);
void Delay_4us(uint z);

#endif
```

在 music.c 中写入以下程序：

```
/*
53-低音 1 26-中音 1
50-低音 1# 25-中音 1#
47-低音 2 24-中音 2
44-低音 2# 23-中音 2#
42-低音 3 21-中音 3
40-低音 4 20-中音 4
38-低音 4# 19-中音 4#
36-低音 5 17-中音 5
34-低音 5# 16-中音 5#
32-低音 6 15-中音 6
30-低音 6# 14-中音 6#
28-低音 7 13-中音 7
-----*/
code uint LittleStar[] =
{
26,0,26,0,17,0,17,0,15,0,15,0,17,0,
20,0,20,0,21,0,21,0,24,0,24,0,26,0,
26,0,26,0,17,0,17,0,15,0,15,0,17,0,
17,0,17,0,20,0,20,0,21,0,21,0,24,0,
17,0,17,0,20,0,20,0,21,0,21,0,24,0,
26,0,26,0,17,0,17,0,15,0,15,0,17,0,
```

```
20,0,20,0,21,0,21,0,24,0,24,0,26,255  
};
```

在 music.h 中写入以下程序：

```
#ifndef _MUSIC_H_  
#define _MUSIC_H_  
  
#define uint unsigned int  
#define uchar unsigned char  
  
code uint LittleStar[];  
  
#endif
```

在 main.c 中写入以下程序：

```
#include "stc12.h"  
#include "born.h"  
#include "music.h"  
  
void main()  
{  
    Born_init();  
  
    while(1)  
    {  
        Music(LittleStar);  
    }  
}
```

关于 PCA 的配置，我们前面已将用了很多次了，所以我们也不再赘述。我们重点看看是如何通过调节 PWM 来调节喇叭发出声音的音调，从而演奏歌曲的。

首先我们先上一节乐理课。

声音是由震动产生的。我们可以拿起一把铁尺，一头摁在桌子边沿，然后用手另一头拨动伸出桌子的一部分。通过这个小实验我们发现，铁尺伸出桌子的一部分越长，震动频率越快，声音越尖锐，反之亦然。可见声音的音调和震动的频率是纯在对应关系的。下表是标准音调对应的震动频率。

音调	频率	音调	频率	音调	频率
低音 1	262	中音 1	523	高音 1	1046
低音 1#	277	中音 1#	554	高音 1#	1109

低音 2	294	中音 2	587	高音 2	1175
低音 2#	311	中音 2#	622	高音 2#	1245
低音 3	330	中音 3	659	高音 3	1318
低音 4	349	中音 4	698	高音 4	1397
低音 4#	370	中音 4#	740	高音 4#	1480
低音 5	392	中音 5	784	高音 5	1568
低音 5#	415	中音 5#	831	高音 5#	1661
低音 6	440	中音 6	880	高音 6	1760
低音 6#	466	中音 6#	932	高音 6#	1865
低音 7	494	中音 7	988	高音 7	1976

表 4.5.1 音调与频率对照表

我们实验板上的喇叭是靠电流的震动来带动内部的鼓膜振动发声。换言之，电流的震动频率就决定了鼓膜的震动频率，进而决定了喇叭发出声音的音调。

说到电流的震动，我们一直在用，那就是 PWM 信号。的确，网上其他教程在调节音调时，都是直接通过控制 PWM 的频率，来直接控制音调的。但随大流的事，我们懒得去做！实验板上我们给 8002A 芯片的输入端设计了一个 PWM 转电压电路，目的是要通过模拟正弦波的方式来控制音调。

正弦波信号是模拟量，我们要把它数字量化。我们将正弦波截成 18 份，中间节点共 17 个。我们通过控制 PWM 的占空比，来模拟每个点的电压，依次循环，就模拟出一个正弦波信号了。如果我们要控制正弦波频率，就要控制每个点之间的时间间隔。

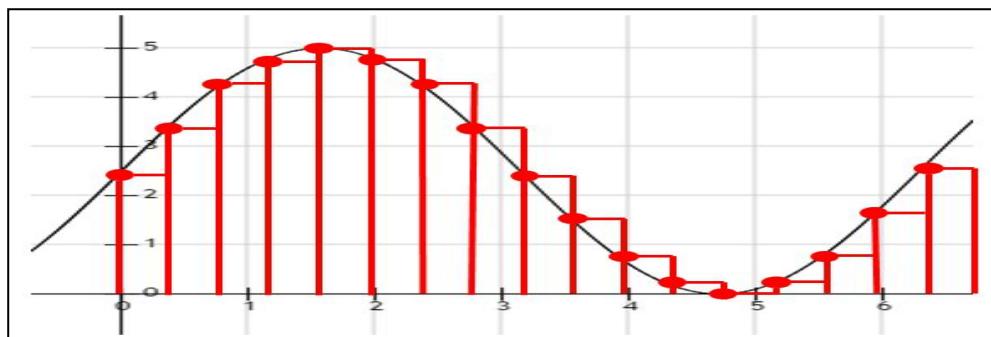


图 4.5.5 模拟正弦波示意图

所以每个点的电压和对应的占空比大家可以手动算一下。为了节省时间，我已经整理好每个点对应的电压值、占空比和 pit 值如下表：

相位	电压	占空比	Pit 值	相位	电压	占空比	Pit 值
0	2.5	50%	0X80	180	2.5	50%	0X80
20	3.35	67%	0X54	200	1.65	33%	0XAA
40	4.10	82%	0X2D	220	0.90	18%	0XD1
60	4.68	94%	0X03	240	0.33	6%	0XEF
80	4.95	99%	0X02	260	0.05	1%	0XFC
100	4.95	99%	0X02	280	0.05	1%	0XFC
120	4.68	94%	0X03	300	0.33	6%	0XEF
140	4.10	82%	0X2D	320	0.90	18%	0XD1
160	3.35	67%	0X54	340	1.65	33%	0XAA

表 4.5.2

```
//音调函数
//pit 控制音调，取值范围为上方注释中的值
void Pitch(uint pit)
{
    uint x,y;

    y = BEAT;

    while(y--)
    {
        for(x = 0;x < 19;x++)
        {
            CCAP0H = CCAP0L = table[x];
            Delay_4us(pit);
        }
    }
}
```

在这个基础上，我们就有了音调函数 Pitch()。Pitch()函数接受一个变量，这个变量控制着每个点的之间的时间间隔。算上 18 倍，就是正弦波的频率，即喇叭发出声音的

频率。这个也是可以计算出来的。根据表，我们将各个音调的频率先倒数求得震动周期，再除以 18，得到各个音调对应的模拟正弦波每个点之间的时间间隔。STC12 能做到的最小的延时函数 Delay_4us() 执行一次要 4us，所以将个点的时间间隔再除以 4us 就得到了 pit 值。低音和中音对应的 pit 参数我们已经算出来了，并以注释的形式写在工程的开头部分。你可以试着调用 Pitch() 函数，改变 pit 值，观察喇叭发出声音的音调变化。

```
//演奏音乐函数
//型参为音乐数组名字
void Music(uint *musictable)
{
    uint x;

    for(x = 0;musictable[x] != 255;x++)
    {
        Pitch(musictable[x]);
    }
}
```

有了音调函数后，我们就能开始演奏音乐了。我们编写了演奏音乐函数 Music()。函数 Music() 接受一个数组变量，而这个数组存放着我们要播放的音乐的音调信息。为了整个工程的美观简洁，我们将音乐数组放在独立的文件 music.h 中。

以例程中的音乐数组 LittleStar 为例。《小星星》的音乐简谱大家可以在网上随便找到，然后将每个音调对应的时间间隔存入数组中。在每个音调间，我们都放一个零，这是为什么呢？想想看，如果正弦波每个点的时间间隔我们给零，模拟正弦波将变成恒定的某个值，相当于一个很定的电压。我们知道喇叭要接受交变电流才会发出声音。所以当时间间隔为零时，喇叭将不会发出声音。可见，每个音调之间我们都放一个零是用来作停顿的。而数组最后面的 255 是一个结束标志符，当执行到这个标志符时，音乐停止播放。

Music() 函数的功能就是将音乐数组的值依次赋给 Pitch() 函数，直至结束标志符的出现。我们把 Music() 函数定义成接受一个数组变量。所以我们直接给数组的名字就能演奏相应的曲目，方便简单。

另外，音乐是有节拍的。节拍越快，音乐越快；反之节拍越慢，音乐也就越慢。所以说白了，控制节拍，就是控制每个音调的时间。所以在音调函数 Pitch() 中，我们有一个 while 循环，循环内容是执行模拟正弦波。while 循环的时间就是每个音调的时间。我们在 born.c 中宏定义了一个节拍常数 BEAT。在函数 Pitch() 中的 while 循环就是执行 BEAT 次。可见 BEAT 越大，while 循环执行时间越长，节拍越慢，反之亦然。

5.4 下载验证

把工程编译生成的 hex 文件烧录进单片机中。

尝试着改变 BEAT 的值，观察节拍的快慢。另外你也可以上网找其他的音乐，将它“翻译”好，并用喇叭演奏它！

5.5 总结思考

从音调表我们发现一个尴尬的问题——只有低音和中音，高音却不见了！

其实不是我偷懒把高音部分吃了，而是无法做出来……首先我们回顾我们的音调函数 Pitch()。我们将音调对应的频率求倒数后除以 18 再除以 4us，得到的 pit 值送给 Pitch()，就能让喇叭发出对应的音调。

随着音调的增高，频率就越高，pit 值就越小。但 STC12 能做到的最短最短延时函数也就是 Delay_4us()——执行一次要 4us。音调频率取倒数后除以 18 后还要除以 4us，所以带来的问题是音调越高，两个相邻音调的 pit 值就越接近。我们看程序中注释的音调对应的 pit 值参数表。低音时相邻音调间 pit 值差 2，中音差 1。如果到了高音，pit 值就变得没有区分度了！

为了能发出高音，我们有两种解决方法：

一，改变模拟正弦波的方法。目前，我们将正弦波截成 18 段，这样就导致了音调频率取倒数后除以 18 再除以 4us 得到的 pit 值间差得很小。如果我们将正弦波截成更少的段数，音调频率取倒数后除以 18 的值就变大，除以 4us 后相邻音调之间的 pit 值差就越大；

二，换主控。这是最直接暴力的方法了。因为 STC12 本身的性能，我们不能做出 1us 级别的延时函数。搞得我们每次都要除以 4us，这就减小了相邻音调间 pit 值差变得很小。如果我们换用能产生 1us 延时的主控，如 STM32 或 Arduino，这个问题也就迎刃而解了。

六、咪头——模数转换进阶

还记得在入门篇电压比较器一节中，我们通过 LM393 芯片对输入电压进行简单的模数转换吗？那时候我们将输入电压与两个已知电压值作比较，这样只能大概知道待测电压与两个已知电压的相对大小关系，并不能得到一个比较准确的值。但如果我们增加的比较的次数，那不就能不断逼近模拟量对应的数字量吗？现在，我们就看看 stc12 内的逐次逼近式 AD 转换器。它可是 10 位 AD 比较器哦！也就是它能逐次比较 10 次！

6.1 硬件准备

既然我们要讲 AD 转换器（ADC），那肯定少不了要输入的模拟量源。这次我们用到了咪头：



图 4.6.1 实验板上的咪头

喇叭和咪头相反：前者是将电信号转换为声音信号，而后者是将声音信号转换为电信号。当外界有声音时，咪头的输出电阻就会随之变化。通过这个性质，咪头就实现了声音信号转换成电信号的第一步。但单片机不能直接测量电阻阻值的变化，所以我们用到了咪头增益放大器芯片：MAX9812。实验板上的原理图如下：

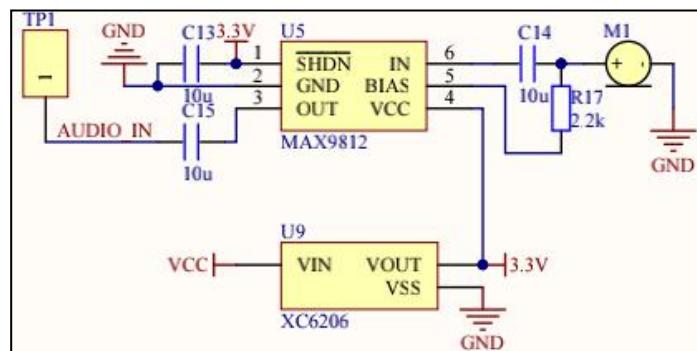


图 4.6.2 咪头模块电路连接

电路中咪头负极接地，正极经 C14 滤波后接 MAX9812 的输入端 IN。MAX9812 的输出电压由引脚 OUT 输出，随着咪头输出电阻的改变而改变。MAX9812 的电压输出端接单片机 P1.0。

6.2 知识点讲解

stc12 内置的 8 路 10 位高速 ADC 是逐次逼近式 (SAR) ADC。逐次逼近式 ADC 由一个比较器和 DAC (数模转换器) 构成。通过逐次比较逻辑，从最高位开始，顺序地对输入电压与内置 DAC 输出的电压值作比较。经过多次比较，使得到的数字量逐次逼近模拟量的对应值。逐次逼近型 ADC 具有速度快、功耗低等优点。除此之外，ADC 还有双积分式、V/F 式等。

stc12 有 8 路 ADC，这 8 路 ADC 的输入分别是 P1 的 8 个 IO 口。所以 stc12 能同时测取 8 个模拟量。

下图是 stc12 内部 ADC 的结构图，详见中文参考手册 P335：

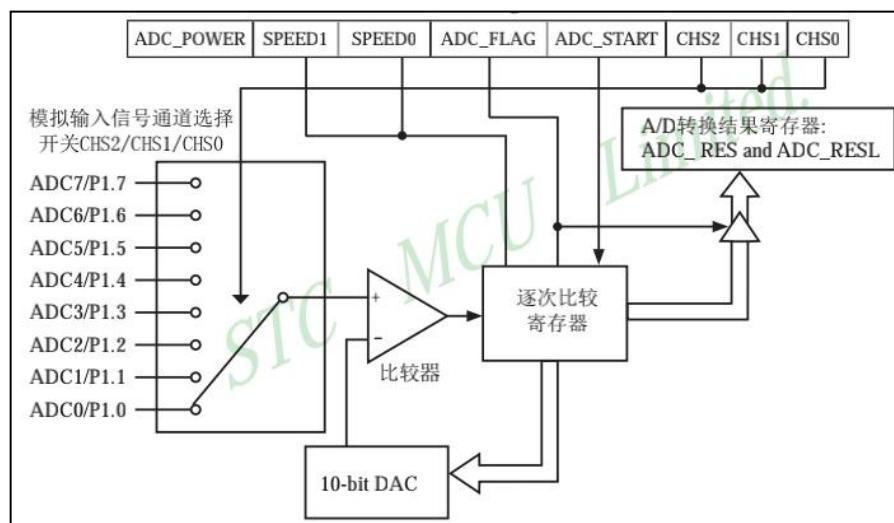


图 4.6.3 STC12 内部 ADC 结构图

接下来我们就通过软件配置 STC12 的内部 ADC，通过咪头把声信号转化为电信号。

6.3 软件实现

新建工程，命名为“咪头”。在 HARDWARE 文件中新建文件夹 AUDIO，文件下新建 audio.c 和 audio.h。从之前的工程中将文件夹 DIGITRON 复制到 HARDWARE 文件夹下。在工程所在的 USER 中新建文件 main.c。

在 audio.c 中写入以下程序：

```
#include "stc12.h"
#include "audio.h"
#include "digitron.h"

//咪头初始化函数
void Audio_Init(void)
{
    P1ASF = 0X01;//P1.0 作为 AD 功能使用
    ADC_CONTR = 0XE0;//打开 AD 转换器电源
        //最快 AD 转换速度，每 90 个时钟周期转换一次
        //将 P1.0 作为 ADC 输入
    ADC_RES = 0X00;//清零 AD 转换结果寄存器
}

//咪头数据读取函数
uchar Audio_Read(void)
{
    uchar valu,ADC_FLAG;

    ADC_CONTR |= 0X08;//启动 AD 转换

    ADC_FLAG = (ADC_CONTR & 0X10) >> 4;//提取 AD 转换完成标志

    if(ADC_FLAG == 1)//检查是否完成转换
    {
        ADC_CONTR &= 0XF0;//清零转换标志

        valu = ADC_RES;//记录数据
    }

    return valu;
}

//咪头数据显示函数
void Audio_Show(void)
{
```

```
uchar value;  
  
value = Audio_Read();//读取 AD 数据  
  
Digi_Num(value);//在数码管中显示数据  
}
```

在 audio.h 中写入以下程序：

```
#ifndef _AD_H_  
#define _AD_H_  
  
#define uint unsigned int  
#define uchar unsigned char  
  
void Audio_Init(void);  
uchar Audio_Read(void);  
void Audio_Show(void);  
  
