8051单片机技术应用系列

附范例源程序光盘

8051 单片机

彻底研究 入门篇

林伸茂 编著

中国电力出版社
www.infopower.com.cn

8051单片机彻底研究 入门篇
8051单片机彻底研究 基础篇
8051单片机彻底研究 实习篇
8051单片机彻底研究 经验篇

本书全面介绍了8051单片机的入门知识。全书共分三个大部分，分别为知识建立、学习与尝试及8051的细节学习，主要包括DIY入门篇、入门工具篇、入门知识篇、8051 ASSEMBLER的认识、ASM51的操作和熟悉、8051的体系结构、8051的寻址方式、8051的指令系统、8051 Timer的认识、8051中断的认识、串行通信和如何写好8051程序。

本书选材的实用性和可操作性强，范例丰富，文字叙述清楚，是8051单片机初学者的入门指南，对8051有一定基础的读者也具有较高的参考价值，非常适合高等院校学生做实验，进行专题制作，也是研究和设计单片机产品的专业参考书，适合于广大单片机从业人员学习使用。

- DIY入门篇、工具篇、知识篇
- ASM51彻底研究
- 如何看懂Data Sheet
- 8051入门指令操作
- 8051定时、中断与串行通信彻底研究
- 如何写好8051程序
- 面包板的使用
- USB-ISP烧录器的使用
- 认识各种元件和编译器
- 认识8051的定时器
- 用定时中断更新显示
- 一个完整的串行程序

ISBN 978-7-5083-5154-4
9 787508 351544
定价：32.00元（含1CD）

责任编辑：白立军
封面设计：杜长清

8051单片机技术应用系列

8051 单片机

彻底研究 入门篇

林伸茂 编著

中国电力出版社
www.infopower.com.cn

内 容 提 要

本书全面介绍了8051单片机的入门知识。全书共分三个大部分，分别为知识建立、学习与尝试及8051的细节学习，主要包括DIY入门篇、入门工具篇、入门知识篇、8051 ASSEMBLER的认识、ASM51的操作和熟悉、8051的体系结构、8051的寻址方式、8051的指令系统、8051 Timer的认识、8051中断的认识、串行通信和如何写好8051程序。

本书选材的实用性和可操作性强，范例丰富，文字叙述清楚，是8051单片机初学者的入门指南，对8051有一定基础的读者也具有较高的参考价值，非常适合高等院校学生做实验，进行专题制作，也是研究和设计单片机产品的专业参考书，适合于广大单片机从业人员学习使用。

图书在版编目（CIP）数据

8051单片机彻底研究——入门篇 / 林伸茂编著. —北京：中国电力出版社，2007.3
（8051单片机技术应用系列）
ISBN 978-7-5083-5154-4

Ⅰ.8… Ⅱ.林… Ⅲ.单片微型计算机—基本知识 Ⅳ.TP368.1
中国版本图书馆CIP数据核字（2007）第004572号

北京市版权局著作权合同登记号 图字：01-2006-5846号

版权声明

责任编辑： 白立军
责任校对： 崔燕菊
责任印制： 李文志

丛 书 名： 8051单片机技术应用系列
书　　名： 8051单片机彻底研究——入门篇
编　　著： 林伸茂
出版发行： 中国电力出版社
地址：北京市三里河路6号　邮政编码：100044
电话：（010）68362602　传真：（010）68316497
印　　刷： 航远印刷有限公司
开本尺寸： 185 × 260　**印 张：** 18　**字 数：** 436千字
书　　号： ISBN 978-7-5083-5154-4
版　　次： 2007年5月北京第1版
印　　次： 2007年5月第1次印刷
印　　数： 0001—4000
定　　价： 32.00元（含1CD）

8051 单片机学习地图

8051 单片机的应用到处都是，可是我一点基础都没有，要怎样开始学习呢?

8051单片机
彻底研究——入门篇

我对 8051 已经有了基本的认识，我想更进一步彻底学好 8051 汇编语言!

8051单片机
彻底研究——基础篇

基本功都练扎实了，想进一步提高水平，吸取前辈的经验是最好的办法。

8051单片机
彻底研究——经验篇

要想学好 8051，练习是少不了的，我希望找一些有趣的题目自己动手做做看，不然怎样写都是一些简单的范例。

8051单片机
彻底研究——实习篇

序　言

用心，您看得见

本书的由来

长久以来我一直想写一本关于 8051 入门的书，带领对 8051 毫无经验的读者进入多采多姿的单片机世界。想学习 8051 单片机的人，应该不只限于电机电子专业领域的人们，机械专业的人学习 8051 时，他能理解某些工作用单片机来做会更有效率。控制专业的人懂了 8051 之后，会发现 8051 可以应用在某些专业的控制领域中，他只要再多知道一些 8051 的程序写法就行了。学生化科技的人一定想知道如何做自动测量和通信，而学了 8051 单片机后，就可以弥补这方面的知识断层。

我们假设本书的阅读对象是 8051 的初学者，对个人电脑的操作稍有概念，懂得如何上网和收发 E-mail，当然也喜欢自己动手 DIY。或许你曾经听说过 8051 单片机，但是不知如何下手学习，没关系，跟着本书的编排步骤走一次就可以学习到许多新的知识。如果还有空的话，请你再多看几遍，这样学习 8051 的心得会更多更广。

无论你身在何处，只要准备一台可以编写程序的计算机、电源供应器、烧录器和几枚 8051 的相关芯片就可以开始学习了。阅读本书是学习 8051 单片机的初期，应该是相当艰辛的。因为程序老是会出错，线路检查了几十次还是出错，不过不要灰心，许多初学者都与你有相同的症状和问题，多试几次就会找出问题的，婴儿学习走路不也是这样的吗。

如何阅读本书

本书的编排分为三大部分：知识建立、学习与尝试及 8051 细节学习。由于着重在初学者的入门上，8051 方面较为复杂的功能和解说都已简化或省略，但相关的重点知识仍然保留。

我们认为学习 8051 单片机绝对不是照着书本打一些范例程序，你应该按照本书建议的步骤，学习如何上网下载 8051 的汇编程序，尝试去写一个简单的 8051 程序，然后操作烧录器将程序转录到 IC 内部，最后还要用面包板连接一个测试线路，进行程序的验证。

书上的每个程序不论大小都有其意义，最好的学习方式不是将程序从光盘上复制下来再编译，我们希望你能对照书中一个字一个字输入，然后再把 8051 程序编译，从中感受程序真正的用意。

学习是要代价的

别人教你如何赚钱，别的书籍教你如何瘦身，其他的 8051 单片机除了程序范例外还是程序范例，可是本书除了教你指令的用法和如何写程序外，还教你如何进行除错，如何使用示波器看波形？

提醒你，阅读本书一定要配合 DIY 自己动手做才行。多花点时间用 DIY 的方式来学习

8051，这就是学习本书的代价。我们希望你看本书就好像经历一次快乐的学习之旅。幼教权威蒙特梭利女士曾经说过：

I hear, and I forget.

I see, and I remember.

I do, and I understand.

听到的是会忘掉的，看过的有时还会记得，可是，只要你做过的，除了不会忘记外，你将会充分地了解整个事情的来龙去脉。很多事情是做了以后，才找到解决的方法。阅读本书的同时，您的电脑应该随时开着，可以立即下载 8051 的相关资料。书上所举的例子都属入门级的，程序本身不长，最长也不会超过 100 行，如果能照着再做一次，绝对会加深学习印象，而这正是 I do, and I understand 的道理所在。

感谢

这本书从构思到整本书的完成花了一年的时间，其间我们还是如期完成多款 8051 设计项目与无刷电动机的控制设计项目，规划这本书的主要目的在于 8051 知识的传承，我们诚挚邀请更多的学习者和工程师投入认识与熟悉 8051 的行列。知识想要独享是错的，而知识服务于大众才是教育的根本，您认为呢？

以下要特别感谢曾经鼓励过我们的人们：

（1）赶工又赶文稿的旗威科技公司同仁：李浩蓁、洪健弼和周宜瑱。

（2）希望能够准时交货的胜光科技公司：许锡铭和王裕进先生。

（3）对我们期望极高的诚岱机械公司：赵福安先生。

（4）指导我们芯片设计技术的应广科技公司：凌全伯先生。

（5）对我们指导和协助的工研院能环所：张钰炯、杨锃忠和罗志忠先生。

（6）十年来对我们帮助不断的擎宏电子公司：林睦钧先生。

（7）对我们出书绝对支持的旗标出版公司：施威铭先生和陈宗贤先生。

当然还要加上家人的鼓舞与小女儿佳婕的适时捣蛋。

林伸茂

旗威科技有限公司

chipware@chipware.com.tw

阅读时建议配合工具（*为必备工具）

书籍或文件部分

8051 单片机彻底研究——基础篇
8051 单片机彻底研究——实习篇
8051 单片机彻底研究——经验篇
AT89C2051 或 AT789S52 Data Sheet（*）
ASM51 编译程序操作手册（*）

硬件部分

PC 个人电脑，可上网及编辑程序（*）
AT89C2051 或 AT89S52 芯片（*）
芯片专用烧录器（*）
实验用面包板（*）
+5V 电源供应器（*）
数字式电表（*）
2CH 数字式示波器
逻辑笔

个人心态部分

一定自己 DIY（*）
自己的程序当然自己除错（*）
学习的事不假手他人（*）
除非万不得已，不会把问题转给别人（*）

版权声明

本书内所引用国内外的产品、画面和网页都是善意的，其中包括：

（1）Aglient 为 Agilent Technologies（安捷伦科技）的注册商标。

（2）ASM51.EXE 为 MetaLink Corporation 的注册商标。

注意：Metalink 没有为该产品提供技术支持和保证。

（3）Atmel 为 Atmel Corporation 的注册商标。

（4）CIRRUS LOGIC 为 Cirrus Logic 的注册商标。

（5）FLUKE 为 Fluke Corporation（福禄克公司）的注册商标。

（6）HIOKI 为 HIOKI E.E.CORPORATION（日置电机株式会社）的注册商标。

（7）IDRC 为 CHYNG HONG ELECTRONIC CO.（擎宏电子企业有限公司）的注册商标。

（8）Intel 为 Intel Corporation 的注册商标。

（9）IWATSU 为 IWATSU ELECTRIC CO.（岩崎通讯机株式会社）的注册商标。

（10）Tektronix 为 Tektronix，Inc.（太克科技）的注册商标。

（11）TI 为 Texas Instruments Incorporated.（德州仪器公司）的注册商标。

（12）TOPWARD 为 Topward Electric Instruments Co.的注册商标。

（13）Windows、MS DOS、Microsoft 记事本、Microsoft Internet Explorer 为 Microsoft Corp. 的注册商标。

以上均属于其合法注册公司所有，本书仅用于说明解释，无任何侵犯意图，特此声明。

关于光盘

感谢您购买本书，此份光盘里面收录书中使用的程序代码，每个程序按照所属章节排列并且存放在文件夹内，如需打开.ASM 的文件，请使用 Windows XP 的记事本或其他文字编辑器打开。光盘内的.TSK 文件为烧录用的二进制文件，因此，打开后产生乱码是很正常的，请勿担心。

除了书中程序代码之外，光盘内还收录了 8051 的四则运算、常用指令、存储器规划和 HEX 的介绍等 8051 常识共 37 篇，其文件格式为 Word。这些课外补充的资料都是学习 8051 不可或缺的，请大家一定要仔细阅读。

如果无法打开.doc 的文件，请自行到 http：//www.adobe.com/网站下载 Acrobat Reader 文件浏览器。

无论您对书中的内容有所疑问或是学习 8051 时遇到问题，都欢迎您到旗威论坛（http://chipware.myvnc.com/phpbb/）与我们分享，最后，祝大家学习顺利。谢谢！

旗威科技有限公司

Chipware System Inc.

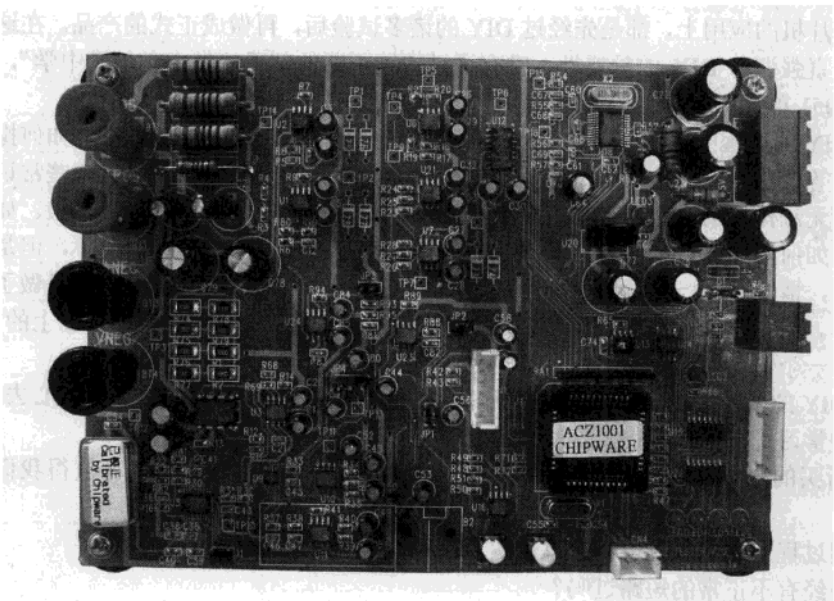
网站：http://www.chipware.tom.tw

论坛：http://chipware.myvnc.com/phpbb/

信箱：chiwpare@chipware.com.tw

目　录

1 DIY 入门篇



每每谈到 DIY，就会想起那段三更半夜不睡觉，画电路图然后想把电路板焊好的日子。经历一个电子作品从无到有的 DIY 过程，是学习电子电路最重要的一环，当作品完成的那一刻，看着所有操作都如自己所规划般地进行时，那种喜悦是无法言喻的。希望通过本章的介绍，能带您一同进入 DIY 的世界，享受 DIY 的真正乐趣。

第 1 章　DIY 入门篇

1.1　自己 DIY

欢迎来到自己动手做 DIY 的世界。

在许多单片机的应用上，都是先经过 DIY 的诸多试验后，再做成正式的产品。在这本书中，我们会一直强调着 DIY 的重要性，唯有通过持续不断的动手“做中学与错中学”，才会深刻地理解 8051 单片机的所有相关的知识和技巧。

在自己 DIY 的过程中，我们会学习到如何看电子元件、如何用电烙铁焊接，如何操作简单的烧录器等等。这些操作技巧在稍后章节的学习上，都会派得上用场，而且有些常识或技巧是课堂上所无法传授的。例如，如何买电子元件，如何看元件，如何做硬件实验，如何上网找资料以及如何用网络订单的方式买到合适的元器件等等。这些过程看似简单，但是轮到你真正面对时，情况就不一样了，所谓不经一事不长一智。很多事情的解决方法是做了之后才找到答案的。如果您不做那就永远与正确的解决方法无缘了。做了就找到一个以上的方法，不做只是多了一个借口而已。

在我们 DIY 的学习过程中，会遭遇到很多不顺心的事情，但是请一定要以平常心去对待，因为没有一种学习是一帆风顺的。

例如，新买的电子元件在实验时突然冒烟损坏了，那一定有许多事情是值得我们探讨的：

（1）使用过程当中电源的电压超过额定电压吗？

（2）你曾经有不正常的短路过吗？

（3）你写的程序本身有问题吗？

（4）你的电脑有问题吗？

（5）你买到的元件是次品吗？

（6）线路接法有问题吗？

（7）参考的电路画错了吗？

DIY 可以做任何电子电路，图 1-1 为速度达 60MHz 的 DIY 数字电路。

我们可以用证明法或逐项消去法去分析所有的问题，最后找到问题的症结所在，而这正是 DIY 最大的收获。以后若再碰到类似的问题就很容易排除困难了，学校没有教到的，本书教导你直到会为止，而我们唯一的要求只是请一定要亲自 DIY 动手做。

图 1-2 所示 DIY 电路板背后是密密麻麻的镀银线，这时细心是很重要的。

模拟电路的 DIY 板（见图 1-3），感觉比较空，但 DIY 模拟电路要留意的因素更多。

在制作烧录器之前，我们也曾经 DIY 先进行烧录波形的确认，如图 1-4 所示。

图 1-5 所示是 8051 无刷电动机控制器的量产品，事先也经过 DIY 的确认后才进行生产。

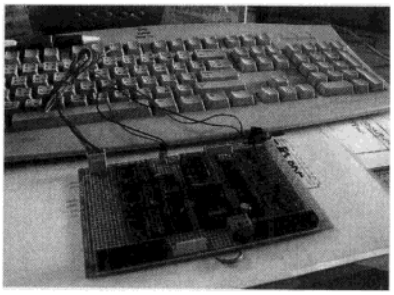


图 1-1　DIY 数字电路

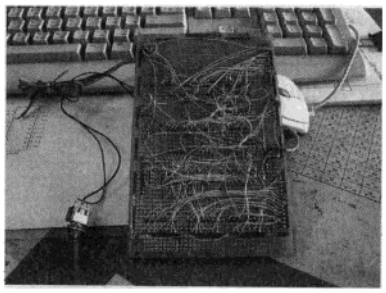


图 1-2　电路板后的线

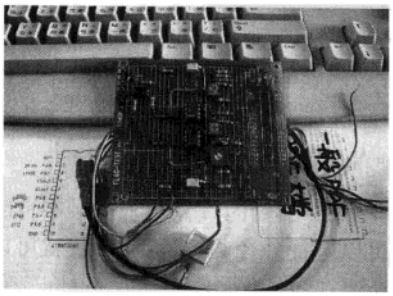


图 1-3　模拟电路的 DIY 板

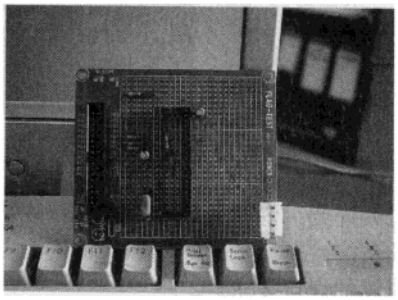


图 1-4　烧录器

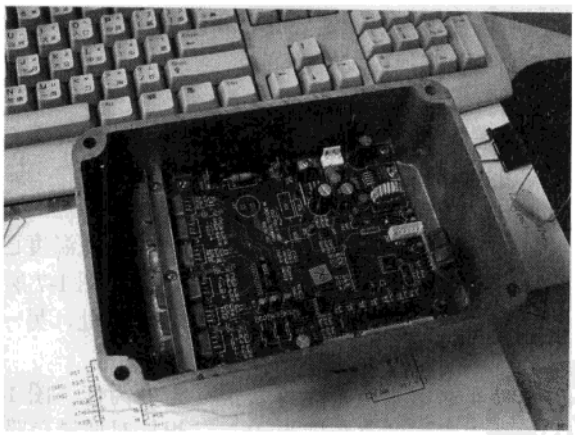


图 1-5　无刷电动机控制器

1.2　焊　　接

一般来说，在 DIY 的过程中，我们会先利用面包板或免焊电路板进行实验，以验证线路的正确性，当线路功能确认无误后，接下来就是将元件焊接在电路板上，因此，焊接是 DIY 必备的重要技能，下面就要来学习如何焊接电路板，学习前要先准备焊接必备的两项工具：

（1）烙铁：一般 DIY 用的电烙铁选择以 30～40W 为主，家里用的话就选用 220V 规格的。烙铁的价格起伏很大（几十元到几百元不等），初次 DIY 的话可以挑选约 50 元上下的电烙铁且瓦特数为 40W 即可，再加上一个烙铁架约 100 元就可以了。

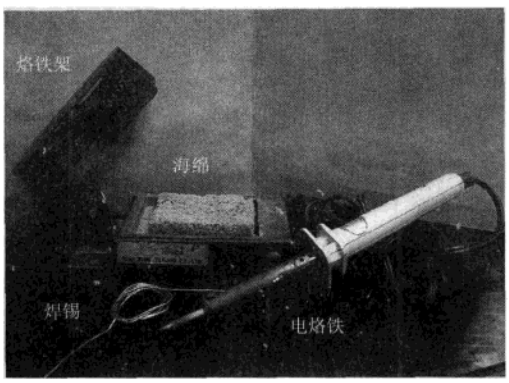


图 1-6　电烙铁、烙铁架和焊锡

（2）焊锡：焊锡可以将电子元件连接到电路板上，减少引脚因氧化导致接触不良或电阻过大的情况。一般的焊锡其锡铅比例为 68:32，我们可以选择外径约 0.8mm 锡线就可以满足一般电路的焊接要求。

工具备妥后，就可以准备开始焊接。焊接前要注意烙铁头的温度会高达 300℃以上，操作时一定要专心并避免小孩碰触。焊锡熔解时的烟雾与电子元件引脚上的化学溶剂残留，都可能会造成身体的伤害，所以在焊接时一定要专心和注意，而且环境一定要通风，烙铁周围避免放置塑胶物品、易燃物品及其他挥发性溶剂。

接下来我们以烙铁来进行焊接操作：

（1）焊接前需先把烙铁插电预热，将烙铁架上的海绵先泡水弄湿，并且准备好要焊接的电子元件和电路板，减少边找元件边焊接的情况。

（2）预热三分钟后即可开始焊接，此时的烙铁头应该是亮晶晶的。焊接前可以先用焊锡接触烙铁前端的烙铁头部分，焊锡若可顺利完全熔化，代表烙铁温度已经足够。烙铁温度到达时，其头部包着一层薄薄的焊锡，且看起来是亮晶晶的，如图 1-7 所示。

（3）烙铁的握法以顺手为主，建议以握笔的方式握着手柄处，另一手拿焊锡，手不可太靠近焊锡线的前端。

（4）开始焊接时，烙铁头和电路板的夹角以 45° 左右为佳。图 1-8 中烙铁头、元件引脚和电路板上欲焊接的元件引脚间的夹角即为 45°，焊接时要避免吸入焊锡熔化时所产生的烟雾。

（5）焊接时先将电子元件引脚折好置于电路板上，再把电路板翻面压好，焊接元件的顺序应该从较矮的电子元件先焊接。

（6）我们先用烙铁头同时接触元件引脚和铜箔，即两方面同时加热，再将焊锡接触烙铁头，焊锡会立即熔解并扩展到元件引脚和铜箔上，这时可以慢慢移开烙铁，让焊锡自然凝固几秒钟，未凝固前不可移动上述元件。

（7）焊好的接点从侧面看应该是呈内弧线的圆柱状焊点，如果你的焊点是球状的，那焊

点很可能内部是中空的，会形成所谓的假焊或虚焊状态，导致电路不通或导电不良，如果出现这样的焊点，请一定重焊。图 1-9 中左边是好的焊点，样子是内凹弧线的圆柱状，右边焊点则为虚焊的球状焊点。

（8）焊接二极管、IC、三极管、电容、SMD 等电子元件时，请不要将烙铁接触引脚超过五秒钟，否则很可能会造成元件内的线路烧毁或短路。

（9）若发现有多余的焊锡聚在烙铁头上，可将烙铁头轻刷湿海绵，以去除多余的锡，在烙铁头上待过久的锡已经算氧化了，适度的舍弃才是正确的做法。

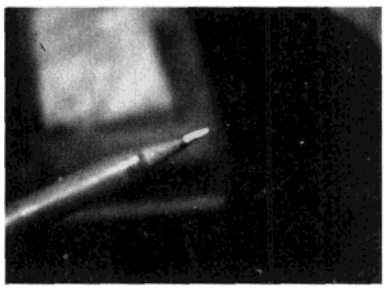


图 1-7　亮晶晶的烙铁头部

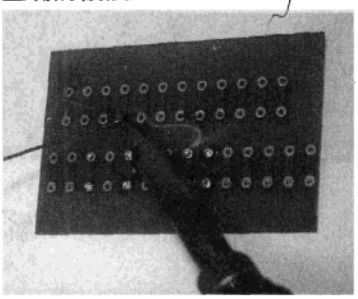


图 1-8　烙铁的用法

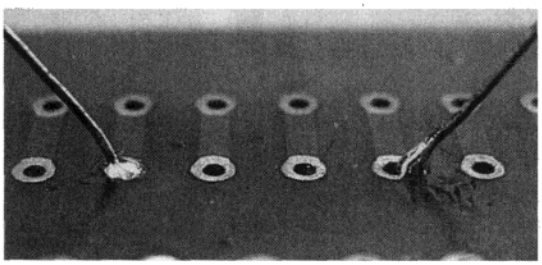


图 1-9　焊点的好坏

1.3　洞洞板的使用

除了使用面包板可以验证电路外，我们也可以用洞洞板来做电子的 DIY 实验，洞洞板的外观如图 1-10 所示。洞洞板在使用上没有面包板那么方便，而且元件与元件间要另外用焊锡焊接在一起，所以必须要事先熟悉烙铁的相关操作。洞洞板虽然比较不方便，但是元件经过焊接后会非常牢固，不会有面包板上元件容易松动的缺点。

请注意边上有些铜箔是相连的，这是用来做电源与 GND 点的连接的。

我们把洞洞板的正面称为元件面（Component Side），所有的元件都插在此面上。而布满铜箔供焊

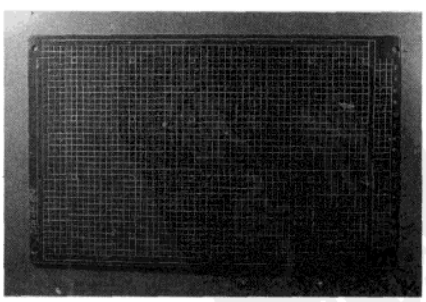
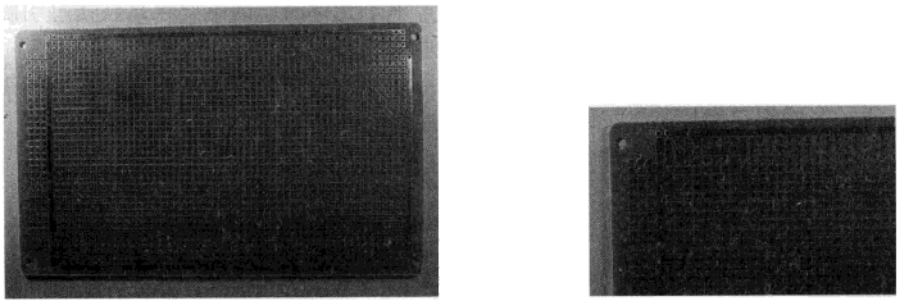


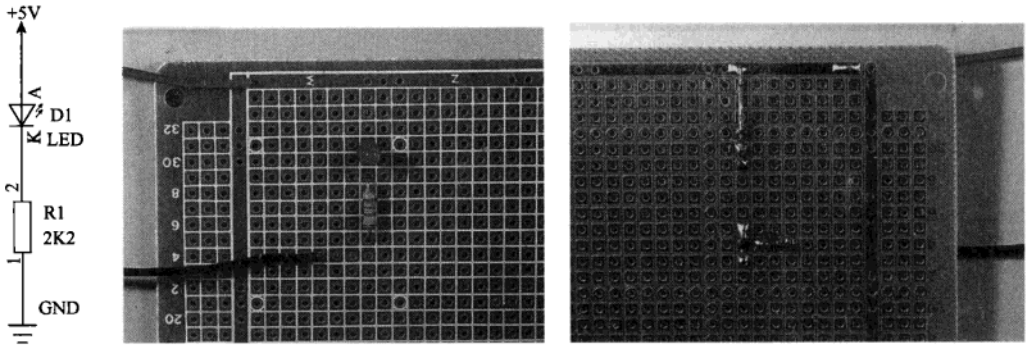
图 1-10　洞洞板的正面图，看起来果然全都是洞洞

接用途的背面称为焊接面（Solder Side），仔细观察焊接面上的铜箔接点安排还是有点不同，有些接点是圆形的，有些是方形的，有些则是一长串相连着，其主要用途是做系统的电源或 GND 地点的多点连接。



（a）　　（b）

图 1-11　洞洞板的背面



（a）+5V 供电端我们习惯用红色线，GND 端则用黑色线　　（b）

图 1-12　LED 点亮的电路图改用洞洞板来接线，下方为焊接面的连接情形

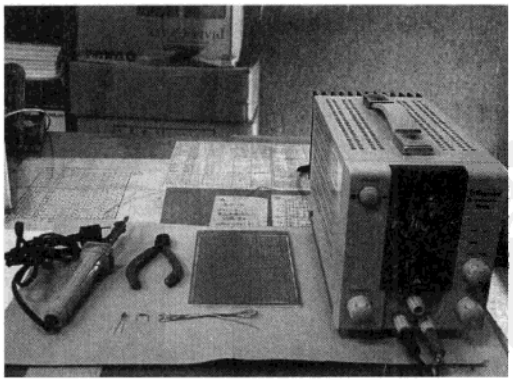


图 1-13　DIY 电路板时必备工具

洞洞板在 DIY 时，要准备的元件和工具：洞洞板、LED、2K2 电阻、电源供应器、斜口钳、烙铁与少许焊锡丝。

洞洞板上元件的配置可以几乎跟电路完全相同，最上面供应+5V 电压，经过 LED 和 2.2kΩ 电阻后到达地端。整体看起来几乎是原电路图的翻版，图 1-14 为焊接面的连接情况，焊接原则是用最少的接点将元件连接焊接在一起。DIY 焊接完后，第一项操作是先以目测的方式检查元件面与焊接面，看看是否电路有接错的情形。第二项操作用电表的电阻挡检查+5V 和 GND 间的电阻值。无论如何，其阻值都不该在 1Ω 以下。为什么呢？根据 $V=IR$ 公式来看，供应电压是 5V，若阻值是 1Ω 的话，通过的电流就有 $I=V/R=5/1=5A$ 之多，而不是原先我们计算的 10mA 以下。第三项操作才是由电源供应器上加入+5V 与 GND 到洞洞板上。

经过层层检查后，就可以通电测试了，当 LED 亮的时候，你就会知道焊接的整个电路是正确的，恭喜你做到了！而这正是 DIY 的真正精神所在。

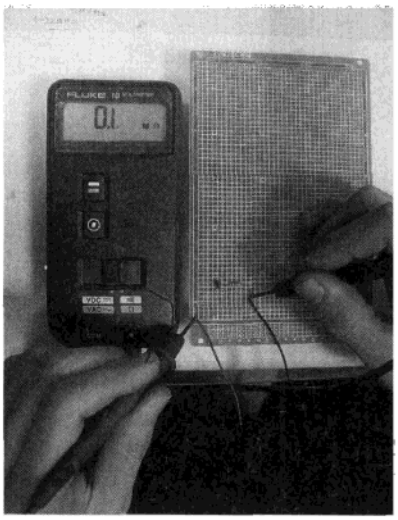


图 1-14　用数字电表检查电源与地端的阻值，电表显示阻值几乎无穷大，代表没有短路情形

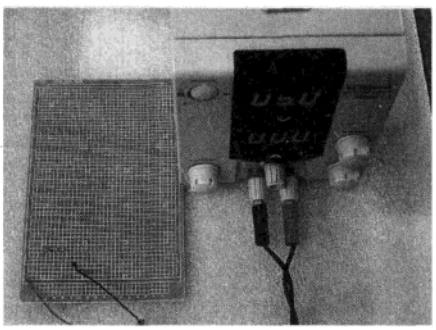


图 1-15　电路 DIY 过程

虽然只是一个简单的电路，经过焊接、目测和电阻值确认，最后再接上+5V 电源。LED 终于亮了，你的 DIY 技术也更上一层楼了。

1.4　面包板的使用

在拿到一个线路图后，总是想实际买元件回来 DIY 一下，可是元件买回来了，如果不焊接的话，还可以实验吗？答案是肯定的，你可以用面包板配合元件来模拟我们手边的线路图，不过线路图上的元件符号，该怎么对应到面包板上？面包板上那么多洞洞，到底要摆在哪里？又怎么摆上去才是对的呢？

我们先认识一下面包板，面包板主要分成两个部分：

第一部分：正负极电源配置端

面包板上下都有一个提供正负电源的地方。如果以图 1-16 实验用的面包板来看，两旁各有标示“+”和“-”两个符号，就是面包板上规划出来给正负极的，此部分整个横向都是相通的。

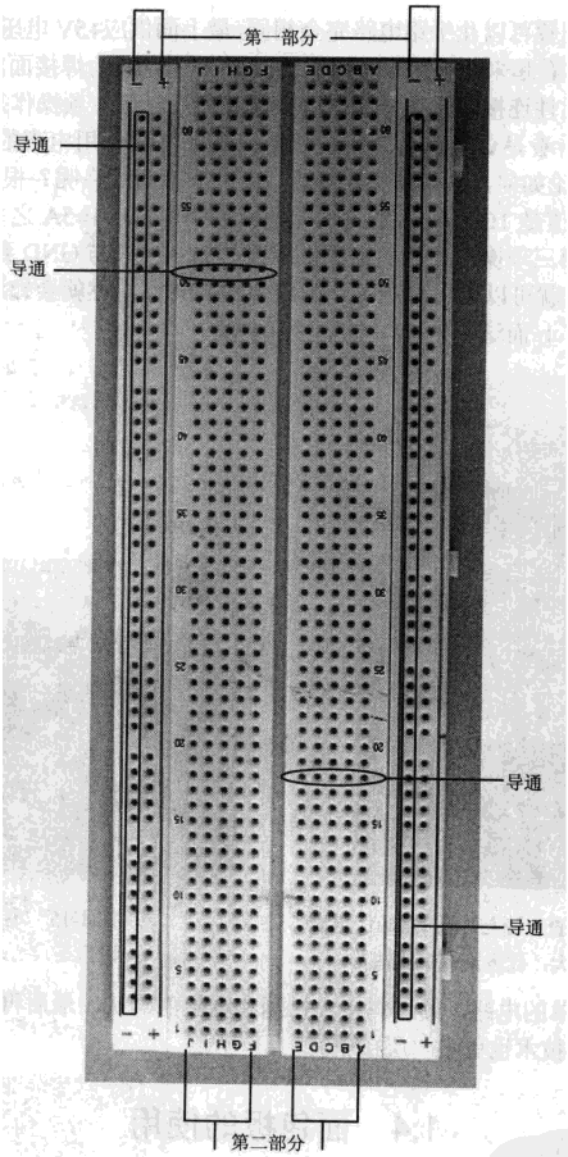


图 1-16　实验用的面包板

图 1-16 是实验用的面包板，上面的洞洞就是要插元件用的，不过它可是有一定的规则在，不能随意乱配置的，否则不仅是实验不会成功，连元件都可能会因短路而烧坏。

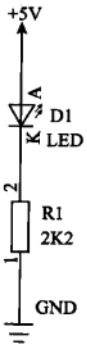
第二部分：一般导通配置

面包板上还有标有符号 A～J 的。这里提供一般元件互相导通的区块，不过 A～E 为一区，F～J 为另一区，两边是没有互通的，所以中间隔了一个凹槽，这两区则依 A～E 和 F～J 横向做导通。