#endif
```

在 main.c 中写入以下程序：

```
#include "stc12.h"  
#include "audio.h"  
#include "digitron.h"  
  
void main()  
{  
    Audio_Init();  
    Digi_Init();  
  
    while(1)  
    {  
        Audio_Show();  
    }  
}
```

在进行 AD 转换前，我们要进行相关的初始化配置。

STC12 的 ADC 输入端是 P1 的 8 位 IO。回想一下，我们在控制数码管前，因为 P4.4 和 P4.5 有复用功能，我们要配置寄存器 PS4W 将它们确认为当普通 IO 口来用。所以不用多说，这里我们同样要设置寄存器告诉单片机，我们要把 P1 的某个 IO 口充当 ADC 输入口。

STC12 有专门设置 P1 为 ADC 输入口的寄存器：P1ASF。P1ASF 是一个 8 位寄存器，刚好对应 P1 的 8 个 IO 口。如果我们要将 P1.1 设置成 ADC 输入口，我们就将 P1ASF 对应的位置 1。关于 P1ASF 的位定义，详见中文参考手册 P337。与本节相关的位定义如下：

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	P17ASF	P16ASF	P15ASF	P14ASF	P13ASF	P12ASF	P11ASF	P10ASF

表 3.4.1 **P1ASF** 相关位定义

P10ASF：置 0，将 P1.0 设置成普通 IO 口模式；

置 1，将 P1.0 设置成 ADC 模式。

设置好 IO 口模式后，我们就转向去配置内部 ADC 的相关寄存器。STC12 内部 ADC 主要由寄存器 ADC_CONTR 来控制。关于 ADC_CONTR 的位定义，详见中文参考手册 P338,。与本节相关的位定义如下：

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	ADC_POWER	SPEED1	SPEED0	ADC_F LAGE	ADC_S TART	CHS2	CHS1	CHS0

表 3.4.1 **ADC_CONTR** 相关位定义

ADC_POWER：ADC 电源控制位。置 0 关闭 ADC 电源，置 1 启动 ADC 电源。

SPEEDx：ADC 转换速度控制位。两位决定着四种 ADC 转换速度。

SPEED1	SPEED0	ADC 转换速度
0	0	540 个时钟周期转换一次
0	1	360 个时钟周期转换一次
1	0	180 个时钟周期转换一次
1	1	90 个时钟周期转换一次

表 3.4.2 **SPEEDx** 与 ADC 转换速度的关系

ADC_FLAGE：ADC 转换结束标志位。ADC 转换结束后由硬件置 1。需要软件清零。

ADC_START : ADC 启动控制位。置 0 停止 ADC 工作，置 1 启动 ADC 工作。

CHSx : 模拟输入通道选择。控制某个 IO 口作为 ADC 输入通道。

CH2	CH1	CH0	ADC 输入通道选择
0	0	0	P1.0 作为 ADC 输入通道
0	0	1	P1.1 作为 ADC 输入通道
0	1	0	P1.2 作为 ADC 输入通道
0	1	1	P1.3 作为 ADC 输入通道
1	0	0	P1.4 作为 ADC 输入通道
1	0	1	P1.5 作为 ADC 输入通道
1	1	0	P1.6 作为 ADC 输入通道
1	1	1	P1.7 作为 ADC 输入通道

表 3.4.3 CHx 和 ADC 输入通道选择

```
//咪头初始化函数
void Audio_Init(void)
{
    P1ASF = 0X01;//P1.0 作为 AD 功能使用
    ADC_CONTR = 0XE0;//打开 AD 转换器电源
        //最快 AD 转换速度，每 90 个时钟周期转换一次
        //将 P1.0 作为 ADC 输入
    ADC_RES = 0X00;//清零 AD 转换结果寄存器
}
```

从高位往低位看。配置 ADC 模式，第一步肯定先要打开 ADC 的电源，要将 ADC_CONTR 的第 7 位置 1。然后是 ADC 转换速度的选择。在这里，我们没有对 ADC 转换速度有特殊的要求，我们就将它设置成最快速度，即将 SPEED1 和 SPEED0 都置 1。因为我们正初始化 ADC，所以结束标志符和启动控制位我们可以跳过。音频芯片 MAX9812 的输出端接单片机 P1.0，而复位后 CHx 均为 0，为 P1.0 作为 ADC 输入通道。所以，我们将 0XE0 送给 ADC_CONTR，完成对 ADC 的配置。

```
//咪头数据读取函数
```

```
uchar Audio_Read(void)
{
    uchar valu,ADC_FLAG;

    ADC_CONTR |= 0X08;//启动 AD 转换

    ADC_FLAG = (ADC_CONTR & 0X10) >> 4;//提取 AD 转换完成标志

    if(ADC_FLAG == 1)//检查是否完成转换
    {
        ADC_CONTR &= 0XEF;//清零转换标志

        valu = ADC_RES;//记录数据
    }

    return valu;
}
```

在配置好 ADC 后，我们就可以读取 ADC 转换后得到的值了。ADC 转换时单片机自行完成的。我们只需要等待结束标志符的到来，然后读取 ADC 转换结果寄存器中的值即可。

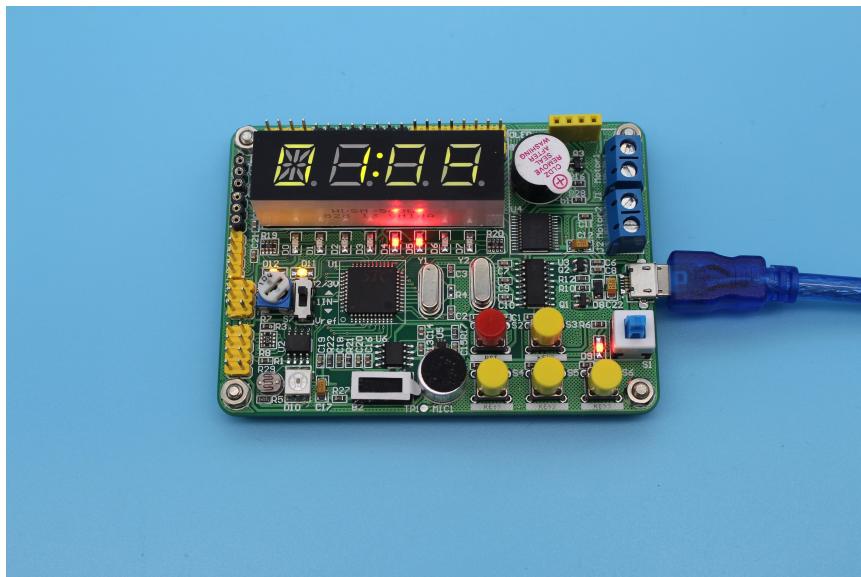
ADC 转换结果寄存器有两个，分别是 ADC_RES 和 ADC_RESL。通过设置寄存器 AUXR1 的第 2 位，ADC 转换结果将以不同形式存放在 ADC_RES 和 ADC_RESL。但这个无关紧要，我们可以不用配置 AUXR1，默认 10 位 ADC 转换结果 8 位存放在 ADC_RES 中，剩下两位存放在 ADC_RESL 中。而我们只用 ADC_RES 中的 8 位。

在读取到结束标志符 ADC_FLAG 置 1 后，我们要及时将它置零。然后将 ADC_RES 的值存放在声明好的变量 valu 中。剩下的，就是在数码管中把 ADC 的转换结果显示出来。

6.4 下载验证

将工程编译生成的 hex 文件烧录进单片机中。

拿出你的手机，播放一段音乐。将手机的喇叭靠近咪头，观察数码管上 ADC 转换结果。



6.5 总结思考

我们来总结一下 STC12 内部 ADC 进行一次 AD 转换所需要的流程：

- ①配置寄存器 P1ASF，将需要的 P1 某一 IO 口设置成 ADC 模式；
- ②配置寄存器 ADC_CONTR，打开 ADC 电源，设置 ADC 速度，设置 P1 某一 IO 口位 ADC 输入；
- ③置零寄存器 ADC_RES，清零 ADC 转换结果；
- ④启动 ADC；
- ⑤等待 ADC 结束标志位；
- ⑥检测到结束标志位后，清零标志位；
- ⑦读取 ADC 转换结果。

另外需要注意的，和定时器溢出标志位一样，ADC 的转换结束标志位 A DC_FLAGE 同样是需要软件清零的。在单片机的寄存器中，有些标志位在寄存器工作结束后会由硬件自动清零，而有些却要程序员自己软件清零。所以在单片机某个功能时，一定要仔细查看相关手册，看清楚标志位是硬件清零还是软件清零。

七、超声波测距——隔空测距于无形



图 4.7.1 超声波测距模块 SR04

超声波模块是测量距离的重要器件，在距离监测和自动避障等许多领域有着广泛的应用。在本节中，我们通过根据时序图来操作超声波模块，并掌握看时序图这一重要能力。

7.1 知识点讲解

超声波模块有四个端口，分别是：VCC+5V 电源端口，Trig 触发信号端，Echo 回响信号端，GND 接地端。

然后我们看看超声波模块 SR04 的操作时序图：

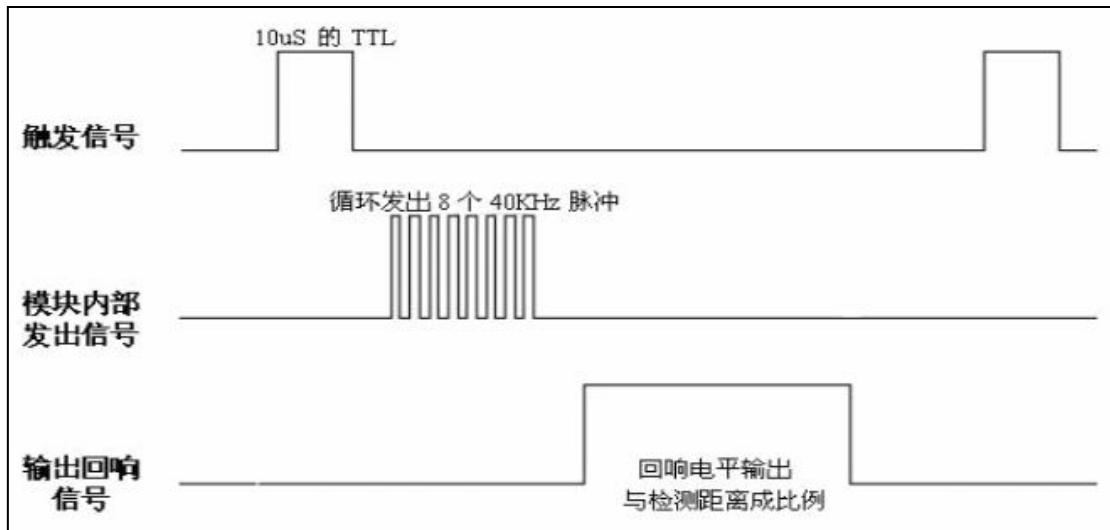


图 4.7.2 SR04 操作时序

许多大厂商生产的外设都配有详细的使用手册。关于 SR04 的中文资料我们已经收录在光盘资料 1. 实验板&硬件资料中。如何从一份使用手册中快速掌握一种外设的使用方法，手册中的时序图就是最好的途径。通过时序图，我们可以直观地看出操作这个外设时哪个引脚该什么时候置高、该什么时候置低、什么时候有信号反馈等等一系列重要信息。

我们看看 SR04 的时序图。看时序图遵循从左往右的原则。首先单片机给触发信号端 1 个 10us 的高脉冲，然后模块发出 8 个 40KHz 的超声波脉冲，当模块接收到回响的超声波信号，就在回响信号端输出 1 个脉宽与回响时间成正比例的高脉冲。

如果我们要测距，就给超声波模触发信号端 Trig 一个 10us 的高脉冲，然后想办法计算回响信号的脉冲。

7.2 软件实现

在工程中新建文件 ultrasonic.c，写入下列程序

```
#include "stc12.h"
#include "intrins.h"
#include "ultrasonic.h"
#include "digitron.h"

uint Time;
float Distance;
uint Overflow_Flag;
```

```
//超声波初始化函数
void Ultra_Init(void)
{
    TMOD = 0x01;

    TH0 = 0;
    TL0 = 0;

    TR0 = 1;
    ET0 = 1;
    EA = 1;
}

//定时器0中断服务函数
void Time0_Handler() interrupt 1
{
    Overflow_Flag = 1;
}

//微妙延时函数
//具体延时时间需要debug来决定
void Delay_us(uint z)
{
    while(z--);
}

//超声波启动函数
void Ultra_Start(void)
{
    trig = 1;
    Delay_us(6);
    trig = 0;
}

//超声波数据处理函数
void Ultra_Conut(void)
{
    Time = TH0*256+TL0;
    TH0 = 0;
    TL0 = 0;
    Distance = Time * 0.017;
}

//超声波显示函数
void Ultra_Show(void)
{
    if(Overflow_Flag == 1)
    {
        Overflow_Flag = 0;
```

```

        Digi_Num(9999);
    }
    else
        Digi_Num(Distance);
}

//超声波工作函数
void Ultra_Work(void)
{
    Ultra_Start();

    while(!echo);
    TR0 = 1;
    while(echo);
    TR0 = 0;

    Ultra_Conut();

    Ultra_Show();
}

```

新建文件 ultrasonic.h , 写入下列程序

```

#ifndef _ULTRASONIC_H_
#define _ULTRASONIC_H_

#define uchar unsigned char
#define uint unsigned int

sbit echo = P3^3;
sbit trig = P3^4;
void Ultra_Init(void);
void Delay_us(uint z);
void Ultra_Start(void);
void Conut(void);
void Ultra_Show(void);
void Ultra_Work(void);

#endif

```

新建文件 main.c , 写入下列程序

```

#include "stc12.h"
#include "intrins.h"
#include "ultrasonic.h"
#include "digitron.h"

void main()
{
    Digi_Init();
    Ultra_Init();
}

```

```
while(1)
{
    Ultra_Work();
}
```

完成超声波测距，我们要完成三个工作：一、在测距开始时，给 Trig 脚一个 10us 脉冲，触发超声波模块工作；二、当 Echo 有高脉冲返回时，测量脉冲宽度；三、通过测得的脉冲宽度，计算出距离，并在数码管显示出来。

首先我们先看看如何触发超声波模块。在喇叭一节中我们已经提到，STC12 并不能产生精准的 1us 延时。所以我们用 Delay_us() 函数替代。Delay_us() 函数只有一个 while 循环。每执行一次循环，参数 z 就减 1。循环每执行一次都是微秒级的，但循环执行的总时间和 z 不成正比，所以我们要延时 10us 的话，z 要通过调试一次次地试出来。最后我们调试得当 z 等于 6 时，循环所用的总时间约等于 10us。

然后第二步，测量 Echo 返回脉冲宽度才是重中之重。脉冲什么时候返回，取决于测量距离的远近，我们是无法预知的。那有没有方法让单片机能自己检测脉冲的到来呢？机灵的你，有没有回忆起我们曾经通过中断法检测按键？让返回脉冲作为中断源，单片机等待外部中断的到来，这确实是一个可行的方法。但学过外部中断中断的我们都应该知道，51 单片机的外部中断的触发方式只有两种：下降沿触发和低电平触发。换言之，通过外部中断，我们只能检测脉冲的结束，但不能检测脉冲的到来。你连脉冲什么时候来都不知道，何谈检测脉冲宽度？

```
//超声波工作函数
void Ultra_Work(void)
{
    Ultra_Start();

    while(!echo);
    TR0 = 1;
    while(echo);
    TR0 = 0;

    Ultra_Conut();

    Ultra_Show();
}
```

我们看一下超声波测距函数 Ultra_Work()。在每次测距前，我们都先给模块发送一个触发信号。这个步骤我们已经封装好一个函数 Ultra_Start()。然后我们就用一个 while 循环，循环的判断条件为 (!echo)。如果返回脉冲还没来，Echo 将一直为低，判

断成立，程序会卡在这个死循环里面。当有返回脉冲时，Echo 会变为高，此时跳出 while 循环，启动定时器 0。因为我们还有等待脉冲的结束，所以我们用到了第二个 while 循环，这次判断条件变为(!echo)。当脉冲持续时，Echo 会持续为高，此时程序就能卡在这个循环内，等待脉冲的过去。当脉冲过去后，我们就立刻关闭定时器 0。

这里，我们借助定时器 0 模式 1 来计量返回脉冲的脉宽。定时器 0 模式 1 下，计数器如果从零开始计数，能计 65536 次，约能计时 65.5ms。常温下声速为 340m/s，即 0.34m/ms。所以能检测的最大距离约为 22m，远大于本实验超声波模块 HS0038 的最大测量距离 1 米。