知道面包板上设计导通的方向后，就开始我们的第一个 LED 亮灯的实验吧！在配置元件

前，我们要先决定正负电压进来的位置，正电压由面包板右方进入，地端（GND）则用左方标示“－”端进入。

接下来，清点一下线路图上的元件。我们需要准备的元件有 LED 灯、2.2kΩ 电阻和一些跳线，最后我们需要一个+5V 的电源，接下来就可以实际对照线路图来接线了。



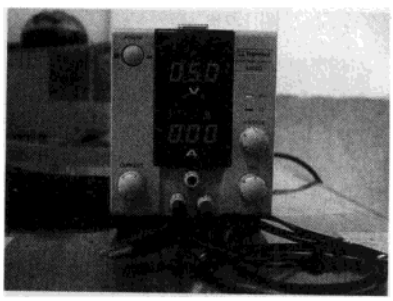
线路和元件的详细说明

（1）LED 直径 3mm 或 5mm 的都可以，颜色不定，但是红色 LED 最容易见到，所以建议选用红色 LED。LED 是有极性的，A 端为正端，引脚较长，K 端为负端

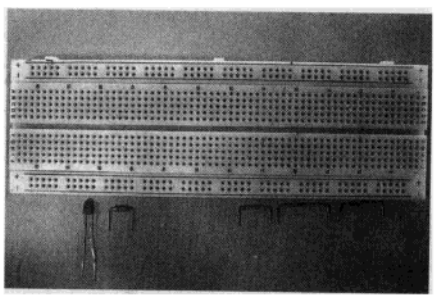
（2）电阻为 1/4W 的 2200Ω 电阻，阻值正确的标示方式为 2.2k（欧姆通常不标），不过在印刷时，小数点有时候会变得很模糊或很难辨认，所以，在实际上我习惯标成 2K2

（3）这个小实验需要+5V 的电源供应器，在 DIY 的场合中,我们建议用可变电压型的电源，并且有限流保证，以免发生错误

图 1-17　一个点亮 LED 的示范电路图，你要准备的元件有：
直径 3mm 的红色 LED、电阻 2.2kΩ 与有+5V 输出的电源供应器

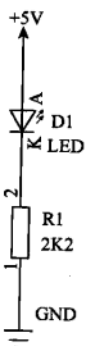


(a)



(b)

图 1-18　准备好面包板及相关元件，准备开始进行实验



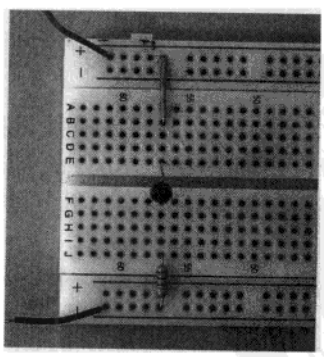


图 1-19　接法一

如果电路图是经过好的规划时，我们用面包板的接线模式几乎可以与电路图完全相同，如图 1-19 所示。接下来示范其他方式的接法。

如图 1-20 所示，由电源正极进来后会接到 LED 灯，这时要注意 LED 引脚有正负极的区别，所以 LED 灯正极接到电源正极，负极是接 2.2kΩ 电阻，电阻没有正负极的分别，所以一端与 LED 灯负引脚连接；另一端则接地端。

如图 1-21 所示，LED 灯的正极使用跳线再接到电源正极上，负极一样接 2.2kΩ 电阻，但 2.2kΩ 电阻的另一端也由跳线与地端连接。图 1-21 中圈起来的地方都是导通的。

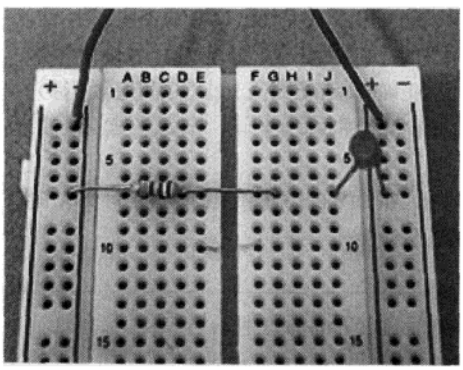


图 1-20　接法二

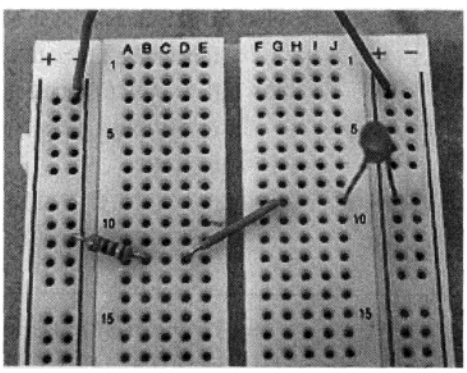


图 1-21　接法三

如图 1-22 所示，LED 正极接到正电源正极，负极由跳线与 2.2kΩ 电阻连接，电阻的另一端直接接地端。

看了上面的四种接法，我们会发现同一个线路图其实有多种接法，LED 灯也都会亮，这是因为只要把握住面包板上互适的原则，就可以顺利地让电路工作。你的接线方式当然可以和这里示范的不同，不过只要遵循导通的原则，要怎么变化都是可以的。

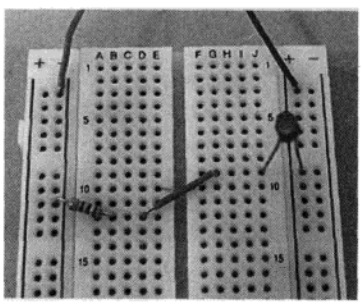


图 1-22　接法四

1.5　面包板与 *V=IR* 的验证

会使用面包板后，我们现在要试着用面包板来验证 *V=IR* 公式。我们一样供应+5V，然后再使用一个 4.7kΩ 和一个 10kΩ 的电阻，这两个电阻的误差都是 5%，线路图的接法如图 1-23 所示。现在要试着计算两个电阻中间的分压电压值为多少呢？

先列出公式来：$V=I \times R$

$$5V=I\times(10+4.7)\times10^{3}$$

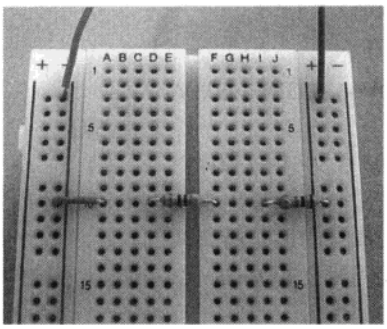
所以 $I=0.34013mA$

两个电阻中间的电压值 V 可以用两种方法计算出：

由 0V 电压算起 $0V+IR_{(R2)}=0V+(0.34013mA\times4.7\times10^{3})=0V+1.598V=1.598V$

由正端电压反算 $5V-I_{(R1)}R_{(R1)}=5V-(0.34013mA\times10\times10^{3})=5V-3.4013V=1.599V$

计算好了再将这些元件实际用在面包板上，再用电表测量，看答案和我们计算的是否会一样？



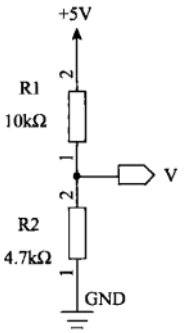


图 1-23　+5V 进来后先接一个 10kΩ 的电阻（R1），另一端接一个 4.7kΩ 的电阻（R2），由 4.7kΩ 的电阻接地端

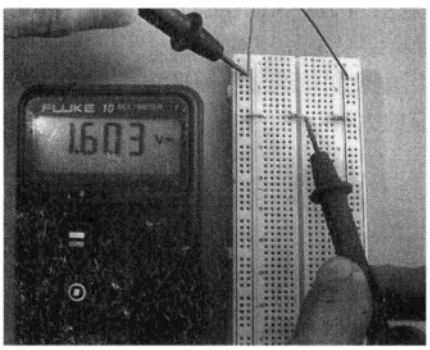


图 1-24　电阻值测量

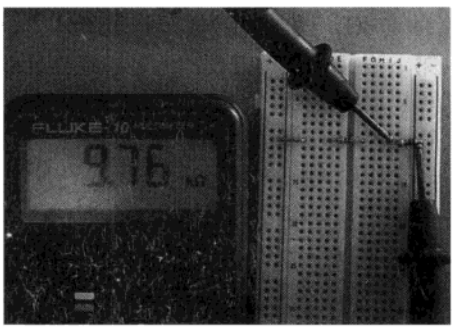


图 1-25　用电表实际量测 10kΩ 的电阻，测量结果为 9.76kΩ

使用电表测量通过两个电阻间的电压值，测棒正端接 10kΩ 接正极的引脚，负端接地端，电表实际测量的电压值为 1.603V，和我们算出的 1.598V 电压值相差 0.005V。

两个电阻的分压值是 1.603V，而套用公式所计算出来的结果是 1.598V，大约有 0.005V 的误差值，是不是电阻的误差造成的呢？我们使用的电阻本身就有±5%的误差，所以我们尝试用电表直接测量实际的电阻值后，重新再计算一次，看看问题是不是出在电阻的误差上面。10kΩ 电阻量测出来的值是 9.76kΩ 而 4.7kΩ 电阻量测出来是 4.61kΩ，两个电阻值都在±5%范围内。

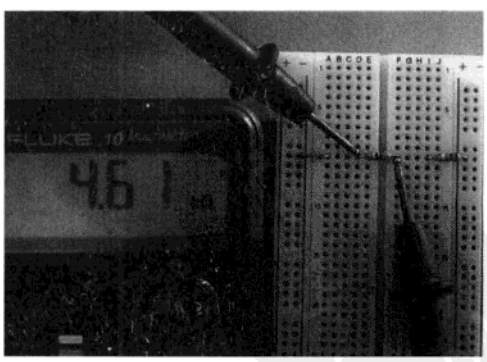


图 1-26　用电表实际测量 4.7kΩ 的电阻，测量结果为 4.61kΩ

我们用测量出来的电阻值再计算一次，一样套用的公式：

$$5V=I\ (9.76+4.61)\times10^3$$

$$I=0.3479mA$$

两个电阻间的分压值（V）：

由地端电压算起

$0V+I_{(R2)}R_{(R2)}=0V+(0.3479mA\times4.61\times10^3)=0V+1.603V=1.603V$

由正端电压反算

$5V-I_{(R1)}R_{(R1)}=5V-(0.3479mA\times9.76\times10^3)=5V-3.396V=1.604V$

计算时虽会有小数点进位的误差，但是这个数据和实际测量出来的电压值已经相当接近了，因此在前一页计算时所得到的误差，应该是电阻本身的误差所造成的，不过这个误差值其实很小，而且是可以忽略的。经由这个例子，让我们对公式 *V=IR* 又有更进一步的认识。

1.6 LED 限流电阻的选用

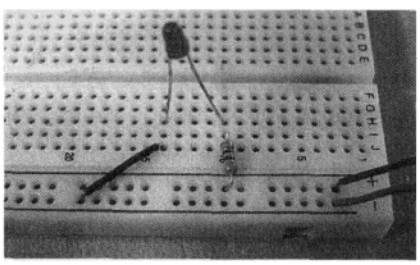
下图元件的接法，是要让一个 LED 灯亮，但是我们要挑选阻值不同的电阻来做实验：电阻大小对 LED 灯会造成什么样的影响？我们先选用 4.7kΩ（4700Ω）做实验，而另一次实验为 470Ω，两个电阻值相差 10 倍，拿这两组实验结果来互相对照。

第一次实验我们先使用 4.7kΩ 的电阻，并且使用电表测量通过电阻的电压为 3.179V，这样就可以利用 *V=IR* 的公式求出流经 4.7kΩ 电阻电流的大小：

$V=I\times R$

$3.179V=I\times4.7\times10^3$

$I=3.179V/4.7k\Omega=0.676mA$



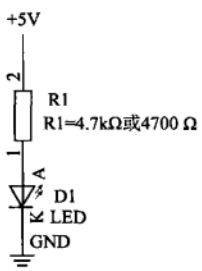


图 1-27　电阻接法

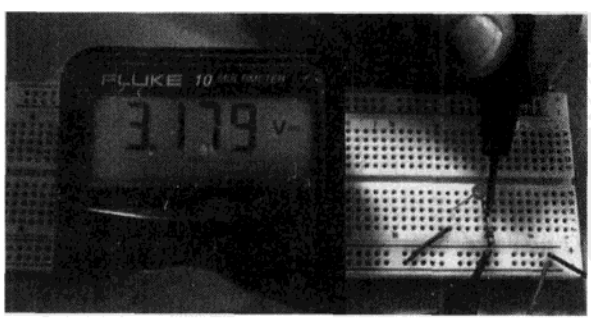


图 1-28　电阻的测量

接下来我们将电阻换成 470Ω，通电后发现 LED 灯的亮度比第一次使用 4.7kΩ 电阻亮，同样测量通过电阻的电压为 3.039V，所以，流经 470Ω 电阻的电流为：

电源供应+5V，通过电阻接其中一引脚，电阻另一引脚接 LED 灯正引脚，LED 灯负引脚以跳线接地端，其中的电阻分别使用 4.7kΩ 和 470Ω。

先使用 4.7kΩ 电阻来做实验，并且用电表测量通过电阻两端的电压值，测棒的两端分别接电阻的两端，测量到的电压值为 3.179V。

$V=I \times R$

$3.039V=I \times 470$

$I=3.039V/470\Omega=0.00646\ A=6.46\ mA$

图 1-29　电阻的测量

用电表测量 470Ω 电阻实际的电阻值，在不通+5V 时代表正负极的测棒分别放在电阻两端，测量出的电阻值为 0.467kΩ=467Ω。

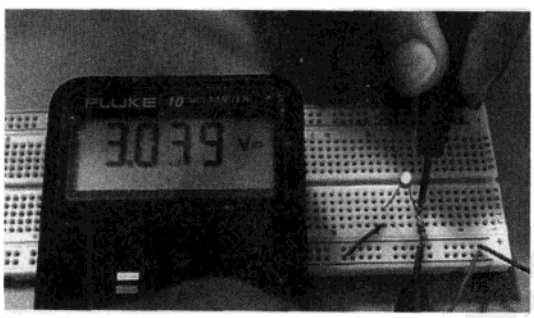


图 1-30　使用 470Ω 电阻并接上+5V 电压后，用电表测量电阻两端的电压为 3.039V

通过计算结果可知，通过 4.7kΩ 电阻的电流为 0.676mA，而通过 470Ω 电阻的电流为 6.46mA，可见电阻越大时电流会越小，反之电阻越小电流会越大，而且使用 470Ω 较小电阻的 LED 灯通电后，比使用 4.7kΩ 电阻的亮度高，但亮度看起来并没有原先的 10 倍亮。

这时我们得到两个结论：

（1）电阻与电流成反比：即电阻越大通过的电流就越小。

（2）电流越大 LED 灯的亮度越高。那是不是不要使用电阻会更好呢？

不是的，一般的 LED 可通过的电流约在 10mA 左右，其两端可承受的最大直流电压约在 2～3V 左右。如果完全不加电阻保护的话，那么 LED 灯应该很快就会承受不了大电流而损坏，其他的元件也是相同的道理，所以加上电阻做适度的限流是有必要的。

您可以从下列公司的网站取得更进一步的信息：

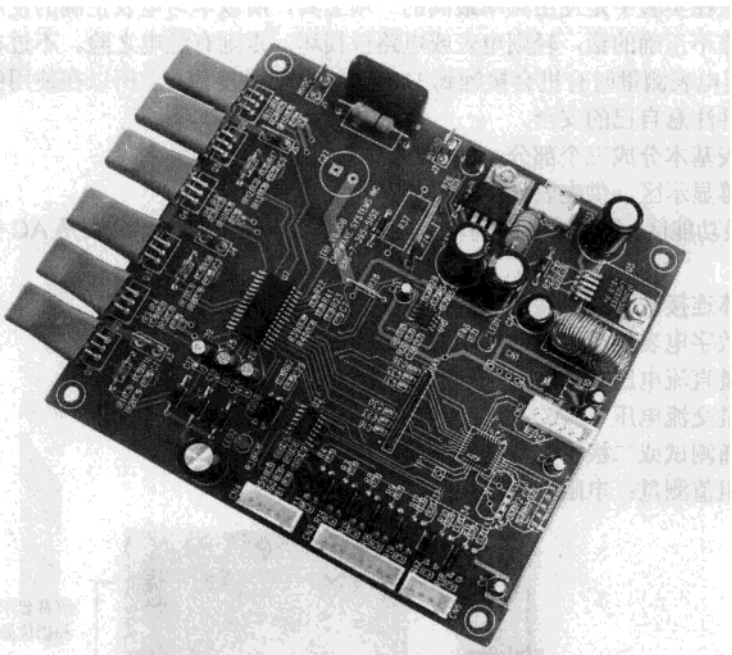
（1）www.chipware.com.tw：查询单片机相关应用与参考资料。

（2）www.fluke.com：查询掌上型数字电表的相关资料。

（3）www.hakko.com：查询焊接工具的相关资料。

（4）www.topward.com：查询电源供应器的相关资料。

2 入门工具篇



“工欲善其事，必先利其器。”这是古人留传下来的一句很受用的话，这句话的意思就是要告诉我们：如果要把工作做得尽善尽美，就要好好利用工具。在学习 8051 的过程中，有一必要的工具是我们必须先掌握的，在本章的内容里，我们将一一介绍这些常用工具的使用技巧和限制。学习使用这些工具，能让您的学习更系统化、效率化。

第 2 章　入门工具篇

2.1　数字电表操作示范

数字电表在实验中是使用频率最高的一项工具，所以学习电表正确的使用方法是很重要的。如果操作不正确的话，轻则电表或电路板损坏，重则有触电之险。不过在这里先提醒大家，因为使用电表测量时有机会接触到 110V 或 220V 的交流电，所以在使用的时候一定要用正确的方法并注意自己的安全。

数字电表基本分成三个部分：

（1）屏幕显示区，供电表显示测量值。

（2）切换功能区，图 2-1 中的基本型电表提供四种测量功能，分别是 AC 交流、DC 直流、电阻与导通。

（3）测棒连接区，红色接正端（+），黑色接地（COM）。

图中的数字电表能测量的四项功能分别为：

（1）测量直流电压（VDC）：直流电测量。

（2）测量交流电压（VAC）：交流电测量。

（3）导通测试或二极管测量：导通或二极管测量。

（4）电阻值测量：电阻值测量。

图 2-1　一台典型的数字万用电表

1. 交流电压的测量

AC 交流电压的测试该如何做呢？一般我们从墙壁上的电源插座取得的电为交流电（或称市电），测量第一步要做的就是将电表切换成测量交流电（VAC）的状态，千万不要切换错误！不然电表可是会因挡位不对而坏掉。

确定电表设定 VAC 后，把两支测棒接触 AC 插座内的金属部位，此时一定要注意手不要碰到任何金属部分，否则会有触电危险，不到半秒屏幕上显示出交流电压值，这就是电力公司供应给家用的交流电压值。如果多花一点时间观察该电压时，你会发觉这个值是变动的。有些更高级的电表还会将频率值显示出来，这个值会影响家中某些电动机的转动速度。

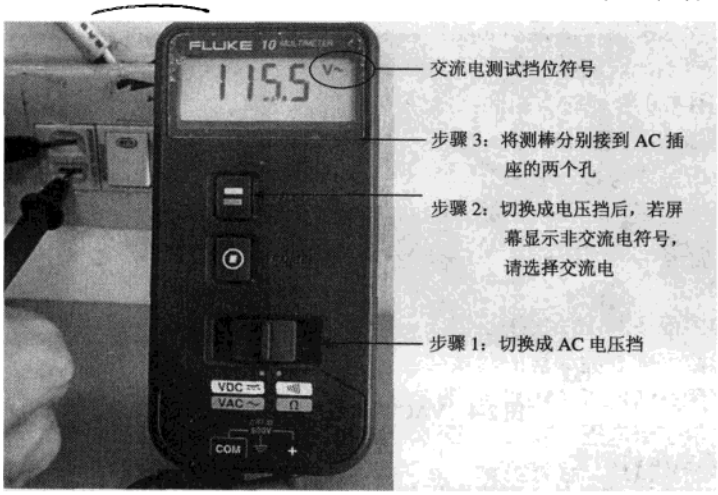
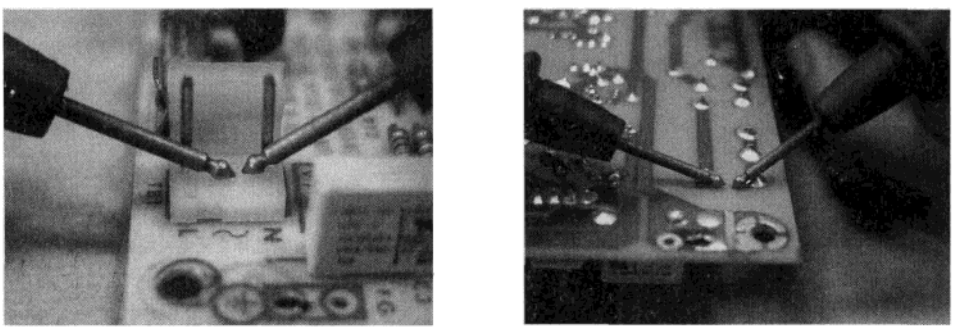


图 2-2　使用万用电表测量交流电压的步骤

图 2-3　图中的电表除了显示电压值外，还多了电压的频率测量，这是针对不同的实验，选择的电表功能也有所增加

进行交流电压值测量时，请一定要注意：不能让两支测棒有短路的情况发生，所谓的短路就如同在 AC 电源端上并联一个很小的电阻，如果 AC 电压为 110V，照公式 $I=V/R$ 来看，当 V 很大而 R 很小时，换算后的电流是很大的。所以短路的瞬间会伴随很大的电流和火花，足以将测试棒的尖端熔化，在 AC 端短路是非常非常危险的行为，操作数字电表测量时一定要留意并避免触电。

实际上如何做呢？我们的建议是这样的：测量时一次只能移动一支测试棒，尽量避免两支测试棒同时移动，当两支测试棒的距离少于 2cm 以内时，就要特别留意了，因为稍微的移动就有测棒短路的情况发生。



(a)　　(b)

图 2-4　VAC 挡要避免测量时的不正常短路

2. 直流电压的测量

除了交流电压测量外，直流电压值的检查也是很常见的。在 DIY 实验时，我们经常会将数字电表切换到 VDC 挡，以便检查电源供应器的电压是否正确，单片机 8051 的供应电压值是多少，RST 重置脚的电压电平判定等等，还有观察电阻上的电压降，进而推算出通过的电流值。

在做更进一步的 AD 模拟和数字实验时，也要用数字电表的 VDC 挡做测量与确认。当测量的 VDC 电压值超过 1V 以上时，在进行测量时，请特别注意不要让两支测棒有短路的可能，VDC 挡上测棒瞬间的短路并不会损坏电表，但是会因通过大电流而损坏测棒。

3. 电阻与导通的测量

我们在 DIY 实验时，数字电表的电阻挡可以用来做许多基本的检查：

（1）电阻值是否正确，DIY 时有时候会看错阻值，例如 10kΩ 看成 1kΩ。

（2）电阻值误差的判定。

（3）线路是否正确连接，正确时其电阻值应该几乎为 0。

（4）电源端是否短路的判断，若短路时会测量到一个非常低的电阻值。

（5）二极管好坏的判断。

4. 选购合适的数字电表

DIY 用的数字电表应该以耐用、保护性好、精度高和价格几项作为购买时的依据，并且尽量选购有自动换挡（Auto-Range）的数字电表，以免因设定的挡位不正确而损坏电表。数字电表的价格在 100～5000 元之间，主要的差异还是在精确度和电路保护上，在这方面绝对是一分钱一分货的。

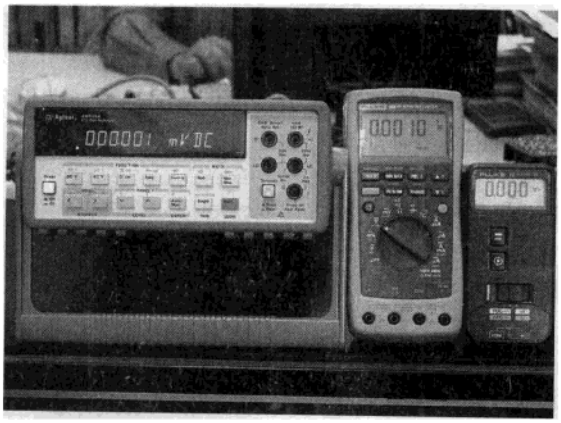
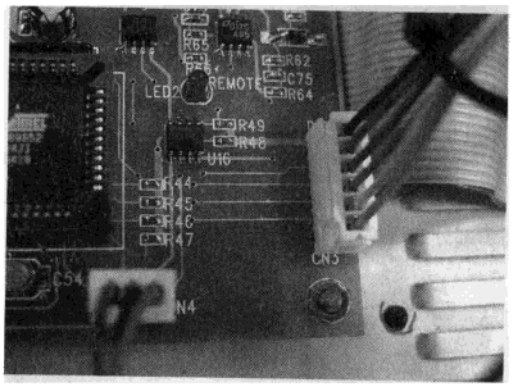


图 2-5　几款适合 DIY 的数字电表，从左到右分别为 Agilent 34401A、FLUKE189、FLUKE10

2.2　逻辑笔的使用

逻辑笔（Logic Probe）主要用来检测数字电路的逻辑状态，而这里所谓的逻辑状态就是数字的 0 和 1，我们只要把逻辑笔的测试端接触待测点，就可以得知该点的逻辑状态。通常逻辑笔的工作电压为+5V，可直接取自待测板上。

在我们这里特地拿一支逻辑笔做测量示范，试验的对象是一块七段显示驱动板的控制接点，图 2-6 是该接头的引脚图。



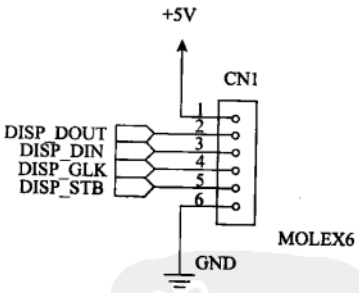


图 2-6　用逻辑笔检测控制板 CN1 接头的逻辑状态

步骤 1：首先要找到逻辑笔要的+5V 电源，把 GND 接黑色鳄鱼夹，+5V 接红色鳄鱼夹，并把待测电路的电源打开。

步骤 2：用手摸逻辑笔的前面金属尖端，代表 Hi/Lo 的灯都会亮，这是因为人体会感应交流信号的结果，如果两个灯都亮，代表逻辑笔正常，可以进行测量。此时我们把信号电平选择切到 TTL 端，准备测试数字信号。

步骤 3：用逻辑笔的前面金属尖端接触图 2-6 的+5V 电源端，此时代表 Hi 的红 LED 灯要亮，这代表此时测量到的状态是数字 1，此时亮的红灯是一直持续的。

步骤 4：用逻辑笔的前面金属尖端接触图 2-6 的 GND 地端，此时代表 Lo 的绿 LED 灯会亮，这代表此时测量到的状态是数字 0。

步骤 5：当把逻辑笔的测试点放在 Data 引脚上。情况就不一样了，此时亮的是红灯，但是伴随 Pulse 的黄色灯一直闪烁着，这是代表此点大部分时间是数字 1，不过有时候会降低到 0，但又回到 1。当 1 与 0 的状态变化时，Pulse 灯一定会亮。

步骤 6：如果测量的电路板是 CMOS 电平的，要把逻辑笔上的信号输入选择切到指示 CMOS 端。

逻辑笔使用上很方便，对于简单的逻辑状态判定是很好的工具。但还是有若干不方便之处，例如只测出状态有变化，但不知道变化的频率或次数，对于复杂的数字电路或内含单片机的电路，逻辑笔在测量上有帮助，但总觉得帮忙有限。

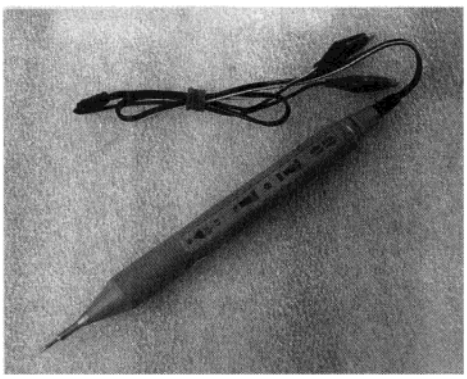


图 2-7　逻辑笔的外观，通常逻辑笔没有内署电源，必须从待测板上另外取得+5V 电源

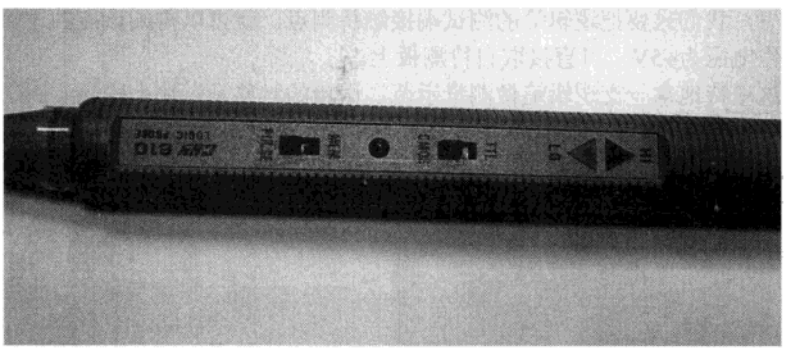


图 2-8　逻辑笔显示部分的近拍图

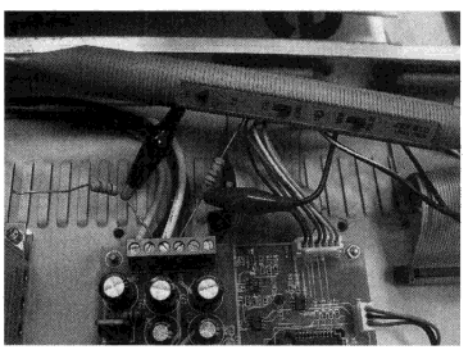


图 2-9　先为逻辑笔找到+5V 电源

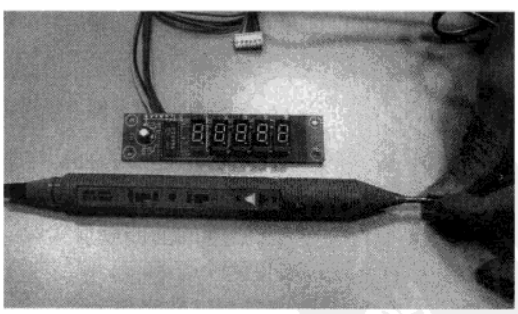


图 2-10　用手摸逻辑笔的测试端，可以看到 1 和 0 一直在变化

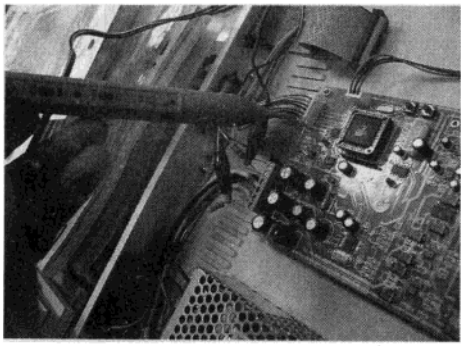


图 2-11　接触+5V 端可以看出其逻辑状态为 1

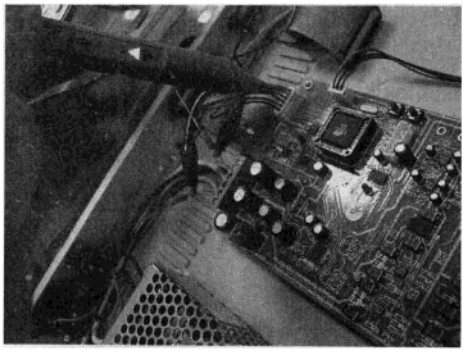


图 2-12　接触电源地端 GND 可以看出其逻辑状态

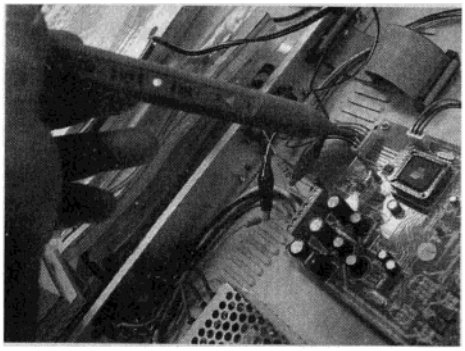


图 2-13　接触 Data 端的数字状态时，可以看出有变化的数据源源不绝送出来

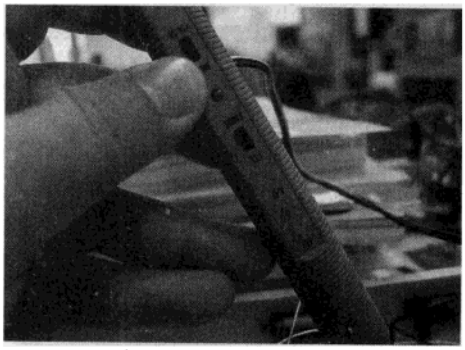


图 2-14　如果检测的电平为 CMOS，须将开关选择从 TTL 切到 CMOS

2.3 蜂　鸣　器

蜂鸣器的英文名称为 Buzzer，念起来就好像真的蜜蜂靠近时的声响。在 DIY 实验中，我们经常会用声响来代表某个状态或条件成立。最常见的例子：数字电表内部也有一个小蜂鸣器，当我们做电阻挡的导通测试时，若测试结果在 1Ω 以下时，电表除了显示欧姆值外，还会伴随声响，而这个声响就是由 Buzzer 所产生的。

在一般的电子元件商店可以买到的蜂鸣器有两种：第一种是内部已有振荡电路，只要一加上+5V 的电源就会发声，不过其音调是固定的；另一种蜂鸣器是没有内置振荡电路，所以我们必须对蜂鸣器加入方波信号，蜂鸣器才会叫，而且其音调刚好就是方波的频率。方波的频率太高或太低都有可能使蜂鸣器失声，最适合的频率应该在数千赫兹间。

内置振荡电路的蜂鸣器会贵一点，但是谈到方便性的话，还是以含振荡电路的蜂鸣器比较好。在这本书里若使用到蜂鸣器都是指内置振荡电路的蜂鸣器，且规格是+5V 的。蜂鸣器内部是否有振荡电路在外观上是看不出来的，在 DIY 购买元件时，请特别要注意这一点。

新买的蜂鸣器如何验证其好坏呢？请先找到直流电源供应器，把电压调整到+5V，蜂鸣器是有极性的，电源正端+5V 接蜂鸣器的较长引脚，电源地端接蜂鸣器的另一引脚，应该就可以立刻听到蜂鸣器的叫声，而且其消耗电流应该只有几十毫安而已。蜂鸣器的故障率应该非常低，出现故障的主因都发生在电源错误反接，或是烙铁焊接的时间过久（超过 5s），因而导致内部接线断开所致，这些也是 DIY 时要注意的。