```
//超声波数据处理函数
void Ultra_Conut(void)
{
    Time = TH0*256+TL0;
    TH0 = 0;
    TL0 = 0;
    Distance = Time * 0.017;
}
```

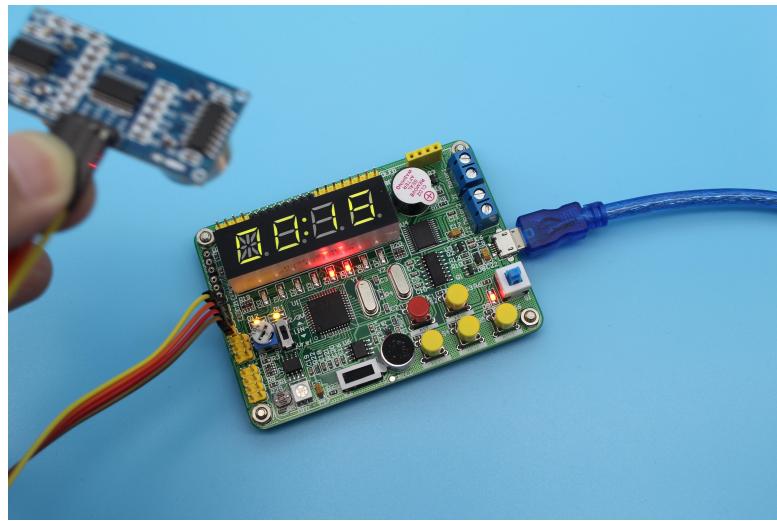
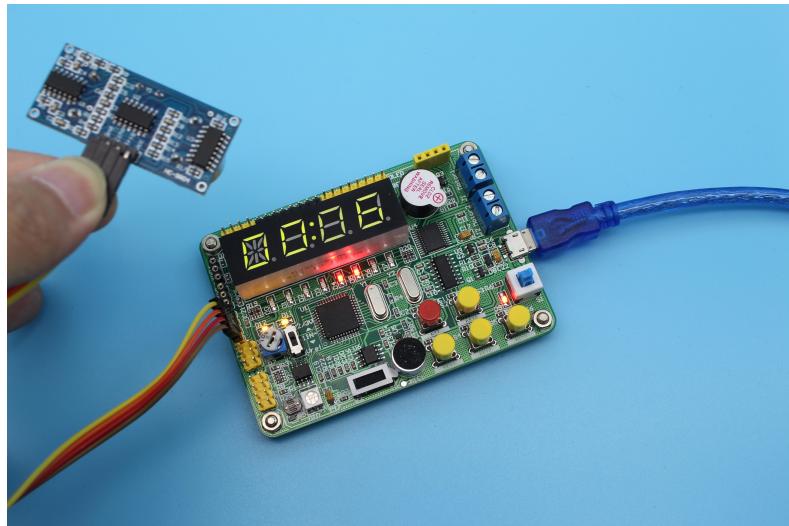
脉冲结束，关闭定时器 0 后，我们就要计算脉宽，从而算出所测得的距离。测量得到的脉宽数据存放定时器 0 的计数器 TH0 和 TL0 中。但要算出总时间，并不能简单地将两者相加！这和我们配置定时器时将置处理后再送给 TH0 和 TL0 一样。数据处理函数 Ultra_Conut() 中，我们将这一过程反过来，现将 TL0 乘以 256，在加上 TH0，就得到计数器总共计数了多少次。因为计数器每 1us 加一，所以得到的值就是超声波来回总共所用的时间，单位为 us。

常温下，声速是 340m/s，转换为 cm 和 us 为 0.034cm/us。但我们别忘了，我们测得的时间是超声波来回所化的总时间，所以我们还要除以二。简化后，就是将测得时间乘以 0.017cm/us，就得出真实的距离。

得到超声波来回所用的总时间，我们就通过速度、时间和距离的公式，把距离算出来。最后将得到的距离显示在数码管上。

7.3 下载验证

将工程编译生成的 hex 文件烧录进实验板中。正确连接超声波模块，移动模块，观察数码管上距离显示值的变化。



7.4 总结思考

超声波其实说白了，也是很简单的。但是如果要做精细，那就不简单了。

首先，常温下，声速为 340m/s 。但声速并不是常数，它和温度与大气压以及传播媒介都有关系。考虑到实际应用中大气压和传播媒介变化不大，因此时刻变化的温度就成了影响我们测距精度的重要因素了。

那我们怎样知道环境中的温度呢？我们在下一节中，将利用温度传感器 DS18B20 来检测环境中的温度。将超声波模块和温度传感器结合，我们就可以修正温度对声速的影响，得到高精度的距离值。

八、温度传感器——感受世间冷暖

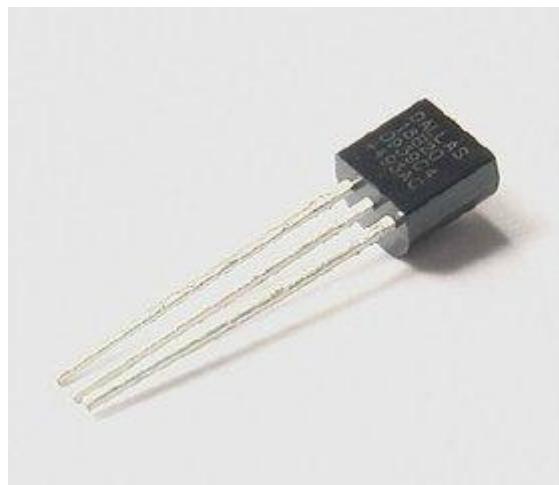


图 4.8.1 直插式温度传感器 DS18B20

在上节的总结中，我们提到，可以用温度传感器测量环境温度，从而修正温度对声速的影响，从而实现高精度的超声波测距。其实，生活中空调、冰箱、电子温度计等等，这些我们身边的产品都需要对外界温度进行实时的监控。而温度检测也是很多智能家居的重要创新点，温度适宜的坐垫、有温度提醒的奶瓶等等。在本节中，我们就走近直插式温度传感器 DS18B20，探讨温度监控的奥秘。

8.1 硬件准备

直插式 DS18B20（以下简称 DS18）有三根引脚，分别是 1 接地引脚、2 数据输出输入引脚、3 电源引脚。

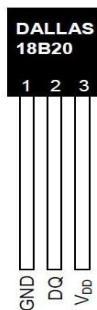


图 4.8.2 DS18B20 引脚图 1



图 4.8.3 DS18B20 引脚图 2

在连接上，DS18 只需将 DQ 脚与单片机引脚相连，完成对 DS18 内数据的读取与写入。在实际应用中，例如要检测一间房间的实时温度，我们往往需要多个 DS18 来检测多点的温度，以防数据过少影响结果的准确性。但如果一下子用十来个 DS18，那岂不是很占用单片机的引脚资源吗？其实，每个 DS18 都有 1 个全球唯一的 64 位序列号，我们可以把用到的 DS18 的 DQ 引脚都连接到单片机同一根引脚上，单片机要读取某个 DS18 内的数据前，先查询该 DS18 的地址，再读取数据。另外，不管那种连接，DQ 引脚都要连接 1 个弱上拉电阻，因此常态下 DQ 引脚为高电平。

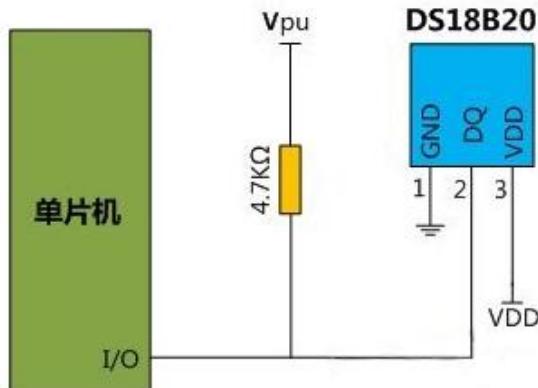


图 4.8.4 单个 DS18B20 与单片机相连

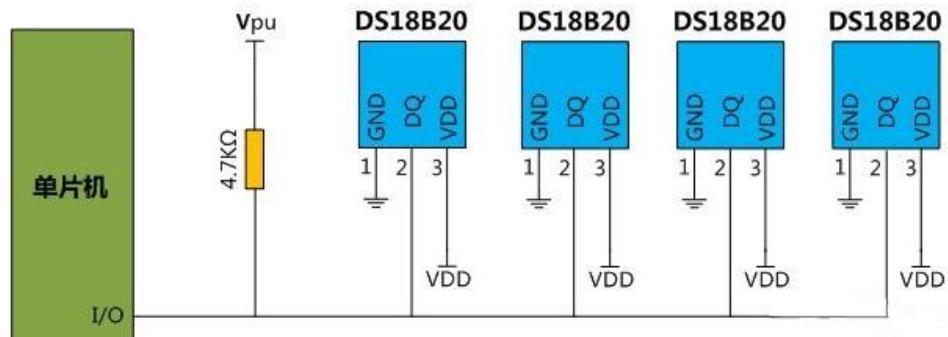


图 4.8.5 多个 DS18B20 与单片机相连

为了操作 DS18，我们要通过相关指令来操作器件中的寄存器。因此在了解 DS18 的操作之前，我们先要了解一下 DS18 的内部结构和操作它时用到的指令。首先我们先看看 DS18 的内部结构以及它相关寄存器：

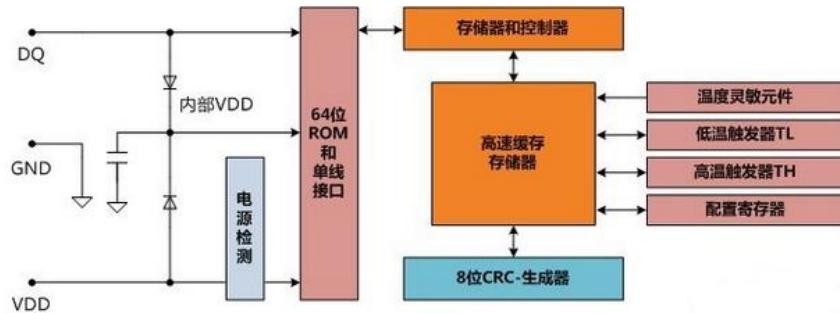


图 4.8.6 DS18B20 内部结构

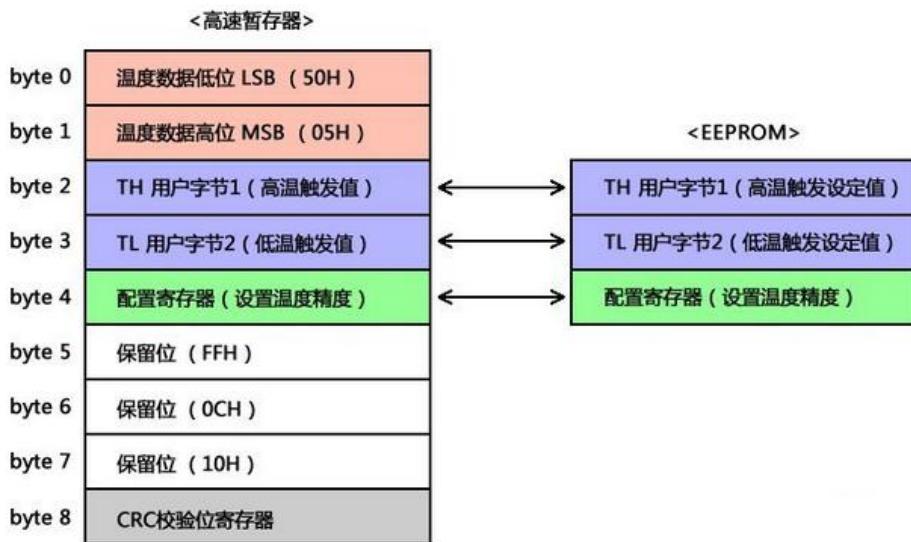


图 4.8.7 DS18B20 内部相关寄存器

8.2 知识点讲解

其实，DS18B20 的原理，就是我们开头时讲到的 ADC 功能。但不同的是，这次控制 ADC 的寄存器不在单片机内部，而在外设当中。所以，我们要用单片机和 DS18B20 进行通讯，通过单片机控制 DS18B20 进行 AD 转换，并将 DS18B20 得到的温度值读回来。

我们通过 DQ 引脚给 DS18 发送一个温度测量和 AD 转换指令后，温度灵敏元件测得的温度数据会以两个字节的形式存入高速暂存器字节 1、2 中的温度寄存器中。在温度转换过程中，DQ 引脚返回，结束后返回 0，我们可以藉此来判断 DS18 是否完成温度转换。DS18B20 读取的值转换为十进制后就是摄氏温度，可直接显示，无需修正。

另外，DS18 还有自动报警功能，我们可以给高速暂存器中的 TH 高温触发设定值和 TL 低温触发设定值填入需要预警的温度，然后 DS18 会把这两字节的值存入 EEPROM 的 TH 和 TL 中保存起来。当 DS18 完成一次温度转换后，会将温度数据和高温、低温触发值进行比较。若温度值高于 TH 或低于 TL，那 DS18 会将内部 1 个报警标示位置位。单片机可以给 DQ 引脚发送 1 个警报搜索指令，查看是否有警报。

8.3 软件实现

新建工程，命名为“温度传感器”。在 HARDWARE 文件中新建文件夹 TEMP，文件下新建 temp.c 和 temp.h。从之前的工程中将文件夹 DIGITRON 复制到 HARDWARE 文件夹下。在工程所在的 USER 中新建文件 main.c。

在 temp.c 中写入以下程序：

```
#include "stc12.h"
#include "temp.h"
#include "digitron.h"
#include "intrins.h"

//微秒延时函数
//具体延时实验需要 debug 来测量
void Delay_us(uint n)
{
    while (n--)
    {
        _nop_();
        _nop_();
    }
}

//温度传感器复位函数
void Temp_Init(void)
{
    uint x = 1;

    while(x)
    {
        DA = 0;//拉低数据线
        Delay_us(480);//等待 480us
        DA = 1;//释放数据线
        Delay_us(60);//等待 60us
        x = DA;//检查 DS18B20 是否有响应
    }
}
```

```
        Delay_us(420);//等待 DS8B20 释放数据线
    }

//给 DB18B20 写一节数据
void Temp_Write(uchar dat)
{
    uchar i;

    for(i = 0;i < 8;i++)
    {
        //写数据时序
        DA = 0;
        Delay_us(1);
        DA = 1;

        //发送数据
        dat >>= 1;
        DA = CY;

        Delay_us(20);

        DA = 1;
        Delay_us(1);
    }
}

//从 DB18B20 读一节数据
uchar Temp_Read(void)
{
    uchar i;
    uchar dat = 0;

    for (i = 0;i < 8;i++)//读取一节 8 位数据
    {
        dat >>= 1;

        //读数据时序
        DA = 0;
        Delay_us(1);
        DA = 1;
        Delay_us(1);

        //读取数据
        if(DA)
            dat |= 0x80;

        //等待读取结束
    }
}
```

```

        Delay_us(60);
    }

    return dat;
}

//温度显示函数
void Temp_Show(void)
{
    uint temp;
    uchar th,tl;

    //开始 AD 转换指令
    Temp_Init();
    Temp_Write(0XCC);
    Temp_Write(0X44);

    while(!DA);//等待转换完成

    //读取暂存存储器数据指令
    Temp_Init();
    Temp_Write(0XCC);
    Temp_Write(0XBE);

    tl = Temp_Read();//读取温度低字节
    th = Temp_Read();//读取温度高字节

    //合成高低温度数据，取低字节高 4 位和高字节低三位，得到温度的整数值
    tl = (tl & 0XF0) >> 4;
    temp = th;
    temp <<= 4;
    temp |= tl;

    Digi_Num(temp);
}

```

在 temp.h 中写入以下程序：

```

#ifndef _DS18B20_H_
#define _DS18B20_H_

#define uint unsigned int
#define uchar unsigned char

sbit DA = P1^1;

void Temp_Init(void);
void Temp_Write(uchar dat);

```

```
uchar Temp_Read(void);
void Temp_Show(void);
void Delay_us(uint n);

#endif
```

在 main.c 中写入以下程序：

```
#include "stc12.h"
#include "temp.h"
#include "digitron.h"

void main()
{
    Digi_Init();
    Temp_Init();

    while(1)
    {
        Temp_Show();
    }
}
```

DS18B20 的基本操作有三个：复位、写数据和读数据。这三步是 DS18B20 转换、读取温度的基础步骤。首先我们看看复位函数：

```
//温度传感器复位函数
void Temp_Init(void)
{
    uint x = 1;

    while(x)
    {
        DA = 0;//拉低数据线
        Delay_us(480);//等待 480us
        DA = 1;//释放数据线
        Delay_us(60);//等待 60us
        x = DA;//检查 DS18B20 是否有响应
        Delay_us(420);//等待 DS18B20 释放数据线
    }

    x = 0;
}
```

复位是一个检测 DS18B20 是否有响应的信号，同时也是告诉开始 DS18B20 工作的信号。在每次给 DS18B20 写数据时，都要先给 DS18B20 发送一个复位信号。DS18B20 的复位时序如下图：

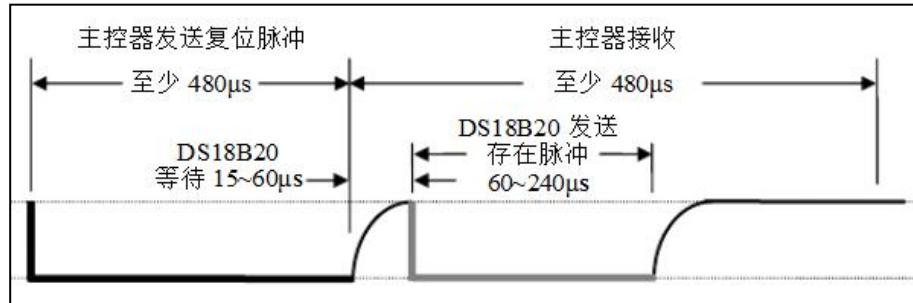


图 4.8.8 DS18B20 复位时序

我们先将信号脚拉低至少 480us。如果 DS18B20 能响应的话，在 60us 的短暂延时后，DS18B20 的信号脚将会被拉低。我们用 while 循环来循环等待 DS18B20 响应的过程，循环条件为 DS18B20 信号脚的电平。这样，直到 DS18B20 有响应，信号脚被拉低，才会跳出循环。

然后我们看看写数据函数：

```
//给 DB18B20 写一节数据
void Temp_Write(uchar dat)
{
    uchar i;

    for(i = 0;i < 8;i++)
    {
        //写数据时序
        DA = 0;
        Delay_us(1);
        DA = 1;

        //发送数据
        dat >>= 1;
        DA = CY;

        Delay_us(20);

        DA = 1;
        Delay_us(1);
    }
}
```

每次写数据给 DS18B20 前，我们要将信号脚拉低约 4us，然后给 DS18B20 送一位数据。送完数据后要拉高信号脚，再作 4us 延时，为下一次写数据做准备。这样用 for 循环循环 8 次，将一节数据完整地发送给 DS18B20。在发送数据给 DS18B20 是，我们用到了 CY 位。CY 位是程序状态字寄存器 PSW 的第 7 位，是进位、错位标志位。

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	CY	AC	F0	RS1	RS0	OV	-	P

表 4.8.1 PSW 相关位定义

CY：进位、错位标志位。有进位、错位时，CY = 1，否则 CY = 0。

我们发送要发送数据 dat 的每一位前，我们先将它右移一位。如果本来 dat 最右一位为 1，dat 右移一位后就发生进位，CY 会被置 1；反过来，如果本来 dat 最右一位为 0，dat 右移后就不会发生进位，CY 将为 0。所以在每次 dat 右移后，CY 位的电平高低就等于 dat 右移前最右一位的电平高低。于是我们就将 CY 直接赋给信号脚 DA，依次循环 8 次，就将发送数据 dat 写进 DS18B20 了。所以，以后需要依次处理一节数据的每个位时，可以灵活运用 CY 位，简化程序逻辑。

最后我们看看读数据函数：

```
//从 DB18B20 读一节数据
uchar Temp_Read(void)
{
    uchar i;
    uchar dat = 0;

    for (i = 0;i < 8;i++)//读取一节 8 位数据
    {
        dat >>= 1;

        //读数据时序
        DA = 0;
        Delay_us(1);
        DA = 1;
        Delay_us(1);

        //读取数据
        if(DA)
            dat |= 0x80;

        //等待读取结束
        Delay_us(60);
    }
}
```

```
    }

    return dat;
}
```

读数据是写数据的反过程。在读数据前，同样先将信号脚拉低 4us。然后判断信号脚返回数据的高低。我们用 for 循环，通过或的形式将读取到的一位数据依次从高到低填入变量 dat 中，将一节数据完整存放在变量 dat 中。

以上，就是 DS18B20 的基本操作。有了复位、写数据和读数据后，我们就能控制 DS18B20 来进行 AD 转换以及读取里面存放的温度数据了。

```
//温度显示函数
void Temp_Show(void)
{
    uint temp;
    uchar th,tl;

    //开始 AD 转换指令
    Temp_Init();
    Temp_Write(0XCC);
    Temp_Write(0X44);

    while(!DA);//等待转换完成

    //读取暂存存储器数据指令
    Temp_Init();
    Temp_Write(0XCC);
    Temp_Write(0XBE);

    tl = Temp_Read();//读取温度低字节
    th = Temp_Read();//读取温度高字节

    //合成高低温度数据，取低字节高 4 位和高字节低三位，得到温度的整数值
    tl = (tl & 0XF0) >> 4;
    temp = th;
    temp <<= 4;
    temp |= tl;

    Digi_Num(temp);
}
```

DS18B20 有专门的指令集。指令集在 DS18B20 有详细介绍，我们这里整理出需要的指令：

指令类型	指令	功能	详细描述
ROM 指令	0XF0	搜索 ROM	当系统初始化时，单片机通过此指令多次循环搜索 ROM 编码，以确认所有 DS18
	0X33	读取 ROM	当控制线上只有一只 DS18 时才会使用此指令，允许单片机直接读取 DS18 的序列码
	0X55	匹配 ROM	匹配 ROM 指令，使单片机在连接多只 DS18 的控制线上定位一只 DS18
	0XCC	忽略 ROM	忽略 ROM 指令，此指令使单片机不必提供 64 位 ROM 编码就使用功能指令
	0XEC	报警搜索	当控制线上有 DS18 满足报警条件时，该 DS18 将响应此指令
功能指令	0X44	温度转换	此指令用来控制 DS18 启动一次温度转换，生成的温度数据以 2 字节的形式存储在高速暂存器中
	0X4E	写暂存器	此指令向 DS18 的暂存器写入数据，开始位置在暂存器第 2 字节（TH 寄存器），以最低有效位开始传送
	0XBE	读暂存器	此指令用来读取 DS18 暂存器数据，读取将从字节 0 开始，直到第 9 字节（CRC 校验位）读完
	0X48	拷贝暂存器	此指令将高速暂存器中 TH、TL 以及配置寄存器的数据拷贝到 EEPROM 中得以保存
	0XB8	召回 EEPROM	将 TH、TL 以及配置寄存器中的数据从 EEPROM 拷贝到高速暂存器

表 4.8.2 **DS18B20** 相关指令集

每次写数据前，我们都给 DS18B20 发送一个指令 0XCC，这样就能直接控制 DS18B20。我们先给 DS18B20 温度转换指令 0X44，DS18B20 就会进行温度 AD 转换。转换过程中，信号脚会一直被拉低，我们就用 while 循环，等待 AD 转换结束。

AD 转换后得到的数据会存放在 DS18B20 内部温度暂存器 TH 和 TL 中：

位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

表 4.8.3 DS18B20 温度暂存器 TL

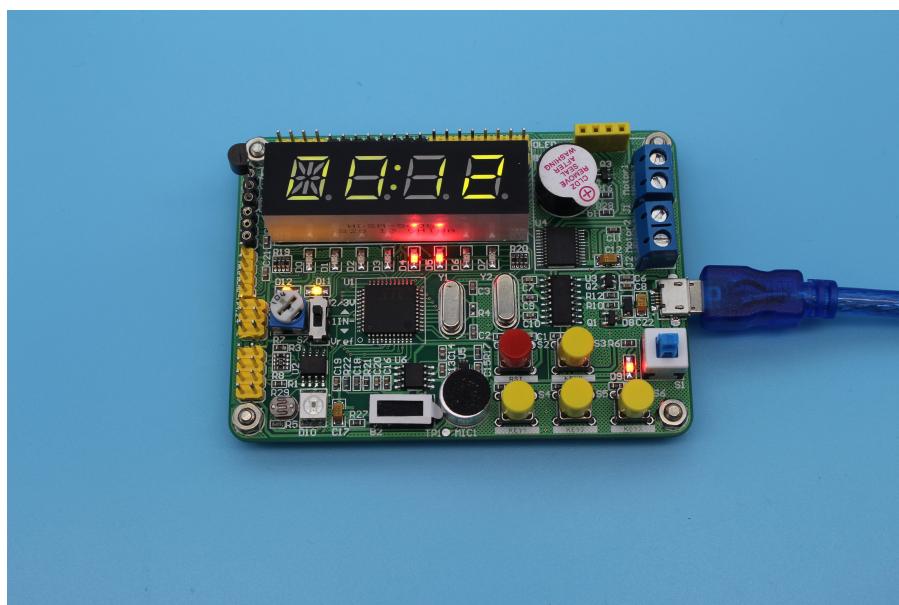
位	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
位定义	-	-	-	-	-	2^6	2^5	2^4

表 4.8.4 DS18B20 温度暂存器 TH

例如，如果读取数据 TH : 00000111、TL : 11010000，其中整数部分 01111101 (其中 0111 在 TH 中，1101 在 TL 中) 对应十进制为 125，即温度为 125°C。DS18 测量温度的范围是 -55°C 到 +125°C，会有 $\pm 0.5^\circ\text{C}$ 的误差。

在 DS18B20 完成温度转换后，我们发送读暂存器指令 0XB0。在下两次读取 DS18B20 数据时，就能分别读取 TL 和 TH 中的值。我们只取整数做处理，所以取 TH 的低 4 位和 TL 的高 4 位，进行位运算后合并成一个 8 位 2 进制数据。这个数据的十进制，就是环境温度的值。

8.4 下载验证



8.5 总结思考

这是第一次，我们使用单片机以外的模块里的寄存器。

我们在准备篇就说过，在数字电路中，我们将用于存储一组数据的存储电路叫做寄存器。所以寄存器并不是单片机的“专利”。在很多需要数据处理的模块中，往往都会有寄存器。但是和单片机一样，这些外设中的寄存器在数据手册中都会有详细的介绍和使用说明，只要我们认真阅读，就能掌握它们的使用。

九、RGBLED——全彩精灵

前面我们用到的 LED 都是单色的，那有没有 LED 只需一盏就能发出各种各样的颜色呢？今天我们就看看一款能呈现 16777216 种（我没打错！）颜色光的神奇精灵——WS2812。

9.1 硬件准备

WS2812 是一款 RGBLED，它就是我们实验板上一个白色的小家伙：



图 4.9.1 实验板上的 RGBLED

仔细观察 WS2812 你会发现，它里面其实有三盏 LED：红色 LED、绿色 LED 和蓝色 LED。但和控制普通 LED 不同，WS2812 内通过一块控制芯片同时控制三色 LED。通过给 WS2812 发送数据，它就能控制三盏 LED 的亮度。红绿蓝每种颜色 LED 的亮度由一个 8 位二进制数据决定。通过给控制芯片发送数据可以独立控制三盏 LED 的亮度。

我们再看看 RGBLED 模块的电路图：

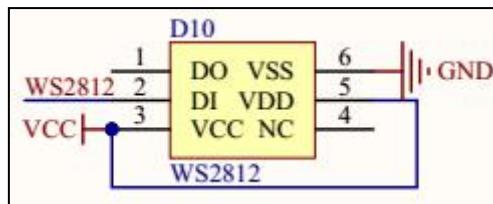


图 4.9.2 WS2812 电路连接

一般 WS2812 有 4 引脚和 6 引脚两种封装，这里我们用的是 6 引脚封装。这 6 个引脚我们重点关注 DI 和 DO 两个引脚。DI 是 Data In 引脚，DO 是 Data Out 引脚。DI 顾名思义是用来给 WS2812 内控制芯片发送数据的。那 DO 是干什么用的呢？其实你想看，WS2812 只是一个小不点，实际应用中肯定不止用一盏。DO 是信号输出端，它

可以接到下一盏 WS2812 的 IN，这样我们就能将多个 WS2812 连成一条灯带。这样我们只用一个 IO 口就能控制一条 RGBLED 灯带了！

如今我们只用到一盏 RGBLED，所以 ON 悬空。而 IN 接单片机 P3.5，作为 WS2812 的控制端。

9.2 知识点讲解

RGB 色彩模式是工业界的一种颜色标准。R 是红色，G 是绿色，B 是蓝色。通过控制红绿蓝三种颜色的强度变化以及它们的叠加得到各式各样的颜色。每种颜色的强度范围为 0~255，所以 RGB 可以呈现的颜色种类达 256^3 种即 16777216 种！

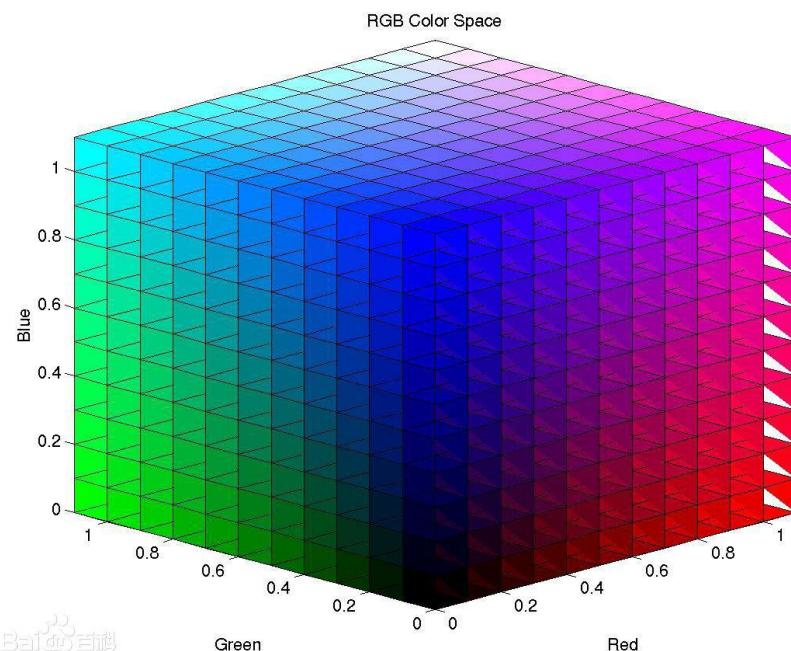


图 4.9.3 RGB 色谱

WS2812 就是通过控制三盏 LED 的光强度和叠加来呈现各种各样的光的。三盏 LED 的光强由 WS2812 内的控制芯片控制。每种光的强度是 0~255，刚好是 8 位二进制。发送的值越大，相应 LED 的光强越大。所以我们要给 IN 引脚发送 8 位二进制来实现三盏 LED 光强的控制。

但是要怎样控制三盏 LED 的光强呢？

首先，三组 8 位二进制数据总共 32 位数据可不是任意顺序发送给 WS2812 的。我们先将控制绿色 LED 光强的 8 位二进制数据从高位到地位依次发送给 WS2812，再发送红色的，最后发送蓝色的。

其次，这些控制数据的格式也是有要求的。如果发送数据 0，先给 700ns 高电平再给 90ns 的低电平。而如果发送数据 1，就先给 180ns 高电平后再给 360ns 低电平。

所以我们只要按照数据格式将 32 位数据发送给 WS2812，它就能按照我们的所想显示颜色了。但真的有那么简单吗？

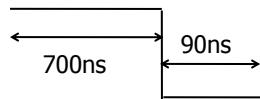
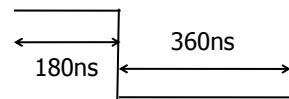


图 4.9.4 数据 0 格式



4.9.5 数据 1 格式

9.3 软件实现

建立工程，命名为“RGBLED”。在 HARDWARE 文件中新建文件夹 RGBLED，文件下新建 rgbled.c 和 rgbled.h。在工程所在的 USER 中新建文件 main.c。

在 rgbled.c 中写入以下程序：

```
#include "stc12.h"
#include "intrins.h"
#include "rgbled.h"

uchar bdata LED_DAT;
sbit bit0 = LED_DAT^0;
sbit bit1 = LED_DAT^1;
sbit bit2 = LED_DAT^2;
sbit bit3 = LED_DAT^3;
sbit bit4 = LED_DAT^4;
sbit bit5 = LED_DAT^5;
sbit bit6 = LED_DAT^6;
sbit bit7 = LED_DAT^7;

uchar RGB_DATA[3] = {0x00,0x00,0X00};

//毫秒延时函数，误差 2us
void Delay_ms(uint z)
{
    uint x,y;
```

```
    for(x = z;x > 0;x--)
        for(y = 921;y > 0;y--);
}

//RGBLED 复位函数
void RGBLED_Rest(void)
{
    uint x;
    for(x = 50;x > 0;x--);
}

//RGBLED 数据发送函数
void Send_Data(uchar *dat)
{
    LED_DAT = *dat++;
    if(bit7 == 1)
        CODE1
    else
        CODE0
    if(bit6 == 1)
        CODE1
    else
        CODE0
    if(bit5 == 1)
        CODE1
    else
        CODE0
    if(bit4 == 1)
        CODE1
    else
        CODE0
    if(bit3 == 1)
        CODE1
    else
        CODE0
    if(bit2 == 1)
        CODE1
    else
        CODE0;
    if(bit1 == 1)
        CODE1
    else
        CODE0
    if(bit0 == 1)
        CODE1
    else
        CODE0

    LED_DAT = *dat++;
    if(bit7 == 1)
        CODE1
```

```
else
    CODE0
if(bit6 == 1)
    CODE1
else
    CODE0
if(bit5 == 1)
    CODE1
else
    CODE0;
if(bit4 == 1)
    CODE1
else
    CODE0;
if(bit3 == 1)
    CODE1
else
    CODE0
if(bit2 == 1)
    CODE1
else
    CODE0;
if(bit1 == 1)
    CODE1
else
    CODE0;
if(bit0 == 1)
    CODE1
else
    CODE0

LED_DAT = *dat;
if(bit7 == 1)
    CODE1
else
    CODE0
if(bit6 == 1)
    CODE1
else
    CODE0
if(bit5 == 1)
    CODE1
else
    CODE0
if(bit4 == 1)
    CODE1
else
    CODE0
if(bit3 == 1)
    CODE1
else
    CODE0
```

```

        if(bit2 == 1)
            CODE1
        else
            CODE0
        if(bit1 == 1)
            CODE1
        else
            CODE0
        if(bit0 == 1)
            CODE1
        else
            CODE0
    }

void RGBLED_Init(void)
{
    P4SW = 0X40;
}

//RGBLED 设置函数
//R、 G、 B 分别为红色、 绿色和蓝色的颜色值， 取值范围为 0~255
void RGBLED_Set(uchar R,uchar G,uchar B)
{
    RGB_DATA[0] = G;
    RGB_DATA[1] = R;
    RGB_DATA[2] = B;

    RGBLED_Rest();
    Send_Data(RGB_DATA);
}

//RGBLED 演示
void RGBLED_Test(void)
{
    uint r = 100,g,b;

    for(g = 0;g < 100;g++)
    {
        RGBLED_Set(r,g,b);

        Delay_ms(5);
    }

    for(r = 100;r > 0;r--)
    {
        RGBLED_Set(r,g,b);

        Delay_ms(5);
    }

    for(b = 0;b < 100;b++)

```

```

{
    RGBLED_Set(r,g,b);

    Delay_ms(5);
}

for(g = 100;g > 0;g--)
{
    RGBLED_Set(r,g,b);

    Delay_ms(5);
}

for(r = 0;r < 100;r++)
{
    RGBLED_Set(r,g,b);

    Delay_ms(5);
}

for(b = 100;b > 0;b--)
{
    RGBLED_Set(r,g,b);

    Delay_ms(5);
}
}

```

在 rgbled.h 中写入以下程序：

```

#ifndef _RGBLED_H_
#define _RGBLED_H_

#define uint unsigned int
#define uchar unsigned char

//发送数据 1 时序
#define CODE1{DATA = 1;\n    _nop_();_nop_();_nop_();_nop_();_nop_();\n    _nop_();_nop_();_nop_();\n    DATA = 0;\n    _nop_();\n}
}

//发送数据 0 时序
#define CODE0{DATA = 1;\n    _nop_();_nop_();\n    DATA = 0;\n    _nop_();_nop_();_nop_();_nop_();\n}
}

```

```

sbit DATA = P3^5;

void Delay_ms(uint z);
void RGBLED_Rest(void);
void Send_Data(uchar *dat);
void RGBLED_Init(void);
void RGBLED_Set(uchar R, uchar G, uchar B);
void RGBLED_Test(void);