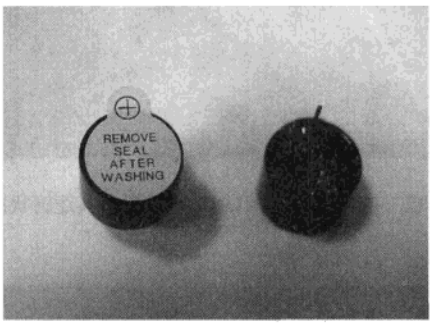


图 2-15　蜂鸣器的正反面近拍

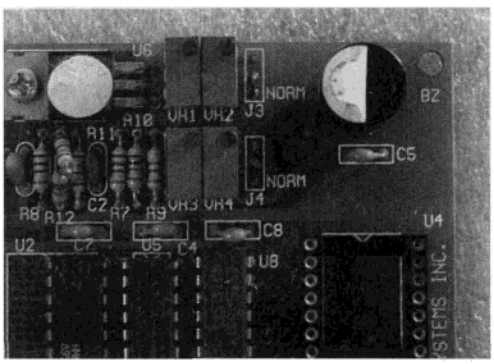


图 2-16　蜂鸣器正面的贴纸若去掉的话，声响会变大

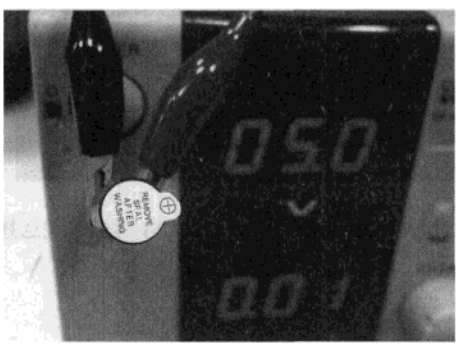


图 2-17　用电源供应器测试蜂鸣器的好坏，请注意极性

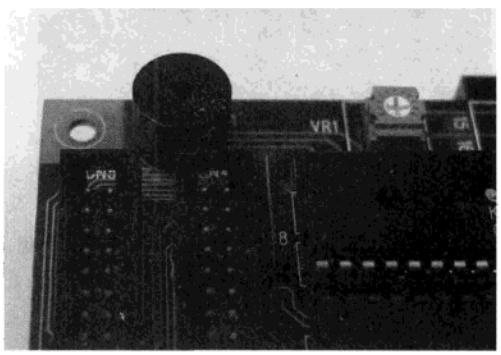


图 2-18　ChipWare51 控制板上也有一蜂鸣器做声响指示

2.4　USB-ISP 烧录器的使用

在线烧录（In System Programming）是最近新单片机的一致趋势，只要通过一条下载线，就可以由电脑的 USB 端口上将程序代码传入 IC 中，不仅烧录时间加快，也省去了 IC 拔上拔下的时间。以美国 ATMEL 公司为例，若 IC 编号上有 S 字样的 8051 单片机就有支持 ISP 在线烧录的功能，这里我们就以 USB-ISP 烧录器和 AT89S52 当成范例，示范烧录该 IC 的重要步骤。

步骤 1：烧录器与电脑连接。

用提供的 USB 线将烧录器与电脑的 USB 端口连接，该烧录器直接由 USB 端口供电，所

以不需额外电源。

步骤 2：安装烧录器的驱动程序。

第一次连接电脑会显示“找到新硬件”并出现安装驱动程序的窗口。

步骤 3：执行 USB-ISP 烧录程序。

执行后会出现一个非常简洁 USB-ISP 的烧录画面。

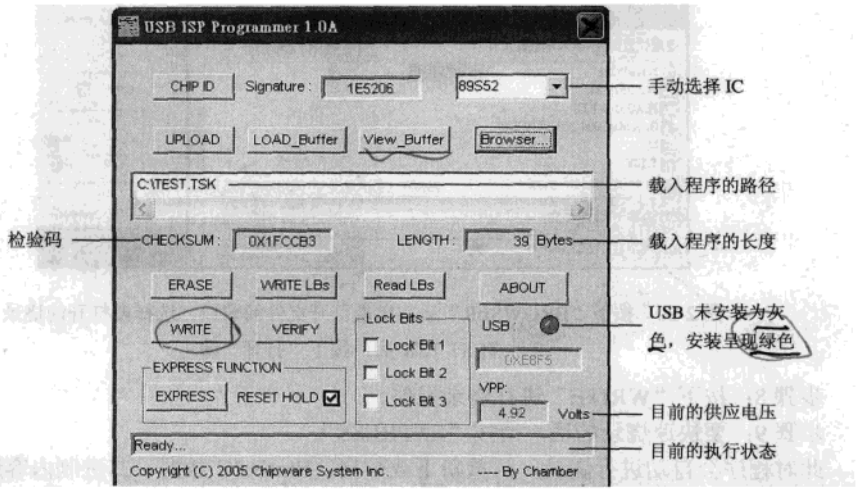


图 2-19　USBAISP.EXE 打开后的烧录界面

CHIP ID：自动检测目前使用 IC 的型号，这一版本可以侦测 AT89S51、S52 和 S53。

UPLOAD：将烧录文件上传到 PC 上。

LOAD_Buffer：下载烧录内容到暂存区，准备进行烧录。

View_Buffer：查看读入 Buffer 后的内容。

Browser：浏览需载入资料文件的完整路径。

ERASE：把 IC 内容清除成 FFH。

WRITE：烧入暂存于 Buffer 的内容。

WRITE LBs：烧录完成后加上 Lock 位。

VERIFY：进行烧录后数据的比较验证。

EXPRESS：快速烧录，自动进行 ERASE、WRITE、VERIFY 和 LB 加锁等一连串的操作。

完成这些初步的操作并了解后，接下来要进入到烧录操作了。

步骤 1：确定目标板上的下载接头和 USB-ISP 所定义是一致的。

步骤 2：连接 ISP 线，并确定目标板上没有短路情形。

步骤 3：执行 USB-ISP.EXE。

步骤 4：按下“CHIP ID”，确定 IC 型号是正确的。

步骤 5：按下“BROWSER”浏览并选择欲烧录的二进制文件。

步骤 6：指定完文件名后，按下“LOAD_Buffer”立即载入烧录数据到烧录暂存区。

步骤 7：按下“View_Buffer”观看刚载入的内容。未进行加载前，整个 BUFFER 区会事先被填成 FF。

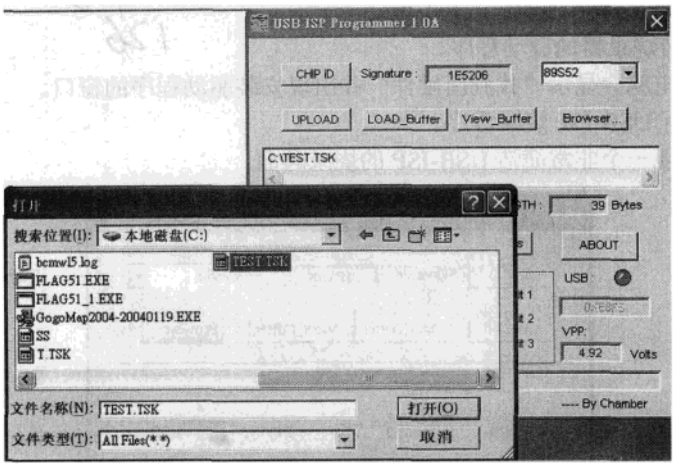


图 2-20　按下“BROWSER”后，出现打开文件的窗口，选择要打开的烧录文件 TEST.TSK 后，按下“打开”即可

步骤 8：按下“WRITE”进行烧录操作。

步骤 9：要快速烧录的话，请按“EXPRESS”。

此时程序会自动进行烧录，从重新下载数据、Erase IC 数据、二进制内容烧录写入、比较与 Lock 位锁码操作等等，你当然也可以一个一个分开进行。但是烧录的内容与前一笔相同时，这个选项还是可以节省一连串设定的操作。

步骤 10：烧录完成后，USB-ISP 程序会自动对目标板产生 RESET 重置信号，若一切正常的话，目标板上的程序应该立即执行。

USB-ISP 硬件方面比较特殊的是它有 USB 电源电压的侦测并显示，若测得 USB 的电压小于+4.5V 时，可能就无法将程序代码烧录到 AT89S 系列的 IC 当中。在您的 8051 设计当中，若要采用 ISP 线直接下载的功能时，请先预留如图 2-21 的接头，即可运用 USB-ISP 下载工具进行下载。