#endif

```

在 main.c 中写入以下程序：

```

#include "stc12.h"
#include "rgbled.h"

void main()
{
    RGBLED_Init();

    while(1)
    {
        RGBLED_Test();
    }
}

```

首先我们先看看 rgbled.c。在 rgbled.c 里面，我们用到了一个新的关键字：bdata。用 bdata 声明的变量它每个位都是可以单独位寻址的。因此，我们又用关键字 sbit 定义好变量 LED_DAT 的 8 个位。

接着是 RBGLED 复位函数 RGBLED_Rest。在给 WS2812 发送数据前我要先将信号线拉低 50us。

然后是数据发送函数 Send_Data。数据发送函数接受一个数组变量。这个数组变量是事先声明好的数组 RGB_DATA，它用来存放 RGB 值的。函数的功能是将 3 个 8 位二进制数由高位至低位转换为控制 WS2812 的信号。

```

//发送数据 1 时序
#define CODE1{DATA = 1; \
    _nop_();_nop_();_nop_();_nop_();_nop_(); \
    _nop_();_nop_();_nop_(); \
    DATA = 0; \
    _nop_(); \
}

//发送数据 0 时序

```

```
#define CODE0{DATA = 1;\  
    _nop_();_nop_();\  
    DATA = 0;\  
    _nop_();_nop_();_nop_();_nop_();\  
}
```

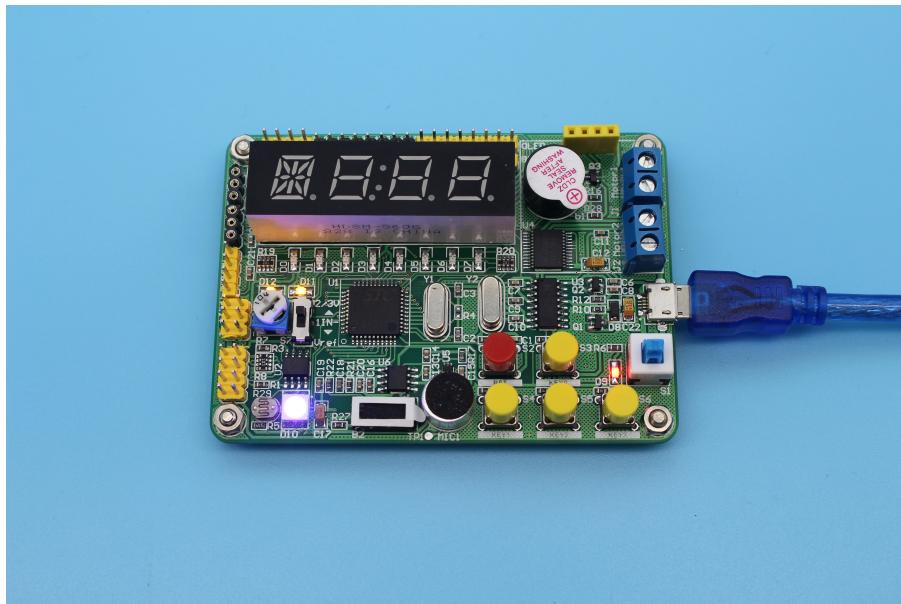
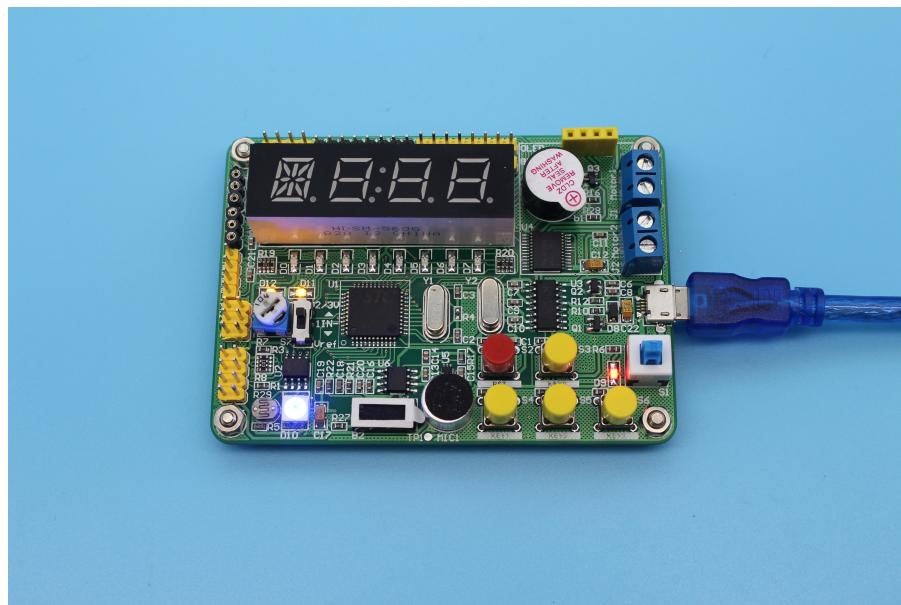
函数 Send_Data 中我们一次判断数组 RGB_DATA 中每位是 1 还是 0。是 1 就执行 CODE1，0 就执行 CODE0。CODE0 和 CODE1 我们都在 rgbled.h 中宏定义好了。我们看到#define 不仅可以对变量或关键字进行宏定义，而且能对一段函数进行宏定义。不过在#define 中如果要换行，行末要加上 “\” ，否则编译器会报错。

```
//RGBLED 设置函数  
//R、G、B 分别为红色、绿色和蓝色的颜色值，取值范围为 0~255  
void RGBLED_Set(uchar R,uchar G,uchar B)  
{  
    RGB_DATA[0] = G;  
    RGB_DATA[1] = R;  
    RGB_DATA[2] = B;  
  
    RGBLED_Rest();  
    Send_Data(RGB_DATA);  
}
```

回到 rgbled.c。如果我们要控制 RGBLED，就要给数组 RGB_DATA 填入三个值。为了能直观地通过赋值来直接控制 RGBLED，我们封装了一个函数 RGBLED_Set()。RGBLED_Set()接受 3 个变量，分别是 R、G 和 B 三个值。函数中将三个接受到的变量赋给数组 RGB_DATA，然后再调用函数 Send_Data()把数据发送出去。所以我们可以值 main.c 中直接使用函数 RGBLED_Set()直观方便地控制 RGBLED。

另外我们还封装了一个 RGBLED_Test()函数，它能循环显示 RGBLED 能显示的所有颜色。

9.4 下载验证



9.5 总结思考

其实，用 stc12 控制 WS2812 是很勉强的。WS2812B 的时序是纳秒级的，而且非常严谨。但 stc12 并没有纳秒延时函数，最短的一条指令用时也要 90n 秒。我们用 bdata 声明变量来直接判断变量的某一位是 1 或 0，而不用 for 语句循环检测每一个位是因为执行 for 判断一次都是毫秒级的。我们宏定义发送数据 1 和 0 的时序而不是将其封装成函数。宏定义执行起来比函数快，这样就可以提高程序运行速度，减少对时序的影响。

响。所以说 stc12 控制 WS2812 是很勉强的，这是 stc12 本身性能决定的。如果想更简单地控制 WS2812，可以选择能实现精准纳秒延时的单片机如：STM32 或 Arduino 等。

另外还有一点要注意的是，WS2812 一般用于大型显示屏。你在一些旅游景区中的大幅 LED 屏幕中找到它的身影。因为用于大型显示屏，所以 WS2812 的亮度非常高。我们在测试程序中只是用到一半的亮度亦然觉得有点刺眼。所以我们要避免在 WS2812 亮度最大时直视它，保护好双眼。

十、红外遥控——决策千里之外

电视机遥控、空调遥控……生活中都得益于各种各样的遥控，我们才能像军师一样坐在沙发上就能控制家里各种的电子产品——这可谓决策千里之外。虽然大家都知道这些遥控都是通过红外遥控实现的，但你又知道是怎样实现红外遥控的吗？今天我们就一起探秘红外遥控，通过 HS0038 来遥控 stc12！

10.1 硬件准备

今天我们要介绍 HS0038。HS0038 是一款红外接收探头：



图 4.10.1 HS0038 近照

HS0038 只有三根引脚：分别是 VCC、GND 和信号线。而有些 HS0038 三个引脚分别为 GND、VCC 和信号线。我们通过外形就能区分两者。

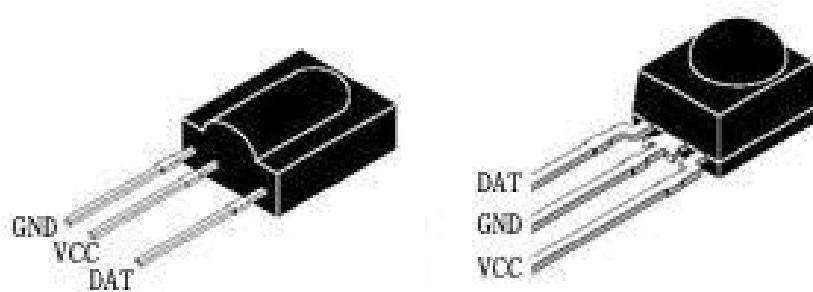


图 4.10.2 两种 HS0038 引脚图

打开实验板原理图，找到红外遥控模块。我们实验板用的接口是适用 VCC、GND、信号线型的 HS0038，如果你购买了 GND、VCC、信号线型的 HS0038 就要自己用杜邦线连接，否则反接 VCC 和 GND 将损坏模块。HS0038 信号线与单片机外部中断 0，即 P3.2 连接。

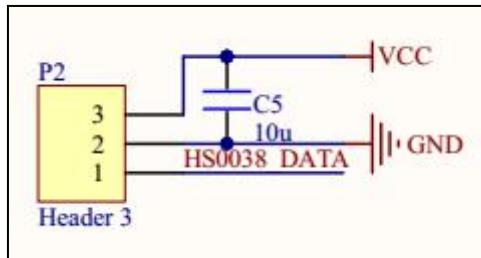


图 4.10.3 HS0038 电路连接

HS008 的接口在实验板的侧面，使用时只需插上去即可。**不过还是要留意不要插反！**



图 4.10.4 安插在实验板上的 HS0038

10.2 知识点讲解

既然有了红外接收探头，就要配一个红外遥控器。遥控种类很多，里面用的芯片也不尽相同，所以性能上也会各有差异。既然我们是一起来玩红外遥控的，为了防止硬件上的差异给我们带来不便，就统一用下图的 Car MP3 红外遥控。

遥控上一共有 21 个按键。当按下不同的按键时，遥控器都会发送一端数据供 HS0038 接收。

Car MP3 红外遥控发送和 HS0038 接收的数据都使用 NEC 协议。NEC 码由 5 部分：同步码头、地址码、地址反码、控制码和控制反码组成，每个数据都有 8 位：

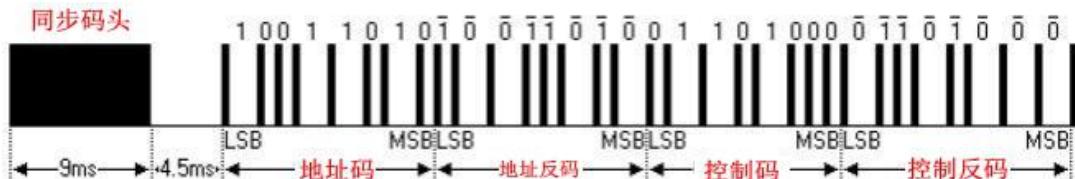


图 4.10.5 NEC 码格式

同步码头是一个起始标志，由 9ms 低电平和 4.5ms 高电平组成，标志 NEC 码的开始。地址码是遥控器的地址。不同遥控器发送的地址码是不同的，现在的 Car MP3 红外遥控地址码为 0。控制码是红外遥控发送的按键数据。我们按下按键时遥控都会发送对应这一个按键的按键数据，称为键值。至于地址反码和控制反码是用来校验数据的。如果通过编程将控制码取反后与控制反码判断不相等，则可以视为数据接收出错。另外在控制反码后，还有一个连发码。连发码由 9ms 低电平、2.5ms 高电平、0.56ms 低电平和 97.94ms 高电平组成。当按下一个按键不松手，遥控器发送完控制反码后会一直发送连发码。

NEC 码的逻辑 0 由 560us 低电平和 560us 高电平组成，而逻辑 1 由 560us 低电平和 1680us 高电平组成。不管是逻辑 0 还是逻辑 1，低电平的时间都是 560us。但两者高电平时间是不同的。所以我们要通过高电平时间而不是不能通过低电平的时间来判断逻辑 1 或逻辑 0。

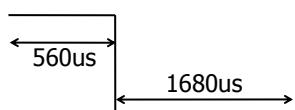


图 4.10.6 NEC 码逻辑 1

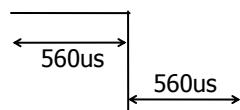


图 4.10.7 NEC 码逻辑 0

所以通过单片机解析 HS0038 收到的 4 个 8 位数据：地址码、地址反码、控制码和控制反码。每一位通过判断高电平时间来判断每一位的逻辑电平高低。然后我们就将解析出的控制码与键值表对照，就可以让单片机知道我们按下遥控哪个按键了。

10.3 软件实现

新建工程，命名为“红外遥控”。在 HARDWARE 文件中新建文件夹 REMOTE，文件下新建 remote.c 和 remote.h。从之前的工程中将文件夹 DIGITRON 复制到 HARDWARE 文件夹下。在工程所在的 USER 中新建文件 main.c。

在 remote.c 中写入以下程序：

```
#include "stc12.h"
#include "remote.h"
#include "digitron.h"

uchar retime;//记录高电平时间
uchar bitnum;//数据的位数
uchar startflag;//红外接收开始标志位
uchar receiveflag;//红外数据接收完成标志位
uchar finishflag;//红外数据处理完成标志位

uchar redata[33];//存放红外数据
uchar recode[4];//存放用户码和 4 个键值

uchar num = 0;//

//定时器 0 初始化函数
void Timer0_Init(void)
{
    TMOD = 0X02;//设置成工作方式 2
    TH0 = TL0 = 0;

    ET0 = 1;//打开定时器 0 中断
    EA = 1;//打开总中断
    TR0 = 1;//启动定时器
}

//定时器 0 中断服务函数
void Timer0_Handler(void) interrupt 1
{
    retime++;
}

//外部中断 0 初始化函数
void Init0_Init(void)
```

```

{
    IT0 = 1;//设置成下降沿触发
    EX0 = 1;//打开外部中断 0
    EA = 1;//打开总中断
}

//外部中断 0 服务函数
void Init0_Handler(void) interrupt 0
{
    //如果已经在接收当中，则继续接收
    if(startflag)
    {
        //检测到用户码，开始接收数据
        if(retime < 63 && retime > 32)
            bitnum = 0;

        redata[bitnum] = retime;//记录每次高电平的时间
        retime = 0;
        bitnum++;

        if(bitnum == 33)//判断是否接收完成
        {
            bitnum = 0;
            receiveflag = 1;//置 1 接收完成标志位，表示接收已经完成
        }
    }
    else//否则开始接收
    {
        startflag = 1;//置 1 开始标志位
        retime = 0;
    }
}

//红外数据处理函数
void Remote_Handler(void)
{
    uchar k,x,y;
    uchar value;

    k = 1;

    for(x = 0;x < 4;x++)//取 4 组数据
    {
        for(y = 0;y <= 8;y++)//每组数据 8 位
        {
            value >>= 1;
        }
    }
}

```

```

//判断红外数据是 1 还是 0
//如果高电平时间大于 6*255us 即 1.53ms , 则为高电平
if(redata[k] > 7)
    value |= 0X80;

k++;
}

recode[x] = value;
value = 0;
}

finishflag = 1;//置 1 处理完成标志位
}

//红外键值判断函数
uchar Remote_Count(void)
{
    uchar key;

    switch(recode[2])
    {
        case 34 : key = 0;
                    break;
        case 97 : key = 1;
                    break;
        case 227 : key = 2;
                    break;
        case 43 : key = 3;
                    break;
        case 225 : key = 4;
                    break;
        case 99 : key = 5;
                    break;
        case 171 : key = 6;
                    break;
        case 168 : key = 7;
                    break;
        case 170 : key = 8;
                    break;
        case 169 : key = 9;
                    break;
        default : key = 9;
    }

    return key;
}

//红外遥控解码函数

```

```
//在数码管上显示键值
uint Remote_Work(void)
{
    if(receiveflag == 1)
    {
        receiveflag =0;
        Remote_Handler();
    }
    if(finishflag == 1)
    {
        finishflag = 0;
        num = Remote_Count();
    }

    Digi_Num(num);

    return num;
}

//红外遥控控制 LED 函数
void Remote_ConLED(void)
{
    uint key;

    key = Remote_Work();

    switch(key)
    {
        case 0 : P0 = 0XFF;
            break;
        case 1 : P0 = 0X7F;
            break;
        case 2 : P0 = 0XBF;
            break;
        case 3 : P0 = 0XDF;
            break;
        case 4 : P0 = 0XEF;
            break;
        case 5 : P0 = 0XF7;
            break;
        case 6 : P0 = 0XFB;
            break;
        case 7 : P0 = 0XFD;
            break;
        case 8 : P0 = 0XFE;
            break;
        case 9 : P0 = 0X00;
            break;
        default : P0 = 0XFF;
    }
}
```

在 remote.h 中写入以下程序：

```
#ifndef _REMOTE_H_
#define _REMOTE_H_

#define uchar unsigned char
#define uint unsigned int

void Timer0_Init(void);
void Init0_Init(void);
void Remote_Handler(void);
uchar Remote_Count(void);
uint Remote_Work(void);
void Remote_ConLED(void);