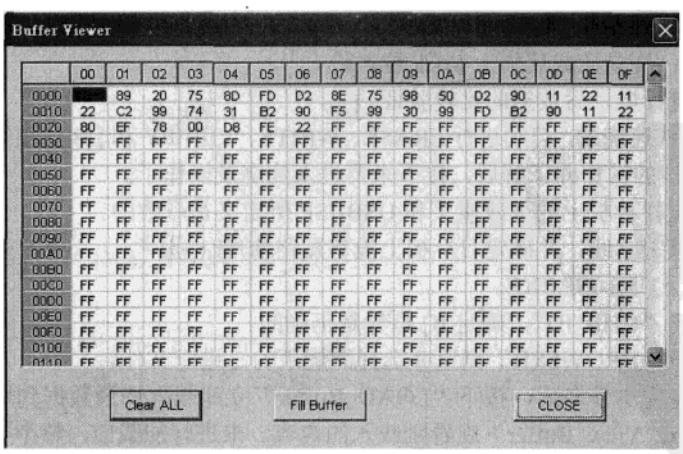


图 2-21　按“VIEW_Buffer”按钮，可以清楚地看到将被烧录到 IC 的内容

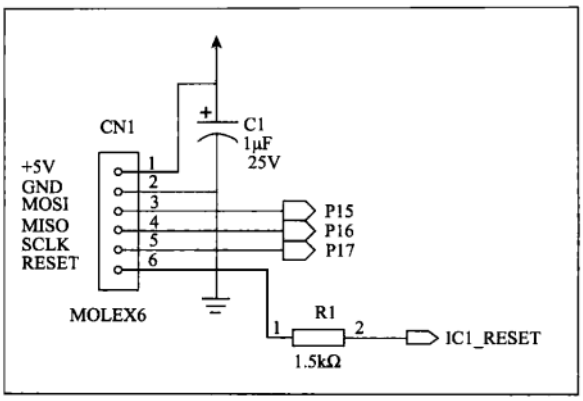


图 2-22　下载接头的规格和信号排列

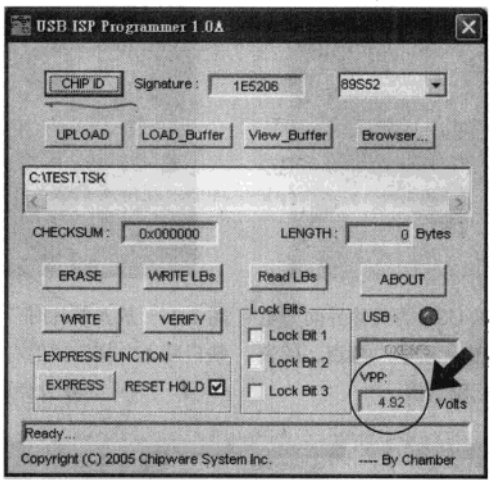


图 2-23　每次检查 CHIP ID 一次，就会更新 USB 的电源电压一次

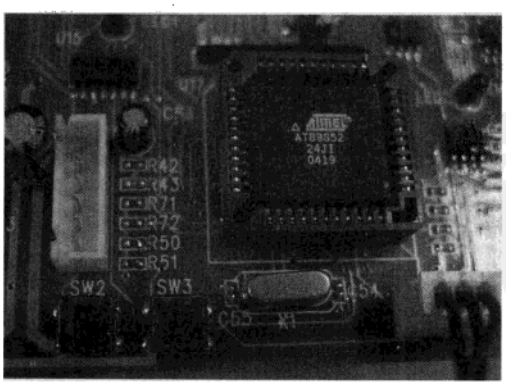


图 2-24　旗威科技公司新开发的 8051 应用板，CPU 采用 AT89S52，CPU 插座旁的接头（左边）就是 ISP 下载专用接头

2.5 PGM2051 烧录器的使用

PGM2051 的软硬件安装，在附录中有详细的介绍，在这里我们仅操作烧录的步骤，并验证是否烧录成功。

1. 一般烧录法

步骤 1：将烧录器通过 USB 传输线与 PC 连接。

步骤 2：将 AT89C2051 放入烧录器，并注意其引脚是否摆放正确。

步骤 3：选择“程序”中的“单片机烧录器”启动烧录程序。

步骤 4：选取正确的芯片型号 AT89C2051。

步骤 5：选取所要烧录的文件，把文件载入烧录器（Load)。

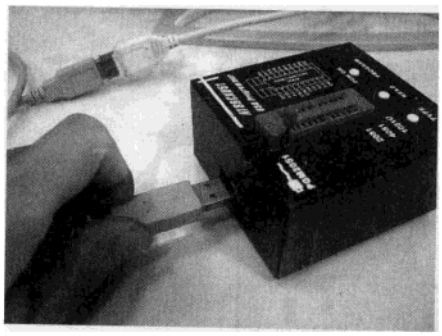


图 2-25　PGM2051 烧录器以 USB 为传输界面与电脑连接，连接后烧录器会发出“哧哧”两声，代表烧录器已经正确连接

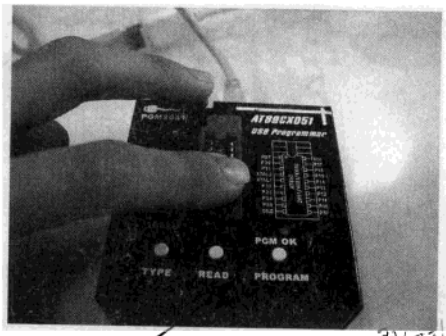


图 2-26　放入芯片时一定要注意引脚是向下对齐但缺口朝上，烧录器的右边有图标，如果不正确摆放会减少烧录器的使用寿命

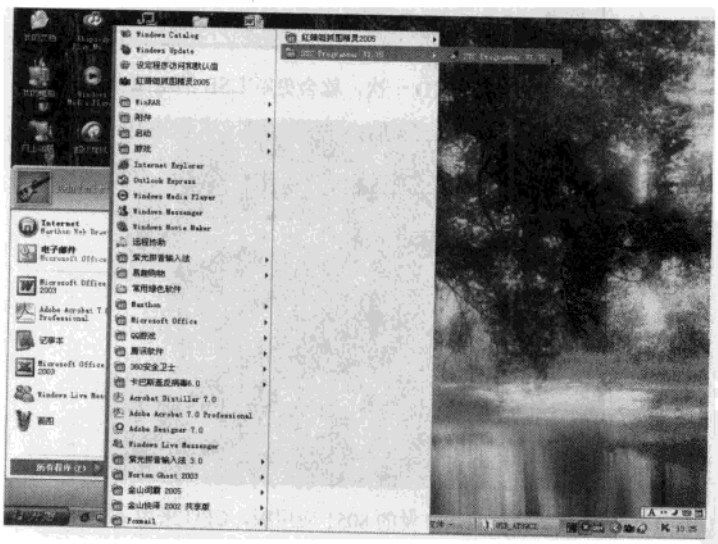


图 2-27　烧录器安装完成后就可以从“程序”里直接启动烧录程序

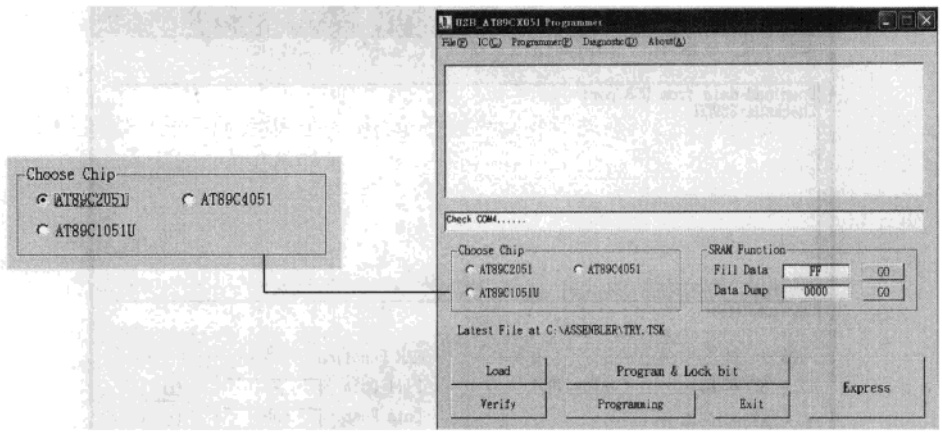


图 2-28　运行程序后要选择待烧录芯片型号

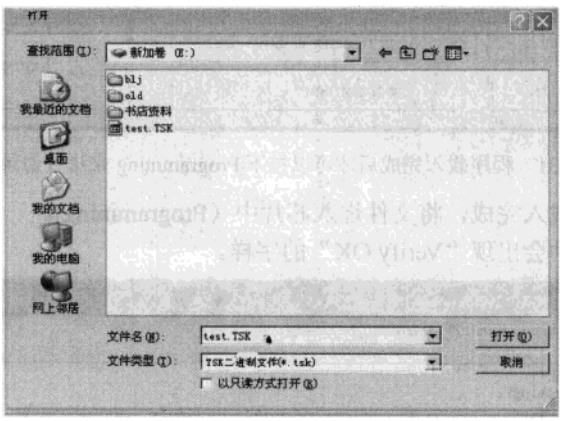


图 2-29　按下 Load 按钮选取准备烧录的程序文件

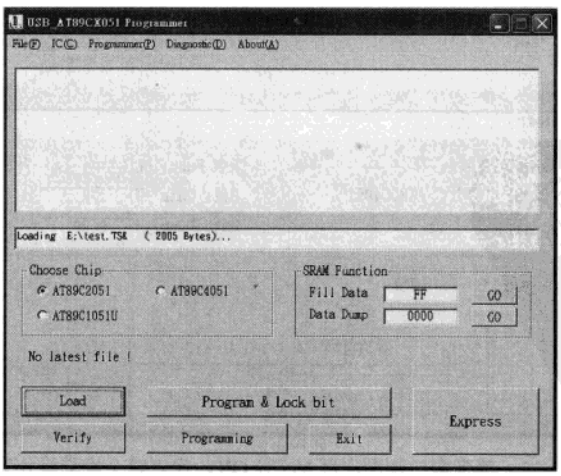


图 2-30　烧录程序正在进行载入操作

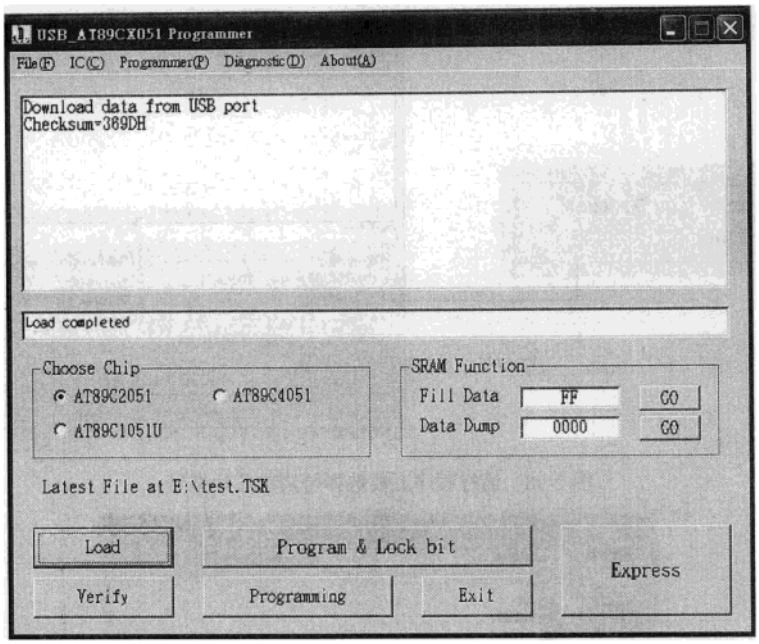


图 2-31　程序载入完成后才可以按下 Programming 来进行烧录操作

步骤 6：待程序载入完成，将文件烧入芯片中（Programming）。

步骤 7：烧录成功会出现“Verify OK”的字样。

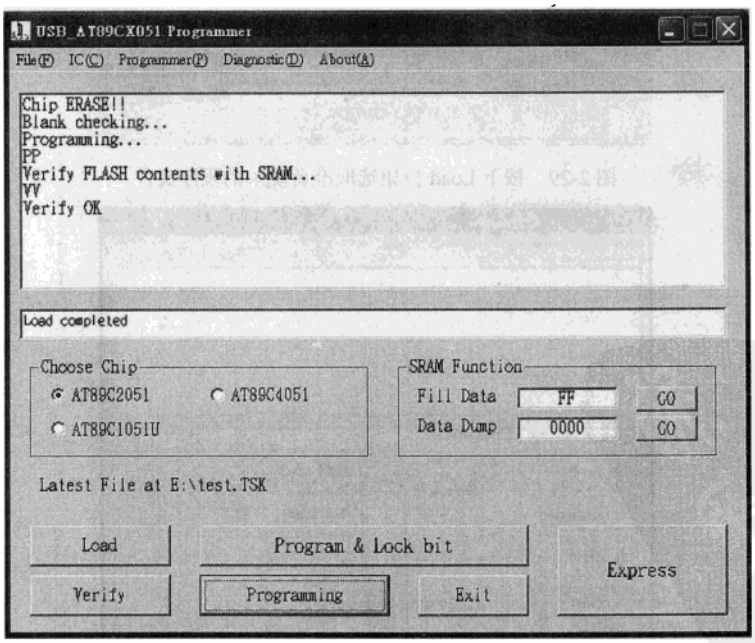


图 2-32　等到窗口画面出现 Verify OK，代表整个烧录操作完成

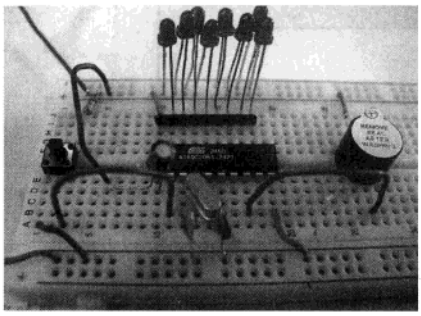


图 2-33　最后只要将烧录好的芯片放到实验电路板上，就可以开始验证程序操作

步骤 8：将烧录成功的 AT89C2051 取下，放到实验电路上验证操作。

若烧录失败，会出现“Verify FAIL at addr[????H]”，请取下 AT89C2051 将引脚清洁整理一下，再放入烧录器烧录即可。若还是烧录失败，请确认[????H]是否每次都一样？如果都不一样，那您所使用的 AT89C2051 应该是坏了；如果都一样，则可能是烧录夹的某个引脚已经氧化而无法进行烧录，请将烧录器送厂维修保养吧。

如果您修改自己所编写的程序后，文件名及文件路径不变，您可以用下面的方式快速烧录您的程序。

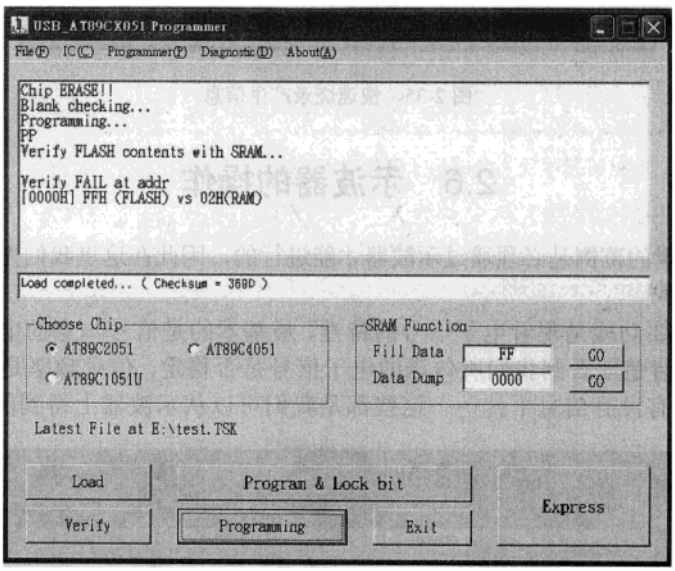


图 2-34　烧录失败出现的信息状态，这时要开始检查烧录器的使用状况

2. 快速烧录法

步骤 1：将烧录器通过 USB 传输线与 PC 连接。

步骤 2：将 AT89C2051 放入烧录器，并留心其引脚是否摆放正确。

步骤 3：单击“程序”中的“单片机烧录器”启动烧录程序。

步骤 4：选取正确的芯片型号 AT89C2051。

步骤 5：单击快速烧录键（Express），待信息窗口出现。

“Program and Verify are disabled”字样即完成烧录。

按下 Express，执行快速烧录完成时所产生的操作信息，不过一定要注意此时快速烧录的程序为上一次执行的烧录文件，如果路径或文件名有变更，还是要使用一般步骤进行烧录才是正确的。

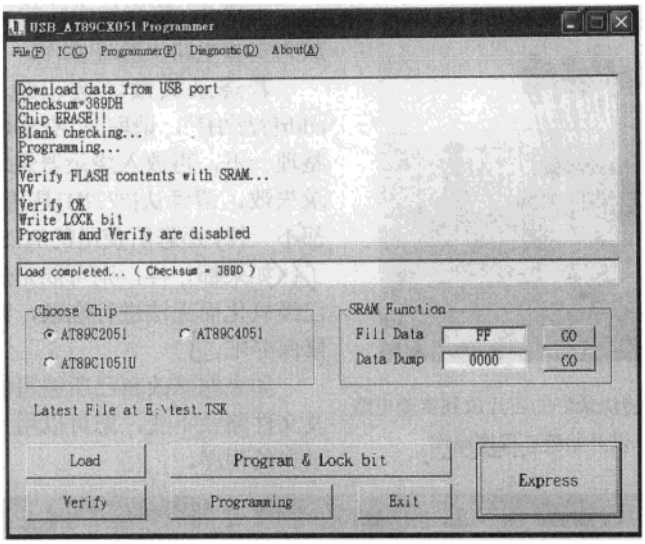


图 2-35　快速烧录产生信息

2.6　示波器的操作

本书中有大量的范例是必须通过示波器才能进行的，因此在这里我们先将示波器的操作方法熟悉一下。

示波器的主要功能是观看电子信号的特性，最基本的是信号本身的电平变化。通过示波器，我们可以清楚地看到我们所传送的电子信号是否稳定，信号频率是否正确，电平的变化是否正常，有没有信息干扰……这些都是我们可以从示波器上得到的重要信息。

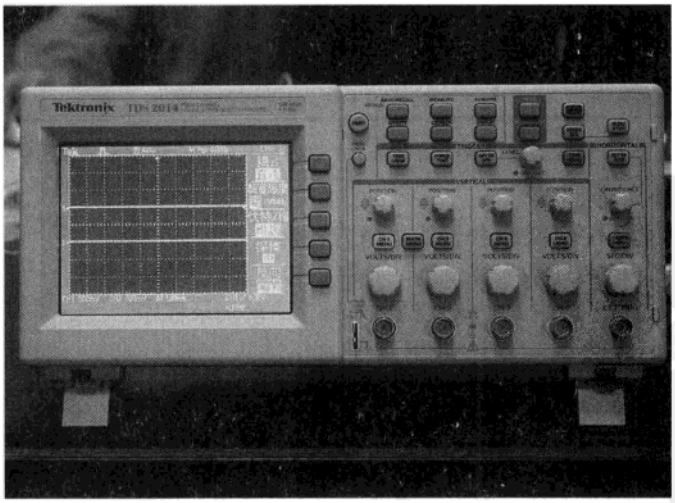


图 2-36　数字存储式示波器的外观

TDS2014 是一台四通道(4CH)、彩色液晶显示的存储式示波器，频率为 100MHz，信号取样率高达 1 GHz，可以满足绝大多数的应用。本书的所有范例，都可以用 TDS2014 来进行信号观察与除错。

示波器在操作上有三个要点：①选择正确的电压范围；②调整观看波形的适当时间范围；③定义提取信号的触发条件。

1. 选择正确的电压范围

以观察 8051 的信号为例：MCU 的供应电源是 DC5V，所以我们在调整电压范围时，以示波器显示画面的 1/3～1/2 可以观看到完整的 DC5V 为最好。如果要调整为 1/3 画面，则全屏幕可呈现的电压范围应为 5×3=15V，平均每格所需要的电压范围是 15/8=1.875V，最接近的显示范围就是调整为每格 2V，这么一来，我们就可以看到最适合观看的波形比例。

在数字存储式示波器的屏幕上，纵向有八个方格（模拟的通常为十格），横向有十个方格，设定的电压或时间范围，都是以一格来计算。

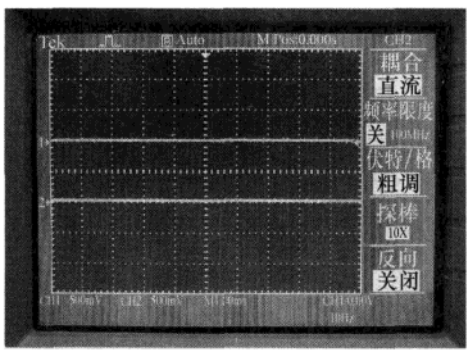


图 2-37　数字存储式示波器的屏幕

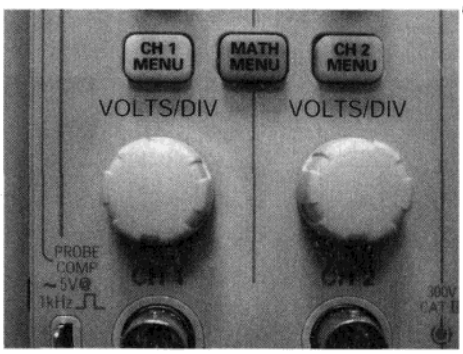


图 2-38　电压调整按钮

当然，如果我们所要观察的波形是很简单的信号，放大到全屏幕来观看也可以，只是大多数的场合，我们会同时比较两组信号，因此，一组信号占用画面的 1/3～1/2 是最恰当不过的了。

电压调整的旋钮，顺时针调整为缩小电压观看范围，换句话说，就是放大波形的意思。反之，逆时针调整为放大电压观看范围，也就是缩小波形的意思。

调整好电压的范围，还要记得把波形移到屏幕的适当位置，这时要通过垂直轴位置旋钮来修正。如果是一组信号，就移到画面的正中央；如果是两组，就一上一下，中间留一小段间隔。

2. 调整适当的观看时间范围

调整好电压范围，接着要调整时间范围。一般来说，有规则性的波形，在一个画面下显示的数量以 3～5 个为最恰当，如果太多，波形看起来会很拥挤；如果太少，当信号上出现噪声时就不容易记录。假设我们要观察周期为 13ms 的方波，应该怎么调整时间范围才合适呢？

时间间隔调整钮。顺时针调整，缩小每格所能呈现的时间范围，能减少屏幕上波形的显示数量；逆时针调整，加大时间范围，让波形显示的数量增加。

同前面所提到的电压调整，以三个完整波形为例，则全屏幕所要呈现的时间范围为

3×13=39ms，每一格的时间范围是 39ms/10=3.9ms，最接近的时间间隔为一格 5ms。

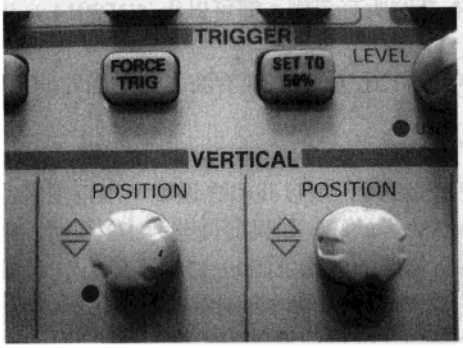


图 2-39　垂直轴位置旋钮。顺时针调整为向上，逆时针调整为向下

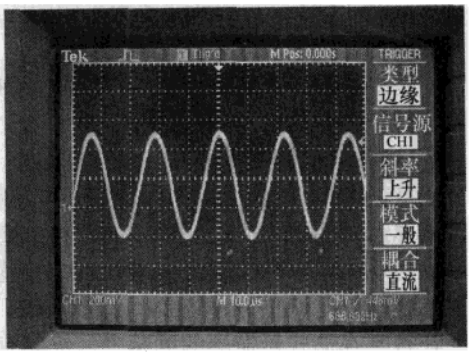


图 2-40　适当的波形调整，可以让波形更容易观察

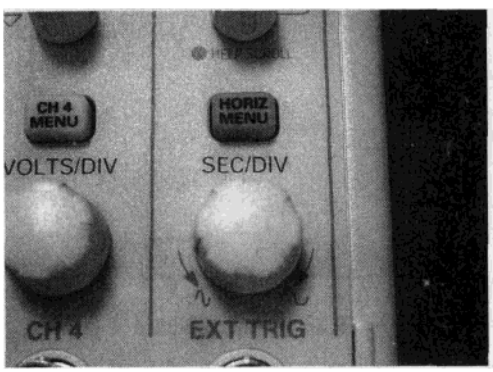


图 2-41　时间间隔调整钮

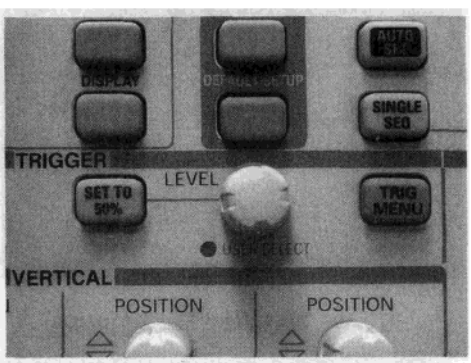


图 2-42　触发电平的调整钮，顺时针就是提高触发电平，逆时针为降低触发电平

3. 触发条件的设定

所谓的触发条件，就是示波器开始记录波形变化的时机，一般可分为上升或下降两种，指的是在电压电平上升或下降时要开始记录波形。怎么决定电压是上升或下降呢?我们要通过触发电平的设定，来告诉示波器何时为上升，何时为下降。只要波形的电压电平本来比触发电平低，经过一段时间后，变得比触发电平高，这个变高的时间点就叫做上升；同理，下降就是电压电平由高变低的时间点。

设定好触发电平和触发的上升或下降后，还有一个重要的事情要做，就是把触发成立的时间点调整到屏幕的适当位置。通过水平位置调整钮，可以把波形左右移动，方便我们看到完整的波形。