#endif
```

在 main.c 中写入以下程序：

```
#include "stc12.h"
#include "remote.h"
#include "digitron.h"

void main()
{
    Timer0_Init();
    Init0_Init();
    Digi_Init();

    while(1)
    {
        Remote_ConLED();
    }
}
```

虽然程序很长又有点复杂，但总体的思路上我们把整个红外遥控过程分为三步：接收，解析和显示。

在接收过程中，我们要连续接收 32 位 NEC 码。NEC 码的逻辑 1 和逻辑 0 是要通过判断每位 NEC 码中高电平时间来区别的。但不管逻辑 1 还是逻辑 0，低电平的时间都是相同的，所以我们就能通过判断每位 NEC 码的时间来判断电平逻辑。我们用定时器 0 的 8 位自动重装载模式来记录每位 NEC 码的时间。定时器 0 每次溢出都代表着 256us。

```
//外部中断 0 服务函数
void Init0_Handler(void) interrupt 0
```

```

{
    //如果已经在接收当中，则继续接收
    if(startflag)
    {
        //检测到用户码，开始接收数据
        if(retime < 63 && retime > 32)
            bitnum = 0;

        redata[bitnum] = retime;//记录每次高电平的时间
        retime = 0;
        bitnum++;

        if(bitnum == 33)//判断是否接收完成
        {
            bitnum = 0;
            receiveflag = 1;//置 1 接收完成标志位，表示接收已经完成
        }
    }
    else//否则开始接收
    {
        startflag = 1;//置 1 开始标志位
        retime = 0;
    }
}

```

每位 NEC 码都是以下降沿结束的，于是我们就用外部中断 0 下降沿触发来时刻捕获红外信号下降沿的到来。这就是为什么我们要将 HS0038 信号端接 stc12 外部中断 0 的原因了。

当下降沿到来时，程序就进入外部中断 0 中断服务函数 Init0_Handler()。一开始接收时开始标志位 startflag 默认为 0，要先将 startflag 置 1。等下次进入中断服务函数时就可以开始计数了。这样可以防止某次接收过程中又重新开始接收。定时器每次溢出 retime 都会加 1，所以在 Init0_Handler()中我们就要通过读取每次 retime 的值并存入 33 位数组 redatal 中。于是我们就将每位 NEC 码的时间都以 retime 大小的形式存放在一个数组里供解析了。

还记得同步码头吗？这个是 NEC 信号的起始标志。当定时器溢出 32 次即这位 NEC 码的时间大于 8ms 时，就说明这是同步码头。当检测到同步码头时我们就要清零 bitnum，重新记录 NEC 码。当 bitnum 等于 33 时就说明接收完成了。于是我们就将接收完成标志位 receiveflag 置 1。

```

//红外数据处理函数
void Remote_Handler(void)

```

```

{
    uchar k,x,y;
    uchar value;

    k = 1;

    for(x = 0;x < 4;x++)//取 4 组数据
    {
        for(y = 0;y <= 8;y++)//每组数据 8 位
        {
            value >>= 1;

            //判断红外数据是 1 还是 0
            //如果高电平时间大于 6*255us 即 1.53ms , 则为高电平
            if(redata[k] > 7)
                value |= 0X80;

            k++;
        }

        recode[x] = value;
        value = 0;
    }

    finishflag = 1;//置 1 处理完成标志位
}

```

接收完成后，我们就要对数组 redata 中 32 位数据进行解析了。在函数 Remote_Handler() 中，我们要将数组 redata 中的 32 个时间数据解析成逻辑 1 或逻辑 0，然后将解析出来的 4 个 8 位 NEC 码存放在数组 recode 中。我们依次判断数组 redata 中 retime 的个数。因为 NEC 码逻辑 0 时间位 1120us (560us + 560us)，而逻辑 1 时间为 2240us (560us + 1680us)。所以如果 retime 大于 6，即这位 NEC 码时间大于 1536us (6 x 256us)，就说明该位为 1，否则该位为 0。这样我们就依次解析得 4 个 8 位 NEC 码了。

虽然我们分别得到地址码、地址反码、控制码和控制反码，但我们会用到控制码来分析我们到底按下哪个按键。Car MP3 红外遥控器每个键的键值如下表。

键值	名称	键值	名称	键值	名称
72	OFF	-	-	8	MENU
104	TEST	232	UP	136	RETURN
0	REWIND	66	PLAY	193	FORWARD

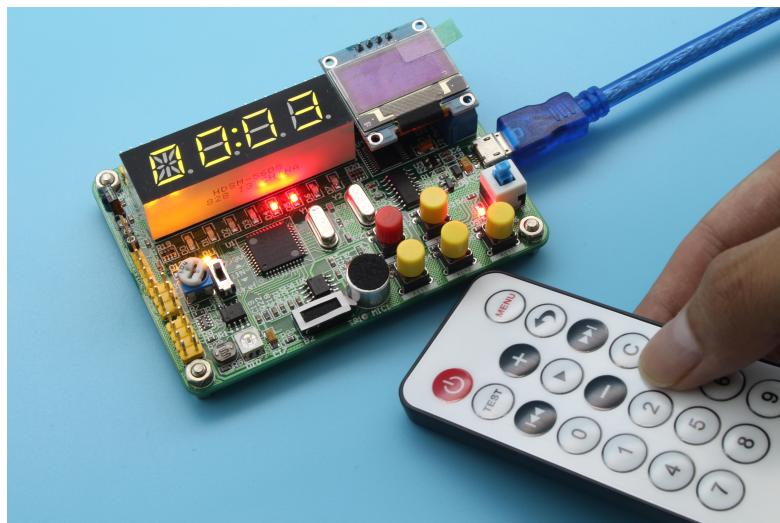
34	0	195	DOWM	65	C
97	1	227	2	43	3
225	4	99	5	171	6
168	7	170	8	169	9

表 4.10.1 Car MP3 红外遥控器键值表

函数 Remote_Count()中通过将数组 recode 第 3 位，即 recode[2]和各个键值作比较，就能得知我们按下哪个按键了。

最后一步是将我们按下数字按键对应的数字显示在数码管上。在 Remote_Show() 函数中我们依次接收和解析 HS0038 接收到的 NEC 码。在判断按键后将相应的数字显示在数码管上。

10.4 下载验证



10.5 总结思考

因为红外遥控过程比较多，涉及的数据量也比较大，所以我们一开始就声明了比较多的变量和数组。

在这些变量中，值得一提的，就是那三个标志符：外接收开始标志位 startflag、红外数据接收完成标志位 receiveflag 以及红外数据处理完成标志位 finishflag。

其实标志位，大家都不会陌生，不管定时器的溢出、串口的通讯和 ADC 的转换结束，都涉及到标志位。但这些标志位都是在特殊功能寄存器中，但符合一定条件是就由硬件置 1 或置 0，籍此来供程序员判断单片机的工作状态。

但在红外遥控中，并没有寄存器来判断单片机的工作状态。而我们却要判断程序到底进行到哪一步，哪一步进行完了要进行哪一步。此时，我们自己声明的标志位就派上大用场了。当某个步骤执行完后，我们就将相应的标志位置 1。

```
//红外遥控解码函数
//在数码管上显示键值
uint Remote_Work(void)
{
    if(receiveflag == 1)
    {
        receiveflag =0;
        Remote_Handler();
    }
    if(finishflag == 1)
    {
        finishflag = 0;
        num = Remote_Count();
    }

    Digi_Num(num);

    return num;
}
```

例如，当接受结束后，我们就将红外数据接收完成标志位 receiveflag 置 1。在解码函数中，我们如果判断到 receiveflag 为 1，就将它清零后执行数据处理函数。可见，除了要自己置 1 外，我们声明的标志位与串口的接收发送标志位和 ADC 的转换结束标志位没任何区别。

所以，我们可以根据程序的实际需要，声明一些标志位，用来判断程序的执行情况。

十一、I²C——总线协议入门

单片机外设种类成百上千，如果每一种外设都按照特定的时序去控制，那岂不是要有成百上千种控制方案？这给硬件开发商和用户都是1个头大的问题。为了解决这个问题，飞利浦半导体公司在上世纪八十年代设计了一种简单、双向、二线制、同步串行总线协议——I²C总线协议。只要硬件上支持I²C控制的外设，单片机都可以用任意两个引脚与之进行通讯进而控制。因为它的简单实用，I²C已经成为一种世界级的通用标准。

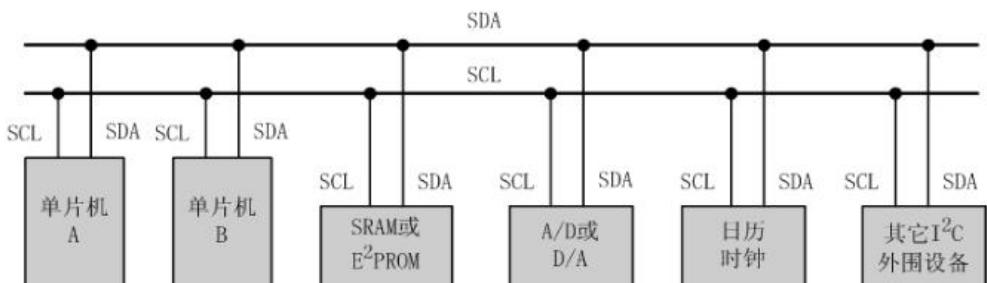
11.1 知识点讲解

I²C总线协议，英文全称 Inter-Integrated Circuit，简称 IIC。

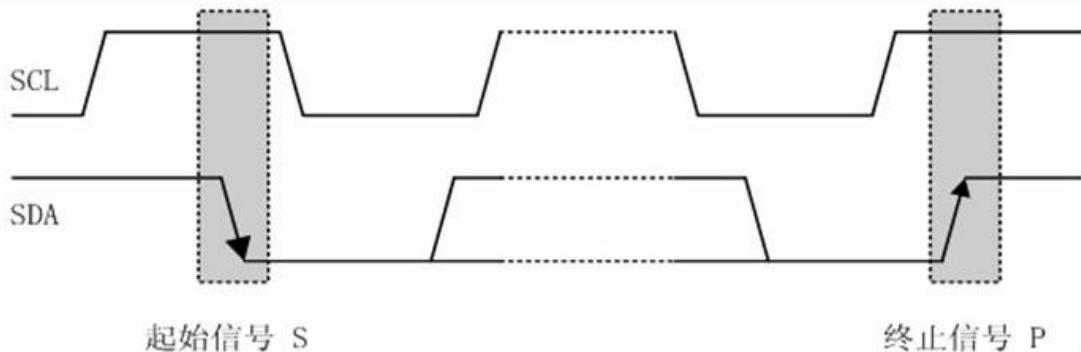
所谓总线，就是从单片机引出一条“高速公路”。只要硬件上适用 IIC 的外设都可以连接上总线，就像与总线搭了一条“国道”。这样单片机就能通过总线与多个外设甚至是单片机进行通讯了。

总线是一条，但外设是可以多个的。单片机引脚发出的信号并不会识别器件，如果把信号发到总线上，那它一下子就传达给所有外设，一呼百应那就必定天下大乱，怎样做到控制某一外设？为此，我们就要给单片机和外设之间规定一个协议。主机和各从机的通讯只有按着这个协议进行，通讯才能有条不紊地进行。

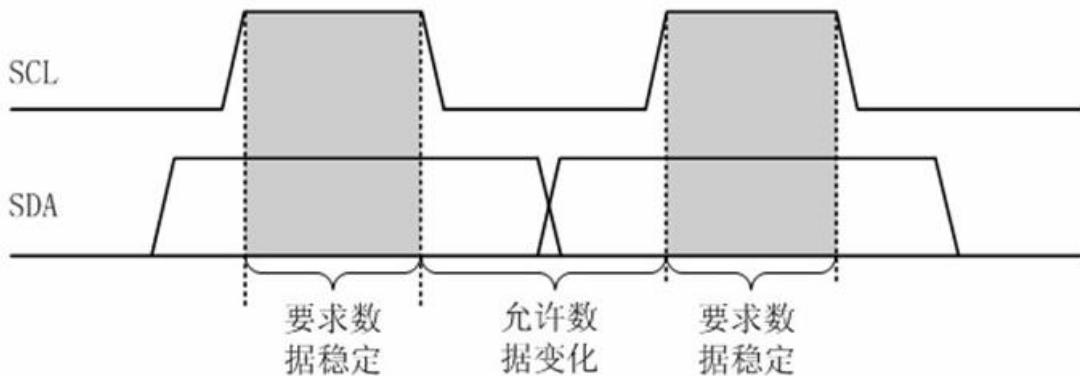
在 IIC 中，发出信号的单片机称为主机，接受信号外设称为从机。**在理解 IIC 发送读取数据时，要站在主机即单片机的角度去理解！**除了模块本身所需的电源线和地线，IIC 只需从单片机任意端口引出两条线：控制线 SCL 和数据线 SDA。



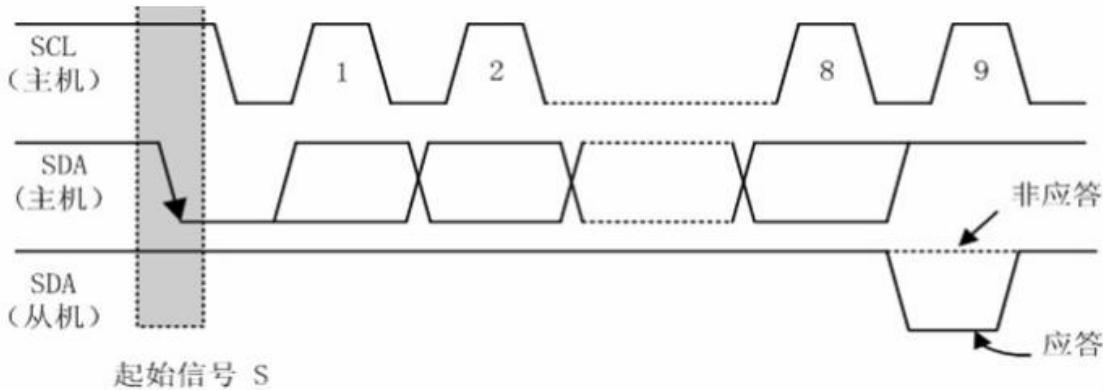
在开始 IIC 通讯时，主机先要发送 1 个起始信号，告诉与总线连接的各个外设要进行 IIC 通讯了。起始信号是当 SCL 为高时，SDA 由高变低。当 IIC 通讯结束时，主机也要发送 1 个终止信号。终止信号是当 SCL 为高时，SDA 由低变高。



在传输数据时，我们切记切记要遵循 1 个规则：当 SCL 为高时，SDA 上的数据要保持不变；当 SCL 为低时，SDA 上的数据才能变化。



数据传输的格式是一帧数据含有 9 位。首先主机从最高位开始发送一节 8 位数据，然后在最后面加一位从机返回的应答位。应答位是保证从机能收到主机的信号的保障。如果主机发送信号后得不到从机的应答信号，就认为通讯失败。



我们开车在高速公路上驾驶，如果我们想知道应当在哪里下高速，路牌是 1 个很好的指示。IIC 总线这条“高速公路”也是如此。为了区别连接在总线上个各个从机和从机内的寄存器，每个从机和其内部的寄存器都有各自的地址。从机的地址有的是硬件生产商设置好、不能修改的；有的是生产商设置了其中几位，用户可以设置另外几位；有的从机有地址寄存器，用户可以自定义其地址。而从机内部的寄存器承担着从机的各个功能，它们的地址是生产商设置好不能改的。

主机向从机发送数据时，第一步先向总线发送 1 个起始信号。然后主机向总线发送 7 位从机地址加最低一位方向控制（加 0 为发送，加 1 为读取），当有从机的地址与该地址相同时，会向主机发送 1 个应答信号。当主机收到应答信号后，就会向总线发送一节寄存器地址。如果从机有该地址的寄存器，就会给主机发送 1 个应答信号。当主机收到应答信号后，主机就和该从机的该寄存器建立起通讯了。接着主机就可以给该寄存器发送数据了。发送结束后，主机向总线发送终止信号终止通讯。

反过来，当主机读取从机数据时，同样先向总线发送 1 个起始信号。然后分别发送从机地址和寄存器地址。在单片机都得到从机响应后，主机先向总线再次发送 1 个起始信号，然后向总线发送 1 个 7 位从机地址加最低一位 0 位，表示要从从机读取数据，然后就可以开始读取数据了。读取数据结束后，主机向总线发送终止信号终止通讯。

11.2 软件实现

在工程中新建文件 iic.c，并写入下列程序

```
#include "stc12.h"
#include "iic.h"

//IIC 开始函数
```

```
void IIC_Start(void)
{
    SCL = 1;
    SDA = 1;
    SDA = 0;
    SCL = 0;
}

//IIC 结束函数
void IIC_Stop(void)
{
    SCL = 0;
    SDA = 0;
    SCL = 1;
    SDA = 1;
}

//IIC 写一节数据函数
void IIC_Write(uchar Byte)
{
    uchar i;

    for(i = 0;i < 8;i++)
    {
        if(Byte & 0x80)
            SDA = 1;
        else
            SDA = 0;

        SCL = 1;
        SCL = 0;

        Byte <<= 1;
    }

    SDA = 1;
    SCL = 1;
    SCL = 0;
}

//IIC 读一节数据函数
uchar IIC_Read(void)
{
    uchar Byte,i;

    SCL = 0;
    SDA = 1;

    for(i = 0;i < 8;i++)
    {
        SCL = 1;
```

```
        Byte = (Byte << 1) | SDA;  
        SCL = 0;  
    }  
  
    return Byte;  
}
```

新建 iic.h , 并写入下列程序

```
#ifndef _IIC_H_  
#define _IIC_H_  
  
#define uint unsigned int  
#define uchar unsigned char  
  
sbit SCL = P3^6;  
sbit SDA = P3^7;  
  
void IIC_Start(void);  
void IIC_Stop(void);  
void IIC_Write(uchar Byte);  
uchar IIC_Read(void);  
  