到此，示波器的主要功能介绍完毕。不过，光说不练是很容易忘记的，接下来的内容，我们要实际测量一下未知的波形。

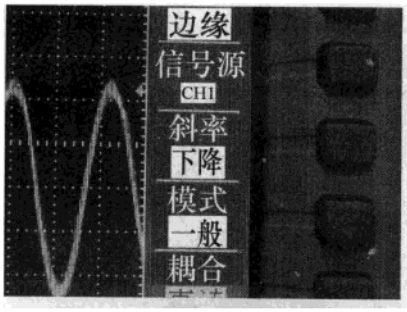


图 2-43　触发条件设定为下降

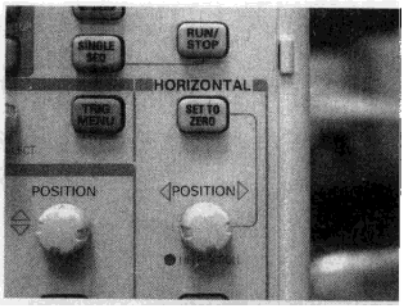
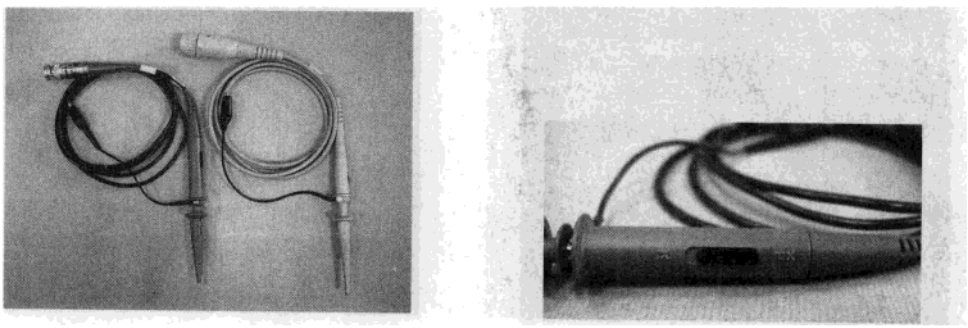


图 2-44　水平位置调整钮。顺时针向右，逆时针向左

2.7　波　形　观　察

在本书的光盘里，有个“WAVE”的文件夹，里面有三个未知的模拟波形程序，观察点是 MCU 的 P1.0 跟 P1.1，运用前面所介绍的示波器功能，我们实际来操作看看。

（1）先把 wave__1.tsk 烧录到 MCU 中，并将示波器的测棒接上，再打开示波器的电源。



（a）　　（b）

图 2-45　一般数字存储式示波器所附的测棒是 10×的，高阶一点的还可以选择是 1×或 10×。所谓的 10×，就是将信号的电压电平衰减十倍，让示波器可以测量的电压范围加大十倍

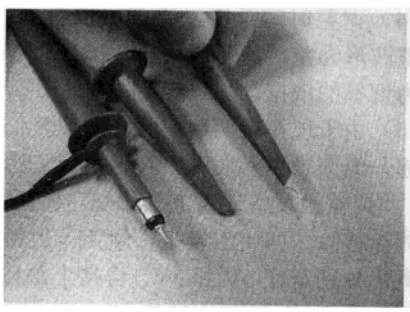


图 2-46　测棒

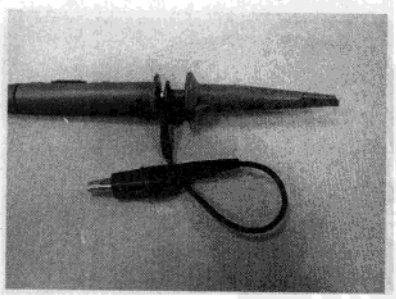


图 2-47　鳄鱼夹

常见的测棒前端由两种形式组成：一种是勾具，另一种是探针，两种形式可以通过伸缩式勾具（Retractable Hook Tip）做为转换。接上时为勾具，拔下时为探针。

接地用的鳄鱼夹。测棒上会有一条接地线，如果没有接上，测量的信号就少了参考电平，所以测量时一定要接上。这一条接地线要越短越好，测量到的信号才不容易受噪声干扰。

(2)将测棒 1 与测棒 2 的接地夹固定在 MCU 的 GND 脚，测棒 1 测量 P1.0，测棒 2 测量 P1.1。

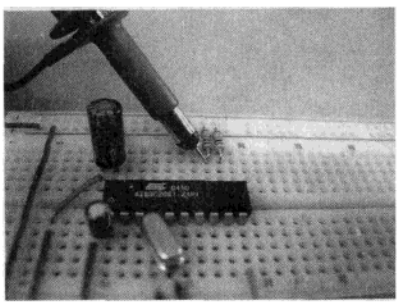


图 2-48　测棒固定的方式直接用探针抵住所要测量的信号来源

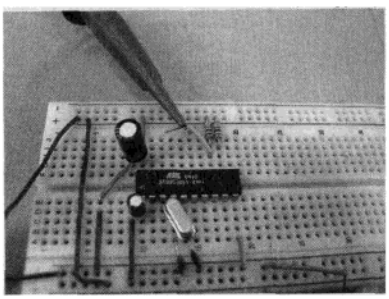


图 2-49　测棒固定的方式通过外接的测试夹或延伸线，用勾具勾住信号来源

（3）调整电压范围，让波形维持在画面的 1/3～1/2，并利用垂直调整钮调整两组波形的位置。

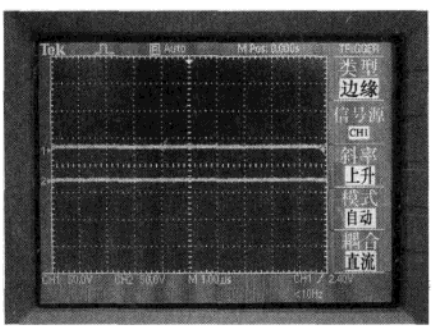


图 2-50　先将电压范围调整为最大，垂直位置与水平位置都选在屏幕正中央，并利用示波器的自动触发功能，观看波形的特点

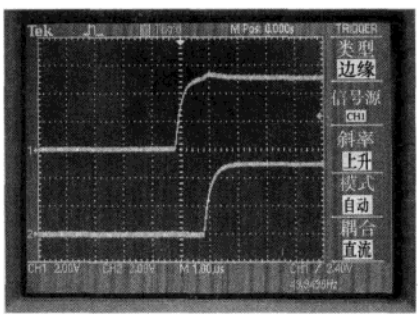


图 2-51　慢慢修正电压范围，并调整两组波形的垂直位置，测棒 1 在上，测棒 2 在下

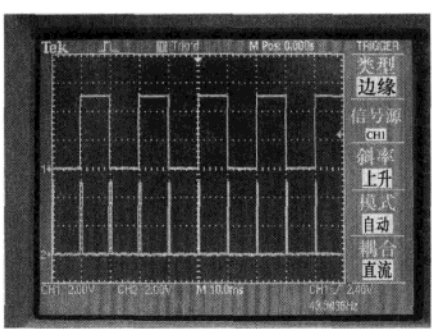


图 2-52　以完整波形较少的一组信号，作为调整时间范围的参考

（4）调整时间范围，直到可以看到 3～5 组完整波形为止，通常我们会以波形较少的一组做为调整的依据。

（5）由于现在有两组信号，所以我们要决定由哪一组来当做触发源，在这里我们选用测棒 1，并调整触发条件为上升触发。选好之后，先将测棒移除，并将触发条件改为一般触发，触发电平分别定在 6V 和 3V，再接上测棒，我们看看有什么差别。

（6）最后，通过水平位置的调整，我们可以看到触发点之前和之后的相关波形。如果要

观看波形间的相互关系，我们也可以把提取信号的时间范围调大，等收到较多的波形后，让示波器停止提取信号，再将波形展开来看（缩小时间范围）。

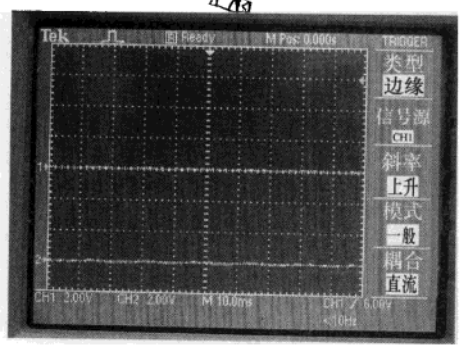


图 2-53　触发电平定在 6V 时，5V 的信号不可能上升到 6V，所以一直看不到波形

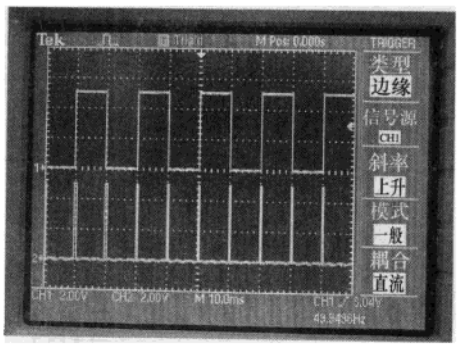


图 2-54　触发电平定在 3V 时，可以看到两组波形稳定地显示在画面上

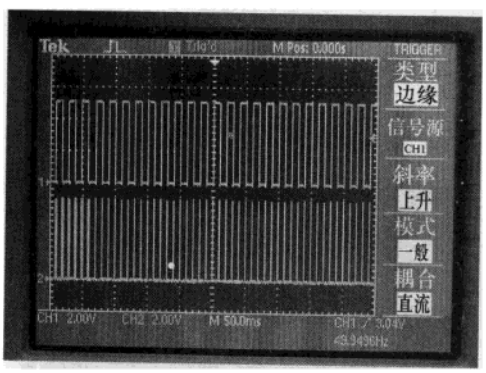


图 2-55　将提取信号的时间范围调大，可以看到较多的波形

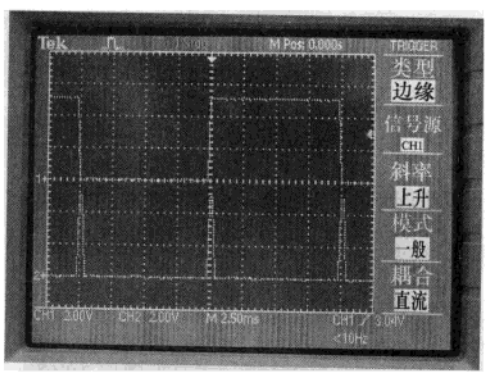


图 2-56　虽然通过展开波形可以看到更多的信息，但展开的次数太多会让波形失真

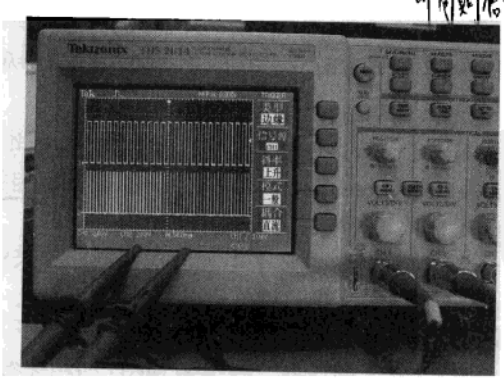


图 2-57　停止示波器提取信号时，已提取的波形会被完整保留，此时不管测棒上的信号如何变化，都不会改变显示波形的样子，这就是数字存储式示波器的最大优点

以上是示波器提取信号波形的完整步骤，大多数的信号波形都可以通过这样的方式来观察，光盘中的另外两个范例程序，就留给你亲手试试看。

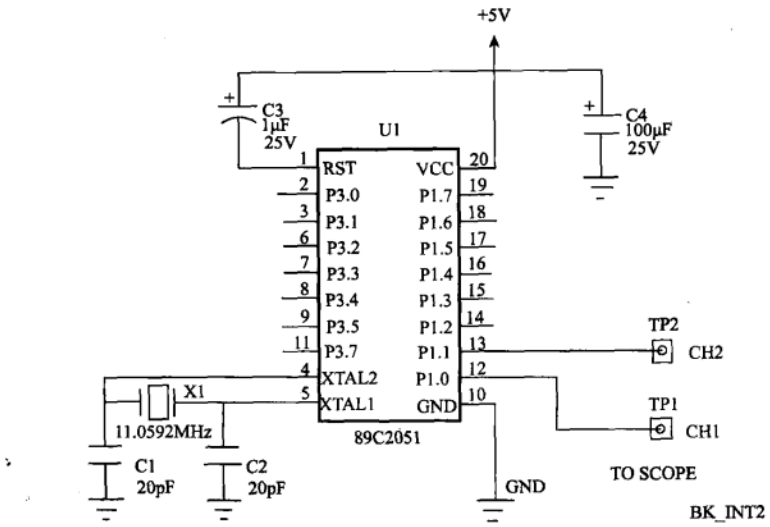


图 2-58　本节所使用的实验线路

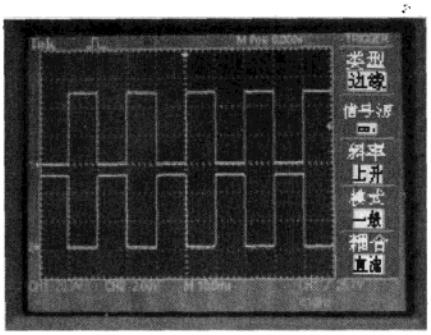


图 2-59　用示波器观察 wave_2.tsk 的波形

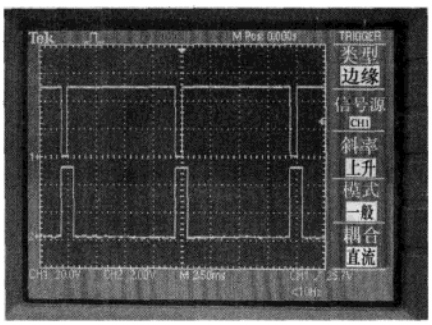


图 2-60　用示波器观察 wave_3.tsk 的波形

您可以从下列公司的网站取得更进一步的信息：

（1）www. chipware. com.tw：查询 PGM2051 与 USB-ISP 的参考资料。

（2）rpelectronics.com/english/content/items/lp-610.asp：逻辑笔相关资料。

（3）www.fluke.com：查询掌上型数字电表的相关资料。

（4）www.agilent.com.tw：查询桌上型数字电表与示波器的相关资料。

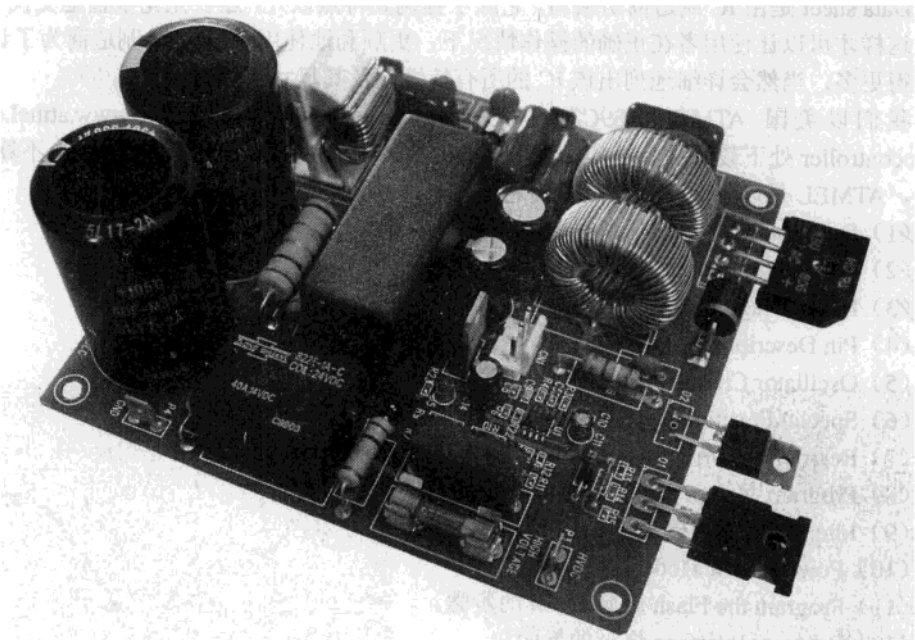
（5）www.tektronix.com：查询示波器的相关资料。

（6）www.topward.com：查询电源供应器的相关资料。

（7）www.atmel.com：制造 8051 的厂商，有很多 MCU 的相关参考资料。

（8）www.intel.com：查询 8051（MCS51）的相关介绍。

3 入门知识篇



我们经常收到读者来信，都是询问 8051 要怎么学才快？有哪些相关的知识是必须先自己理解的？老实说，这是一个没有标准答案的问题，也很不容易回答。不过，我们确实可以整理出在学习 8051 时应该具备的基本条件，如果您跟大多数的读者一样，在学习上也有相同的困扰，不妨仔细看看本章所为您准备的内容，希望可以给您一些启发。

第3章　入门知识篇

3.1　如何看懂 Data Sheet——基础篇一

Data sheet 是由 IC 制造商所编写，它除了强调其特点以外，还要完完整整地交代 IC 的用法，这样才可以让使用者在正确的操作情况下，更顺利地使用这枚 IC。制造商为了让 IC 能够卖得更多，当然会详细地列出该 IC 的所有特性以及其尺寸等等。

我们以美国 ATMELAT89C2051 的 Data Sheet 为例，请先到 www.atmel.com 的 microcontroller 处下载后并且打印出参考。AT89C2051 的 Data Sheet 总共有几页，不算多也不算少，ATMEL 是以下面几个章节来描述这枚 IC 的所有特点：

（1）Description 芯片的基本叙述。
（2）Pin Configuration 引脚图。
（3）Block Diagram 操作结构图。
（4）Pin Description 引脚说明。
（5）Oscillator Characteristics 振荡电路特性。
（6）Special Function Registers 特殊功能寄存器说明。
（7）Restricfions on Certain Instructions 某些指令的限制。
（8）Program Memory Lock Bits 锁存位的说明。
（9）Idle Mode 闲置模式。
（10）Power-down Mode 省电模式。
（11）Program the Flash 烧录 Flash 的步骤。
（12）Program Interface 烧录的界面。
（13）Flash Program Modes 烧录模式说明。
（14）Flash Programming and Verification Characteristic 烧录与读回验证的特性。
（15）Flash Programming and Verification Waveforms 烧录与读回验证的波形。
（16）Absolute Maximum Ratings 绝对最大值。
（17）DC Characteristics 直流特性。
（18）External Clock Drive Waveforms 外部 clock 驱动波形。
（19）External Clock Drive 外部 clock 驱动能力。
（20）Serial Port Timing 串行端口的时序。
（21）Shift Register Mode Timing Waveforms 移位寄存器时序图。
（22）AC Testing I/O Waveforms 输入/输出波形。
（23）Float Waveforms 浮接时的波形展示。
（24）Icc Measurements CPU 耗电 Icc 测量。
（25）Ordering Information 订购信息。
（26）Package Information IC 包装信息。