#endif
```

新建 main.c , 并写入下列程序

```
#include "reg52.h"  
#include "iic.h"  
  
void main()  
{  
    while(1);  
}
```

这里，我们只写了一个 IIC 总线通讯协议，却并没有任何实际应用。大家别急，在下一节中，我们将运用 IIC 来控制一种显示器——OLED。

十二、OLED——I²C 实战演练

这几年，OLED 的势头越来越大。如果你有留意，你会发现，越来越多的电视和手机屏幕开始使用 OLED 屏。OLED 屏幕凭借高清、省电可弯曲等优点，正一步一步冲击着传统 LCD 屏幕的市场。借着这股东风，刚出的 iPhone8 屏幕采用的就是 OLED 屏！

当然，像手机屏幕那种 OLED 屏用 51 单片机是控制不来的。但为了让大家能够一睹 OLED 的风采，我们退而求其次，选用以前 MP3 普遍采用的 0.96 英寸的 OLED，来为大家演示如何操作 OLED。

12.1 硬件准备

OLED 中文名叫有机发光二极管，英文全称叫 Organic Light-Emitting Diode。OLED 显示技术具有自发光、广视角、几乎无限高的对比度、较低耗电和极高速反应等优点。

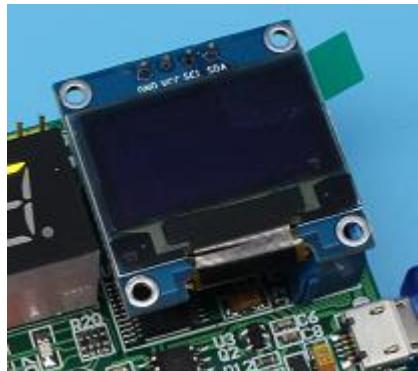


图 4.12.1 实验板上的 OLED

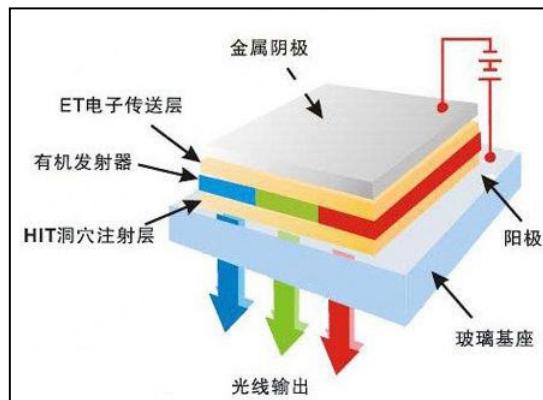


图 4.12.2 OLED 发光原理图

我们教程用的是 0.96 英寸采用 IIC 控制的 OLED，它屏幕有 128*64 个像素点。其中前 16 行像素为黄色，其余像素为蓝色。

12.2 知识点讲解

通过数据发送指令，我们就能在 OLED 上一个一个地打点。但这打点不是随便打的——我们通过 IIC 发送给 OLED 的数据会转为二进制，从低位到高位在 OLED 上从上到下依次控制每个像素点。其中 0 代表这个像素点熄灭，1 代表点亮这个像素点。因为我们数据发送是一个字节 8 位为单位发送的，所以我们没发送一次数据，OLED 上就显示一列 8 个相应的点。

显然，只有一列的点是不可能显示复杂的图案的。在发送一次数据后，OLED 会自动向右换列。如果我们连续给 OLED 发送 8 次数据，那我们就在 OLED 上的一个 8*8 像素的区域打出相应的图案了。

在 OLED 上一个 8*8 像素区域打点只是显示的第一步。大家可以观察一下，8*8 像素在 OLED 上只是一个很小的区域，在一个很小的区域内显示复杂文字往往失真。而且我们要显示的图案、字符不能被控制在 8*8 像素的大小。所以我们通过多个 8*8 像素区域的组合，显示出 8*16、16*16 或者 48*48 等大小的图案和字符。

12.3 软件实现

新建工程，命名为“OLED”。在 HARDWARE 文件中新建文件夹 OLED，文件下新建 oled.c 和 oled.h。从之前的工程中将文件夹 IIC 复制到 HARDWARE 文件夹下。在工程所在的 USER 中新建文件 main.c。

在 oled.c 中写入以下程序：

```
#include "stc12.h"
#include "oled.h"
#include "table.h"
#include "iic.h"

//毫秒延时函数，误差 2us
void Delay_ms(uint z)
{
    uint x,y;

    for(x = z;x > 0;x--)
        for(y = 921;y > 0;y--);

}

//OLED 写命令函数
```

```
void OLED_Com(uchar IIC_Command)
{
    IIC_Start();
    IIC_Write(0x78);
    IIC_Write(0x00);
    IIC_Write(IIC_Command);
    IIC_Stop();
}

//OLED 写数据函数
void OLED_Dat(uchar IIC_Data)
{
    IIC_Start();
    IIC_Write(0x78);
    IIC_Write(0x40);
    IIC_Write(IIC_Data);
    IIC_Stop();
}

//OLED 设置坐标函数
void OLED_Set_Pos(uchar x,uchar y)
{
    OLED_Com(0xb0 + y);
    OLED_Com(((x & 0xf0) >> 4) | 0x10);
    OLED_Com((x & 0x0f) | 0x01);
}

//OLED 全屏
void OLED_Fill(uchar bmp_dat)
{
    uchar y,x;

    for(y = 0;y < 8;y++)
    {
        OLED_Com(0Xb0 + y);
        OLED_Com(0X01);
        OLED_Com(0X10);
        for(x = 0;x < X_WIDTH;x++)
            OLED_Dat(bmp_dat);
    }
}

//OLED 复位
void OLED_CLS(void)
{
    OLED_Fill(0X00);
}

//OLED 初始化函数
void OLED_Init(void)
```

```

{
    Delay_ms(500); //初始化之前的延时很重要！
    OLED_Com(0XAE);
    OLED_Com(0X00);
    OLED_Com(0X10);
    OLED_Com(0X40);
    OLED_Com(0X81);
    OLED_Com(0XCF);
    OLED_Com(0XA1);
    OLED_Com(0XC8);
    OLED_Com(0XA6);
    OLED_Com(0XA8);
    OLED_Com(0X3F);
    OLED_Com(0XD3);
    OLED_Com(0X00);
    OLED_Com(0XD5);
    OLED_Com(0X80);
    OLED_Com(0XD9);
    OLED_Com(0XF1);
    OLED_Com(0XDA);
    OLED_Com(0X12);
    OLED_Com(0XDB);
    OLED_Com(0X40);
    OLED_Com(0X20);
    OLED_Com(0X02);
    OLED_Com(0X8D);
    OLED_Com(0X14);
    OLED_Com(0XA4);
    OLED_Com(0XA6);
    OLED_Com(0XAF);
    OLED_Fill(0X00); //初始清屏
    OLED_CLS();
    OLED_Set_Pos(0,0);
}

//OLED 显示字符函数
//字体大小 : 8*16
void OLED_ShowNum(uchar x,uchar y,uint a)
{
    uint i = 0;

    //显示超出屏幕最右端，自动换行
    if(x > 120)
    {
        x = 0;
        y++;
    }

    OLED_Set_Pos(x,y);
    for(i = 0;i < 8;i++)

```

```

        OLED_Dat(Table_Num[a * 16 + i]);
        OLED_Set_Pos(x,y + 1);
        for(i = 0;i < 8;i++)
            OLED_Dat(Table_Num[a * 16 + i + 8]);

        x += 8;
    }

//OLED 显示一个 4 位数
//x , y 为首位所在坐标
//y 取值范围 : 0、1、2、3
//num 取值范围 : 0~9999
void OLED_Num(uchar x,uchar y,uint num)
{
    uint a,b,c,d;

    if((num > 0) || (num < 9999))
    {
        a = num / 1000;
        b = num % 1000 / 100;
        c = num % 1000 % 100 / 10;
        d = num % 1000 % 100 % 10;

        y *= 2;

        OLED_ShowNum(x,y,a);
        x += 8;
        OLED_ShowNum(x,y,b);
        x += 8;
        OLED_ShowNum(x,y,c);
        x += 8;
        OLED_ShowNum(x,y,d);
    }
}

//OLED 显示小写字母函数
//字体大小 : 8*16
void OLED_Show_let(uchar x,uchar y,uint a)
{
    uint i = 0;

    //显示超出屏幕最右端 , 自动换行
    if(x > 120)
    {
        x = 0;
        y++;
    }

    OLED_Set_Pos(x,y);
}

```

```

        for(i = 0;i < 8;i++)
            OLED_Dat(Table_Let[a * 16 + i]);
        OLED_Set_Pos(x,y + 1);
        for(i = 0;i < 8;i++)
            OLED_Dat(Table_Let[a * 16 + i + 8]);

        x += 8;
    }

//OLED 显示大写字母函数
//字体大小 : 8*16
void OLED_Show_Caplet(uchar x,uchar y,uint a)
{
    uint i = 0;

    //显示超出屏幕最右端 , 自动换行
    if(x > 120)
    {
        x = 0;
        y++;
    }

    OLED_Set_Pos(x,y);
    for(i = 0;i < 8;i++)
        OLED_Dat(Table_Caplet[a * 16 + i]);
    OLED_Set_Pos(x,y + 1);
    for(i = 0;i < 8;i++)
        OLED_Dat(Table_Caplet[a * 16 + i + 8]);

    x += 8;
}

//OLED 显示中文函数
//字体大小 : 16*16
//longsize 为中文字符长度
//table 为中文字符串数组
void OLED_Chn(uchar x,uchar y,uchar longsize,uchar *table)
{
    uint i,j = 0,k;

    //显示超出屏幕最右端 , 自动换行
    if(x > 112)
    {
        x = 0;
        y++;
    }

    for(k = 0;k < longsize;k++)
    {

```

```

        OLED_Set_Pos(x + k * 16,y);
        for(i = 0;i < 8;i++)
        {
            OLED_Dat(table[j]);

            j++;
        }

        OLED_Set_Pos(x + 8 + k * 16,y);
        for(i = 0;i < 8;i++)
        {
            OLED_Dat(table[j]);

            j++;
        }

        OLED_Set_Pos(x + k * 16,y + 1);
        for(i = 0;i < 8;i++)
        {
            OLED_Dat(table[j]);

            j++;
        }

        OLED_Set_Pos(x + 8 + k * 16,y + 1);
        for(i = 0;i < 8;i++)
        {
            OLED_Dat(table[j]);

            j++;
        }
    }
}

//OLED 显示图案
//size_x 和 size_y 控制图案显示的长和宽
//table 为图案数组
void OLED_BMP(uchar x,uchar y,uchar size_x,uchar size_y,uchar *table)
{
    uint row,column,j = 0,k;

    for(column = y;column < size_y + y;column++)
    {
        for(row = x;row < size_x + x;row++)
        {
            OLED_Set_Pos(row * 8,column);
            for(k = 0;k < 8;k++)
            {
                OLED_Dat(table[j]);

                j++;
            }
        }
    }
}

```

```
        }
    }
}
```

在 oled.h 中写入以下程序：

```
#ifndef _OLED_H_
#define _OLED_H_

#define uint unsigned int
#define uchar unsigned char

#define Brightness 0xCF
#define X_WIDTH 128
#define Y_WIDTH 64

void Delay_ms(uint z);
void OLED_Com(uchar IIC_Command);
void OLED_Dat(uchar IIC_Data);
void OLED_Set_Pos(uchar x,uchar y);
void OLED_Fill(uchar bmp_dat);
void OLED_CLS(void);
void OLED_Init(void);
void OLED_ShowNum(uchar x,uchar y,uint a);
void OLED_Num(uchar x,uchar y,uint num);
void OLED_Show_let(uchar x,uchar y,uint a);
void OLED_Show_Caplet(uchar x,uchar y,uint a);
void OLED_Chn(uchar x,uchar y,uchar longsize,uchar *table);
void OLED_BMP(uchar x,uchar y,uchar size_x,uchar size_y,uchar *table);

#endif
```

在 main.c 中写入以下程序：

```
#include "stc12.h"
#include "oled.h"

void main()
{
    OLED_Init();

    while(1)
    {
        OLED_Chn(16,0,6,Table_Chn);
        OLED_BMP(0,2,6,6,Table_BMP);
    }
}
```

我们先从控制 OLED 的最底层开始说起。

我们采用上一节所讲的 IIC 来控制 OLED。IIC 写数据的格式是主机先向总线发送 IIC 开始时序，告诉总线开始 IIC 通讯。然后主机往总线依次发送从机地址、寄存器地址和数据。最后主机发送 IIC 结束时序，结束 IIC 通讯。OLED 的地址 0X78、数据寄存器地址 0X40 和指令寄存器地址 0X00 是已知的，所以我们不用等待从机的回复。根据这个思路，我们就能写出 OLED 发送数据函数 OLED_WrDat() 和发送指令函数 OLED_WrCmd()。

有了向 OLED 发送数据的 OLED_WrDat() 和发送指令的 OLED_WrCmd() 两个子函数后，我们就利用它们来设置 OLED 的相关配置。

第一步是设置坐标。我们说过每给 OLED 发送一字节数据，OLED 就会从上往下显示 8 个相应的像素点。但从哪里开始打点了，这是一个问题。所以我们打点前，先通过函数 OLED_Set_Pos() 来设置当前的坐标。横轴方向 X 取值范围为 0~127，竖轴方向 Y 取值范围为 0~7。这里值得注意的是 Y 轴反向。Y 取值范围只有 0~7，换言之 OLED 整个屏幕被划分成 8 行。所以我们的 8*8 区域只能在每行之内而不能在两行之间。Y 轴方向共 8 行，每行 8 位像素，总共就 64 位像素。

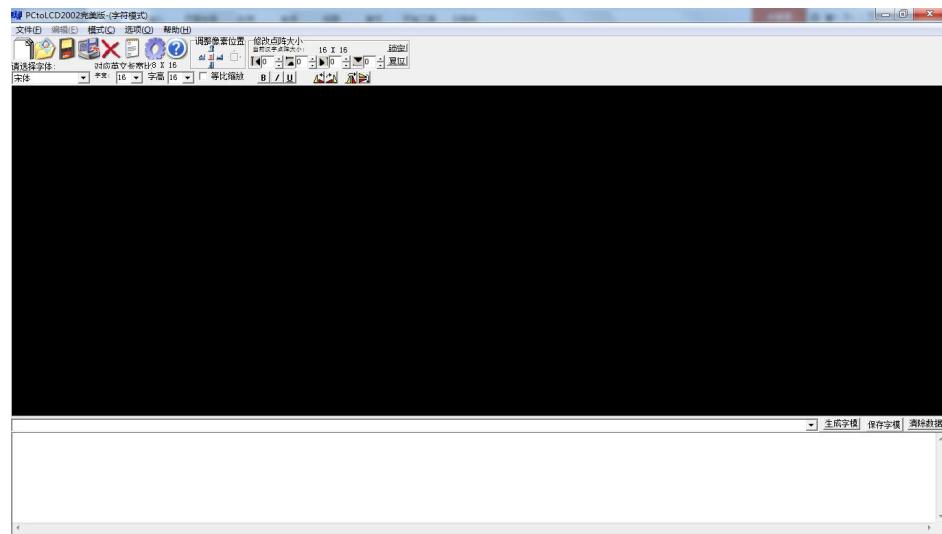
接下来是全屏填充函数 OLED_Fill() 和清屏函数 OLED_CLS()。OLED_Fill() 是逐点将 OLED 点亮或熄灭，而 OLED_CLS() 是调用 OLED_Fill() 给 OLED 全送 0，将 OLED 清屏。在初始化 OLED 时一定要将 OLED 清屏，否则将出现花屏的情况，影响显示。

初始化函数 OLED_Init() 没什么好说的，配置 OLED 时用到的指令都是手册或者官方例程附带的，我们无需理解其中的内容。但还是有点要注意的，在初始化前一定要有一小段延时，否则可能会影响显示。

以上是 OLED 的基本设置。接下来我们就开始在 OLED 上显示数字、字母、中文以及图案。

但是，如果我要显示数字“1”，那该发送什么数据给 OLED 呢？这时候，我们就要借助取模软件了，将字符转换为二进制数据。

打开光盘中的 3. 软件资料 -> 4. OLED 取模 -> PCtoLCD2002



取模软件有两个模式，一个是字符模式，另一个是图形模式。模式选择在上方模式选项中可以选择。我们先用字符模式，而软件默认为字符模式，所以模式选择可以暂时不管。

我们知道 51 单片机编程可以用 C 语言或汇编语言，软件设计者也考虑过这个问题。所以软件既可以生成字符对应的 C 语言码，也可以生成汇编码。软件默认生成汇编码，所以我们要设置好。

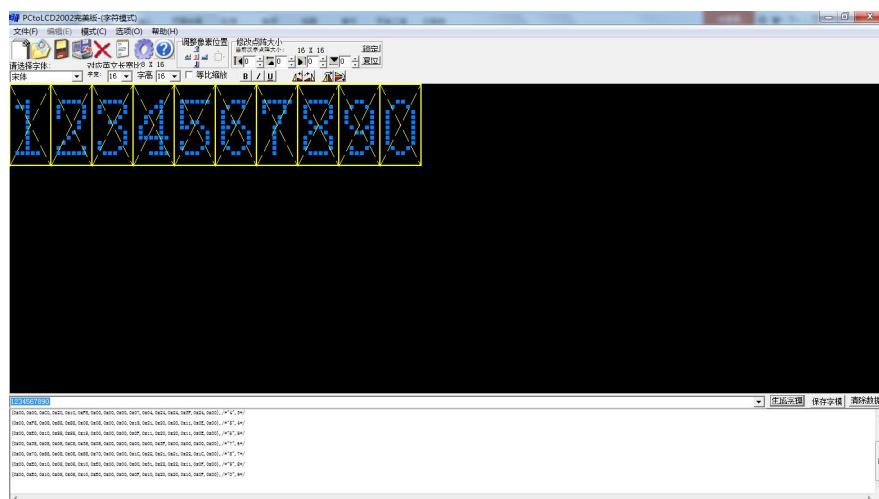
点击上方选项中的选项。



在自定义格式中，我们看到一个“A51 格式”。A，就是汇编语言 Assembly Language 的首字母，说明软件默认情况下是生成汇编码的。我们把它改成“C51 格式”，让软件生成 C51 码。



然后在下方黑白交界处，有个对话框，在里面输入“0123456789”，点击右方“生成字模”，在下方白色区域就会生成每个数字对应的 C51 码。



打开工程，新建文件“table.h”。保存在工程文件 HARDWARE 下的 OLED 文件中。table.h 是用来存放数字、字母、中文以及图案代码的文件。

将刚刚在软件中数字生成的 C51 码复制，以数组的形式存放在 table.c 中

```
//数字
//大小 : 8*16
code unsigned char Table_Num[] =
{
0x00,0xE0,0x10,0x08,0x08,0x10,0xE0,0x00,0x00,0x0F,0x10,0x20,0x20,0x10,0x0F,0x
00,/*"0",0*/
0x00,0x10,0x10,0xF8,0x00,0x00,0x00,0x00,0x00,0x20,0x20,0x3F,0x20,0x20,0x00,0x
00,/*"1",1*/
```

```

0x00,0x70,0x08,0x08,0x08,0x88,0x70,0x00,0x00,0x30,0x28,0x24,0x22,0x21,0x30,0x
00,/*"2",2*/
0x00,0x30,0x08,0x88,0x88,0x48,0x30,0x00,0x00,0x18,0x20,0x20,0x20,0x11,0x0E,0x
00,/*"3",3*/
0x00,0x00,0xC0,0x20,0x10,0xF8,0x00,0x00,0x00,0x07,0x04,0x24,0x24,0x3F,0x24,0x
00,/*"4",4*/
0x00,0xF8,0x08,0x88,0x88,0x08,0x08,0x00,0x00,0x19,0x21,0x20,0x20,0x11,0x0E,0x
00,/*"5",5*/
0x00,0xE0,0x10,0x88,0x88,0x18,0x00,0x00,0x00,0x0F,0x11,0x20,0x20,0x11,0x0E,0x
00,/*"6",6*/
0x00,0x38,0x08,0x08,0xC8,0x38,0x08,0x00,0x00,0x00,0x00,0x3F,0x00,0x00,0x00,0x
00,/*"7",7*/
0x00,0x70,0x88,0x08,0x08,0x88,0x70,0x00,0x00,0x1C,0x22,0x21,0x21,0x22,0x1C,0x
00,/*"8",8*/
0x00,0xE0,0x10,0x08,0x08,0x10,0xE0,0x00,0x00,0x00,0x31,0x22,0x22,0x11,0x0F,0x
00,/*"9",9*/
};

```

同理，我们也可以将 a~z 和 A~Z 取模，生成 C51 码存放在 table.h 中。

在声明数组时，我们用到了关键字 code。我们说过，用关键字 code 声明的数组将存放在 ROM 中，这是因为，除了数字，还有大小写字母以及图案，这些字符图案取模后的 C51 码太大了，加上程序本身，将会是 RAM 溢出。所以我们把它们放在 RAM 中，其中的好处，我们在数码管一节中就已经说过了。这节结束后，你也可以尝试将 code 删去，观察 OLED 的显示情况。

我们默认数字和字母的大小都为 8*16。为了让大家直观了解我们是怎样在 OLED 上把它们打印出来的，我们一起通过图片来逐一分析一下。

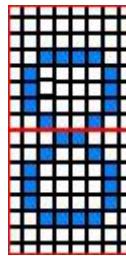


图 4.12.3 取模后是数字 8

以数字“8”为例，可以看到，它是由两上下个 8×8 像素区域组成的。上面我们讲过，我们每次打印一个 8×8 的区域，那我们就要先打印上半部分，然后再打下半部分。根据这个原理，我们来分析一下显示字符函数OLED_ShowNum()。

```
//OLED 显示数字函数
//字体大小：8*16
void OLED_ShowNum(uchar x, uchar y, uint a)
{
    uint i = 0;

    //显示超出屏幕最右端，自动换行
    if(x > 120)
    {
        x = 0;
        y++;
    }

    OLED_Set_Pos(x, y);
    for(i = 0; i < 8; i++)
        OLED_Dat(Table_Num[a * 16 + i]);
    OLED_Set_Pos(x, y + 1);
    for(i = 0; i < 8; i++)
        OLED_Dat(Table_Num[a * 16 + i + 8]);

    x += 8;
}
```

OLED_ShowNum()接受3个型参。X和y是决定字符显示位置的。它是字符最左上角的像素对应的坐标。而型参a就是我们要显示的数字了。

在设置坐标前，我们要先判断有没有超出限制范围。我们知道一个数字占的像素点为 8×16 ，而OLED可以显示的像素点为 128×64 。如果显示数字坐标的x值再120~128之间的话，就会超出OLED的显示范围。所以在显示每个数字前，都要判断坐标x的值。如果x在120~128之间，就将y值加一，即换行继续显示。

修整好显示的位置后，我们就开始在OLED上作画。首先，用坐标设置函数OLED_Set_Pos()将坐标移至我们想要的位置。然后用for循环以及数据发送函数OLED_WrDat()依次将数字上半部分对应的8个数据发送给OLED。然后换行，继续打印数字的下半部分。

你有没有思考过，我们要显示数字a，但为什么要用数组Table_Num[]的第 $a * 16 + i$ 项和第 $a * 16 + i + 8$ 项呢？其实这只是一个数学问题。因为显示一个数字，需要上下部分共16个二进制数据。那我们要显示数字a，不就要从第 $a * 16$ 个数据开始发送吗？

同理，大小写字母也是 $8 * 16$ 的大小。所以显示显示大小写字母的函数和显示数字的函数如出一辙。我们看小写字母函数 OLED_Show_let() 和大写字母函数 OLED_Show_Caplet()。和数字显示函数不同的是，字母显示函数用到的数组也是相应不同的——用的是用取模软件得到的大小写字母二进制代码组成的数组。

```
//OLED 显示中文函数
//字体大小：16*16
//longsize 为中文字符串长度
//table 为中文字符串数组
void OLED_Chn(uchar x,uchar y,uchar longsize,uchar *table)
{
    uint i,j = 0,k;

    //显示超出屏幕最右端，自动换行
    if(x > 112)
    {
        x = 0;
        y++;
    }

    for(k = 0;k < longsize;k++)
    {
        OLED_Set_Pos(x + k * 16,y);
        for(i = 0;i < 8;i++)
        {
            OLED_Dat(table[j]);

            j++;
        }

        OLED_Set_Pos(x + 8 + k * 16,y);
        for(i = 0;i < 8;i++)
        {
            OLED_Dat(table[j]);

            j++;
        }

        OLED_Set_Pos(x + k * 16,y + 1);
        for(i = 0;i < 8;i++)
        {
            OLED_Dat(table[j]);

            j++;
        }

        OLED_Set_Pos(x + 8 + k * 16,y + 1);
        for(i = 0;i < 8;i++)
        {
    }
```

```
        OLED_Dat(table[j]);  
  
        j++;  
    }  
}
```

在理解了如何显示数字和字母后，显示中文和图案就容易的多了。毕竟，不管你要显示什么，都先取模。然后将图案分割成一个个有 $8 * 8$ 个像素组成的区域。然后依次将这个区域显示出来就行了。但依次的顺序是什么？我们在显示数字和字母时，都是 $8 * 16$ 大小的，所以只要从上到下显示就行了。如果遇到中文或者图案，我们就要遵循从左到右再从上到下的原则来显示每个 $8 * 8$ 区域。

我们将“傀儡师实验板”6个字取模后，存放在 table.h 中的数组 Test_Chn[] 中。每个中文都是 $16 * 16$ 大小，换言之由 4 个 $8 * 8$ 区域构成。以“傀”字为例，如下图：

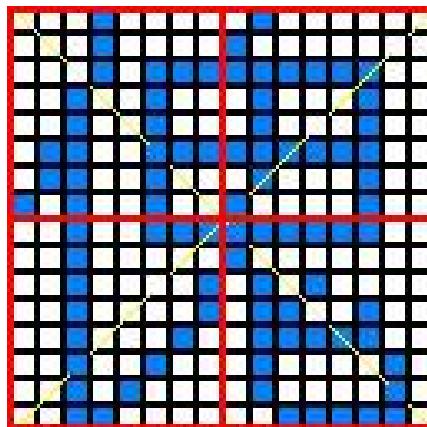


图 4.12.4 取模后的“傀”字

我们遵循从左到右再从上到下的原则，依次打印每个区域。我们看中文显示函数 OLED_Chn()。如何打印每个区域，我们就不再赘述。这里有趣的是在设置坐标时，用数学的方法将一个中文的上下左右四个区域的坐标和每个中文的位置联系起来。我在这里就不过多的分析，你可以先自己动手分析一下，然后在看看例程中是如何实现的。

另外，OLED_Chn()的形参除了坐标 x 和 y 外，还有控制长度变量 longside 和指针变量 table。有了 table 数组，我们将取模后的数据存放在里面，然后用函数 OLED_Chn() 直接调用数组名就能显示相应的中文了。而长度变量 longside 能控制显示字符的长度，不管字符长度怎样改变，我们都不要改写程序，直接改 longside 即可。这两个变量使 OLED_Chn() 有更强的兼容性。

```

//OLED 显示图案
//size_x 和 size_y 控制图案显示的长和宽
//table 为图案数组
void OLED_BMP(uchar x,uchar y,uchar size_x,uchar size_y,uchar *table)
{
    uint row,column,j = 0,k;

    for(column = y;column < size_y + y;column++)
    {
        for(row = x;row < size_x + x;row++)
        {
            OLED_Set_Pos(row * 8,column);
            for(k = 0;k < 8;k++)
            {
                OLED_Dat(table[j]);
                j++;
            }
        }
    }
}

```

最后，就是现实图案了。取模软件只支持将 BMP 格式的图转换为二进制数据。所以在将图案取模前，要先通过其他软件将图案转换成 BMP 格式。而且我们同时要注意图案的大小。一般修图软件都可以将图案缩放成一定的像素大小。不要忘记，OLED 最大只能显示 128 * 64 大小的图案。下图是将傀儡师实验板标志取模后的像素图。

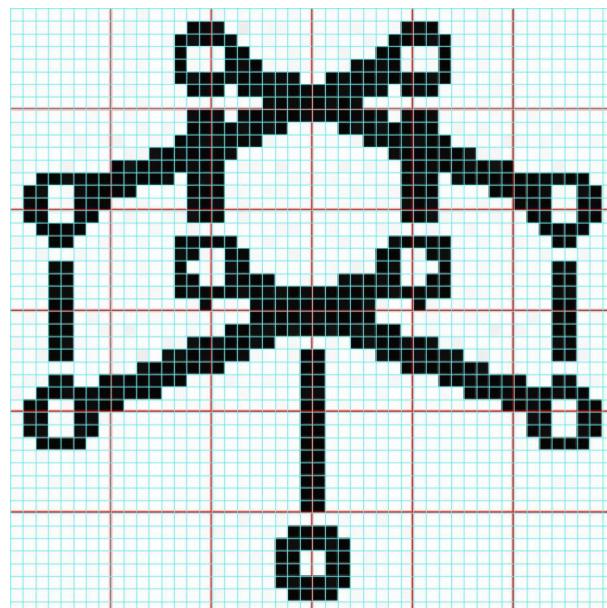
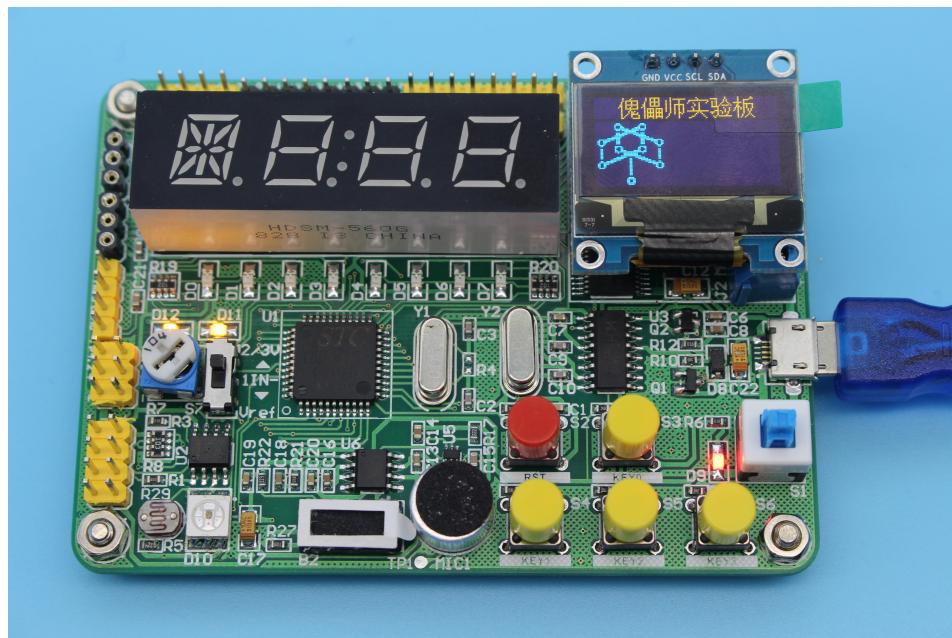


图 4.12.5 取模后的傀儡师标志

我们可以看到，我们将标志缩放成 48 * 48 大小，也就是说，它由 6 * 6 个 8 * 8 像素区域构成。和显示中文一样，遵循从上到下，从左到右，依次将每个区域打印出来。

同样为了增强函数的兼容性，函数 OLED_BMP()型参除了坐标外，还有控制图案大小的 size_x 和 size_y 以及图案数组 table。它们的功能和函数 OLED_Chn()的型参 longsize 和 table 如出一辙。有了它们，你就能方便显示你想要的图案。

12.4 下载验证



12.5 总结思考

市面上主要的 OLED，除了 IIC 控制外，还有就是 SPI 控制。受限于篇幅，我们这里就简单引入一下 SPI。

SPI (Serial Peripheral Interface)，中文串行外设接口。和 IIC 一样，他也是一种通过软件模拟的通讯协议。只要是硬件上使用 SPI 协议的从机，我们都可以通过主机在一条总线上控制它们。

但 SPI 还是和 IIC 有明显的区别的：

首先，和 IIC 一条 SDA 解决数据传输不同，SPI 有数据输入线 SDI 和数据输出线 SDO。用我们在串口那节中说过的概念，IIC 是半双工而 SPI 是全双工。

其次，SPI 有片选。片选是用来选择各个从机的，当然这不是一个线就能实现是。换言之，每增加一个从机，我们就要多用一个主机的 IO 充当片选脚。

基于 SPI 的全双工特性，SPI 的通讯速度远快于 IIC。IIC 最快也就 30Mbps，但 SPI 的传输是没有限制的。但是，当从机一旦增多，片选的增加，就会很耗费单片机的 IO 资源。可以说，两者各有优劣。

当一件事能用两个方法实现，而两个方法又各有优劣时，就免不了孰优孰劣的争论了。很多人喜欢 IIC，因为 IIC 更“优雅”——能用 4 根线解决的事情，简洁优雅，为什么要用 5 条线来解决。而且每加一个从机就要多加一个片选，多麻烦。当然支持 SPI 也有他们的理由——SPI 更简洁明了，数据传输由两条线解决，不像 IIC 那样又有什么通讯规则。要知道工程师可没那么多精力去研究其中的协议。而且相对于 SPI，IIC 的通讯速度太慢了。

所以，到底孰优孰劣并不是它们本身决定的，而是由使用者决定的。所以我们对两者都要有基本的了解，在适当的场合使用适当的方法。

结语

有志者，事竟成，破釜沉舟，百二秦关终属楚。

苦心人，天不负，卧薪尝胆，三千越甲可吞吴。

这幅耳熟能详的对联，送给能坚持走到这里的你。二十节实验能坚持下来，实属不易。但我相信，在学习中你一定收获不少。在掌握了每个模块的使用、时序图的应用、数据手册的查看和编程的思路后，校内外大大小小的比赛你都能轻松对付了。

书山有路勤为径，

学海无涯苦作舟。

韩愈这两句同样耳闻则诵，我同样送给能坚持走到这里的你。因为，只学到这些是远远不够的。因为 51 单片机是最好的入门前单片机，但却不是最实用的单片机。51 单片机确实操作简单，但根据我们从日常生活中得到的第一感觉，简单就往往意味着无法实现更负责的功能。所以在日常比赛中，我们首选能实现更多高级功能的 STM32 或能迅速实现各种设计的 Arduino。其实说白了，它们也是单片机。所以不要怕又要从头开始，而更应该趁热打铁，往更高的方向迈进。

另一方面，随着单片机学习的深入，你需要的技能也就越多。实验板上的资源是死的。以后你遇到一个项目，不可能都那一块实验板去完成。毕竟实验板上的资源都不一定是你项目所用到的。此时，你就要通过设计电路然后制作 PCB，来设计适用于项目的板子。如今机电一体化的浪潮下，很多比赛的作品都需要一定的结构。即使不需要结构，一个好看的外观都能加不少分。所以，单片机开发离不开结构。而结构的加工有很多种，而设计结构的软件又多种多样。所以，你学习的道路还有很长很长。

最后，我把我的座右铭送大家：

业精于勤荒于嬉，

行成于思毁于随。