大部分正规的 Data Sheet 都是有类似的安排，在 AT89C2051 几页的文件中，以第 1、2、3、4、16 与 17 项是最值得注意的，在看懂 Data Sheet 基础篇上我们就先谈到这几项，你一定要弄懂这些资料再开始进行 DIY 的实验。

1. Description 芯片的基本叙述

这枚 IC 的主要功能描述，所有的重要特点都会先在这里说明。

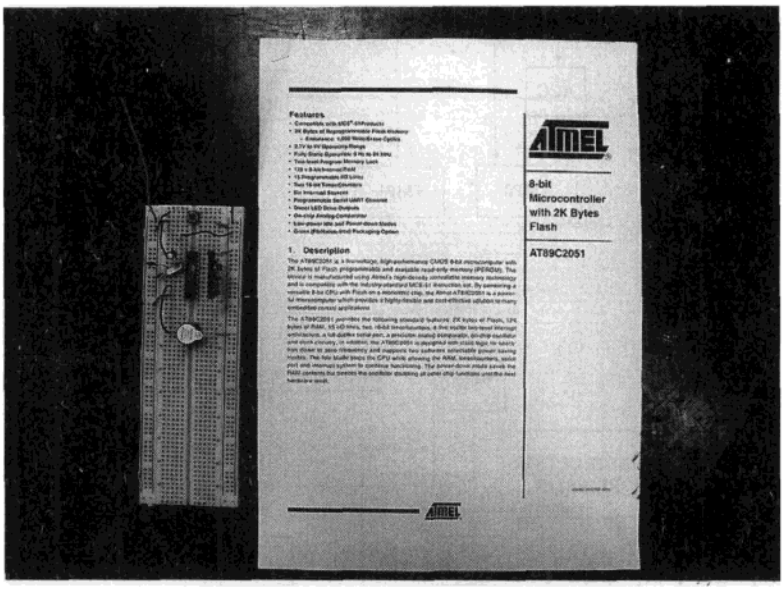


图 3-1　AT89C2051 DataSheet 图片

2. Pin Configuration 引脚图

IC 的引脚图，如果 IC 有两种包装方式时，资料上就会列出两种不同的引脚图。

3. Block Diagram 操作结构图

IC 内部经过简化的结构图，这部分主要告诉使用者 IC 是如何工作的。

4. Pin Description 引脚说明

每个引脚的操作说明。IC 制造厂对这里一定着墨很深，详尽交代每个引脚所扮演的角色，如果不够的话，再加上图表来辅助说明，务必让使用者熟悉所有的功能。我们的经验是这一部分要多看几次，才能理解其中的精髓。

5. Absolute Maximum Ratings 绝对最大值

这部分交待 IC 所能容忍的电压或电流极限值，只要一超过厂商所公布的极限时，IC 不一定立即损坏，但是内伤是一定会有的，如果继续使用的话，何时出错是很难推断的。我们在设计或使用上应该尽量避免让 IC 工作在极限值附近。

以 AT89C2051 为例，其工作温度可在-55～+125℃之间。保存温度范围比较大，为-65～+150℃，每个引脚所能忍受的电压输入范围是-1.0～+7.0V，最大工作电压为 6.6V（正常供应电压为+5V，DC），直流最大输出电流为 25.0mA。这些极限值在 IC 设计时，就已经被设定了，当 IC 制造出来后，还要进行一连串的试验和认证，最后才公布定案，所以绝对是有其参考性的。当我们在做单片机 AT89C2051 的应用时，这些极限值应该随

时记在大脑中。如果 IC 突然不工作了，我们就要马上怀疑 IC 是否已经超过制造厂商所公布的极限值。

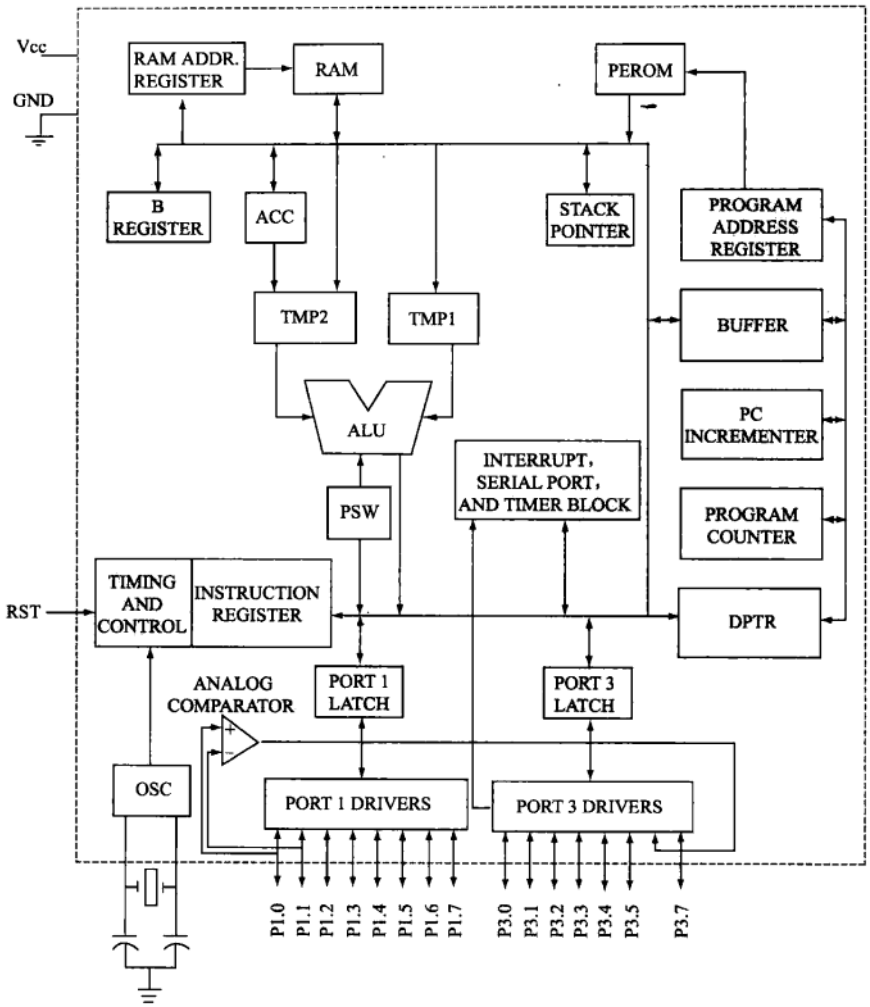


图 3-2　AT89C2051 的内部结构图

16. Absolute Maximum Ratings*

Operating Temperature	-55°C to +125°C
Storage Temperature	-65°C to +150°C
Voltage on Any Pin with Respect to Ground	-1.0V to +7.0V
Maximum Operating Voltage	6.6V
DC Output Current	25.0 mA

*NOTICE: Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions beyond those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

图 3-3　AT89C2051 的最大极限值，长时间处在极限值下时会影响该芯片的可靠性（摘自 Atmel AT89C2051 Data Sheet）

3.2 如何看懂 Data Sheet——基础篇二

制造厂商把所公布的 Data Sheet 分为四大等级，分别代表该特定 IC 是处于那一个生产阶段。

1. Advance Information

这是正在设计 IC 阶段或初次制造的 Data Sheet，主要的规格还要经过生产制造后再确认，所以规格是随时会更改的，这类资料只有在 IC 设计厂内流通，不会开放给一般使用者看。此时制造厂商会提供部分工程样品给外界测试，我们称这些样品为工程样品（Engineering Sample）。

2. Preliminary

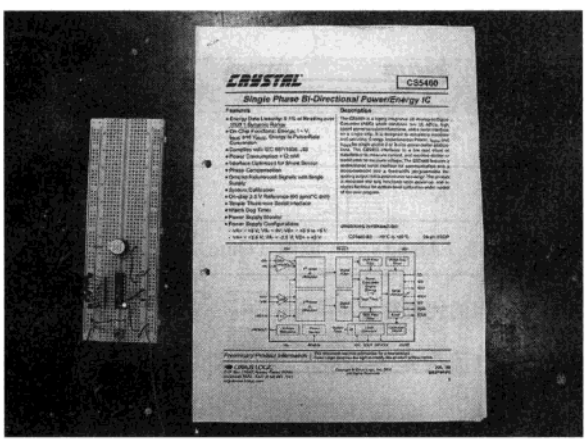
这是 IC 初次试产时暂定的 Data Sheet，有些资料尚未进行确认，所以可以在这类的 Data Sheet 上看到 TBD（To Be Defined）的空白图表或 TBD 的规格，许多特性必须要由制造厂用专门的测试机做进一步或长时间的分析。这也就是说：大部分的规格都已经确定，但是还是有少许规格是会做变动的。Preliminary 的 Data Sheet 是半开放的，一般的工程师也可以看到这样的资料。有些 IC 制造厂有了 Preliminary 的 Data Sheet 后就开始卖 IC 了，当然此时用这 IC 所做出的产品就会有某种风险存在。

3. No Identification Needed

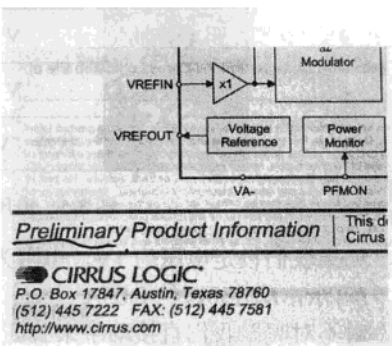
代表 IC 已经生产了，所有的规格都如 Data Sheet 上所写。但是，制造厂为了使 IC 的特性更好，还是会进行小部分的修正，这个时候采用此 IC 是最恰当的。

4. Obsolete

这枚 IC 已经停产（Discontinued）了，而这里公布的资料是供参考用的。设计者不应该再选择该 IC，以免随时有断货的可能。通常负责任的制造厂会在停产前在公司的网页上告知，这消息也会转到正式代理商那边，以方便设计者尽早换用新的元件。一般的 DIY 使用者比较难获得此一类信息，但是若你发现某个 IC 越来越难买，或是越来越贵就要警觉原厂可能准备停产了。



(a)



(b)

图 3-4 CS5460 DataSheet，图左下方有 Preliminary 的字样，代表这些资料是有可能再做修正的

3.3 电流、电压和电阻的关系

在学习 8051 单片机程序之前，当然要对基本的电学知识有一些理解，而在众多理论当中，欧姆定律 *V=IR* 也许是最最重要的一个。口语话一点的描述就是电压等于电流与电阻的乘积。在我们的实际应用上，往往电流 *I* 是未知的，而电压 *V* 与电阻 *R* 是已知的，我们借用 *I=V/R* 反算出电流值。

在实际的测量上，我们可以用数字电表以并联的方式去测量电压 *V* 与电阻 *R* 的值。但是电流值的检测时，就需要以串联的方式将电表切换到电流挡上，才可看到电流 *I* 的值。这种电流的测量方法比较不实际而且也容易因操作错误而损坏电表，通常，我们不建议用直接串联的方式取得电流值。

电无处不在，周围几乎都可看到电的应用，电表上读取单位是 V（伏特）的就是电压挡，可是电压到底是何物呢？在物理现象上，水的某些特性与电相似，所以先来想想水是如何流动的。

家里的储水塔若摆在一楼的话，我们是无法在水龙头上取水的。主要的原因是水压不够。若将储水塔摆在顶楼，距离地面有一段距离的话，它就有足够的水压将水经由管道再从一楼的水龙头中排出水来。如果储水塔的高度越高，我们在一楼水龙头所感受到的水压也就越大。电压在这方面跟水压是完全相同的。

怎么控制水量呢？如果我们需要较大的水量时，当然就选择较大口径的水管，反之亦然。在电压方面，我们则是用电阻来决定通过的电流量，电阻就好像水管口径一样，决定水量的大小，这时欧姆定律就派得上用场了，*I=V/R*，由于电阻 *R* 是在分母上，电阻越小会使电流 *I* 值越大，相反情况下，电阻 *R* 越大流过的电流也会越小。

在 1826 年，德国物理家欧姆为这三者间的关系定义为：

电压（*V*）=电流（*I*）和电阻（*R*）的乘积，*V=IR* 也因此称作欧姆定律，在欧姆定律里明确地划分出三者间的相对关系，电压和电流两者为正比关系，而电压恒定时，电流和电阻为反比的关系。

	电压 Voltage	电流 Current	电阻 Resister
代表符号	V	I	R
单位符号	V	A	Ω
单位读法	伏特（Volt）	安培（Amp）	欧姆（Ohm）

图 3-5　电压、电流和电阻的代表符号和单位

如果我们设定电压为 1V，电阻为 30Ω，那电流会是多少呢？用公式算算看。

I=V/R，*I*=1/30，约 0.03A

对不对呢？我们直接动手来验证答案。

首先，先用电表量出电阻的值，得到电阻值为 30.0Ω，接下来设定电源供应器的电压为 1V，我们来看看电流是不是如同我们计算出来的结果呢？

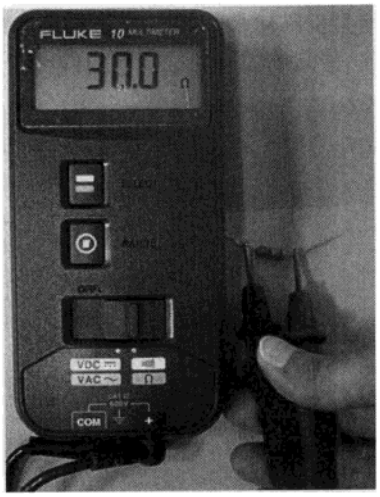


图 3-6　以上是电表，我们用测棒量出这个电阻的阻值为 30.0Ω（误差值的正负 0.1）

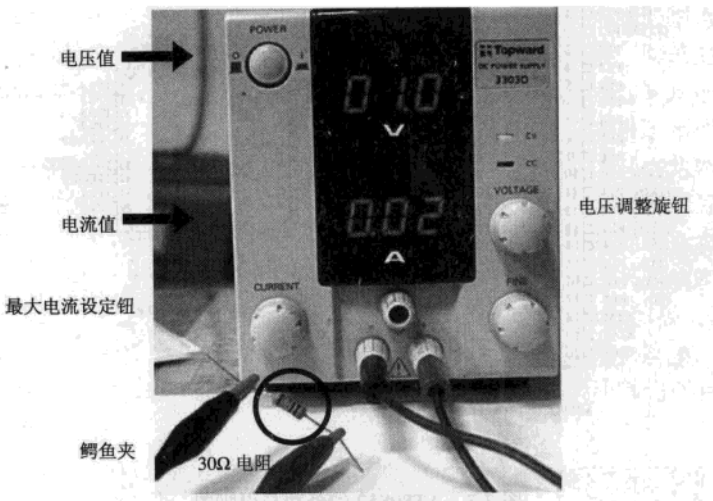


图 3-7　电源供应器

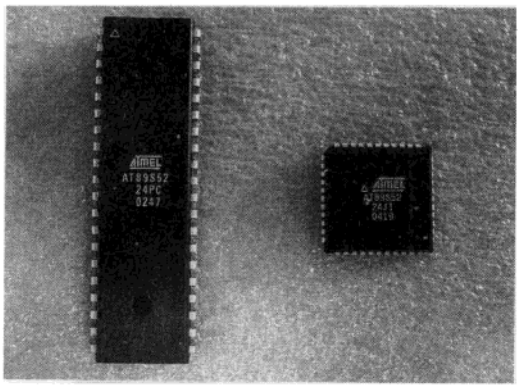
图中是电源供应器，在测量电流大小前，需将电压值调整为 1V，然后接上 30Ω 电阻后发现，此时的电流为 0.02A，明显的与我们计算出来的 0.03A 误差 0.01，这是电源供应器电流挡读值产生的测量误差。

这时要注意，无论我们使用测棒或是鳄鱼夹进行测量时，测量的点越接近待测物，误差会越低。

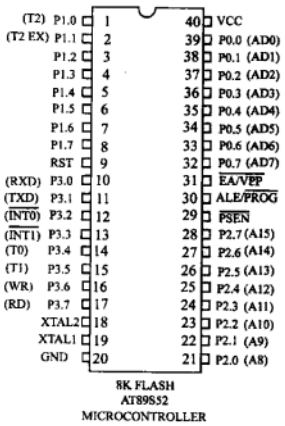
3.4　认识元件——8051 单片机

本书所做的所有 DIY 示范都是以 ATMEL 的 AT89S52（40 个引脚）与 AT89C2051（20 个引脚）为主，这两个单片机都是市场上的主流，而且很容易在一般的电子产品商店里购得。

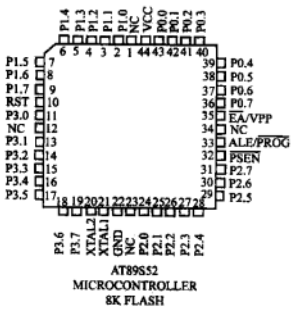
AT89S52 程序空间可达 8KB 而且引脚较多，可以做较多输入/输出的应用。AT89C2051 引脚较少，所占的面积也小，示范一些简单的应用或自己 DIY 时，我们会以 AT89C2051 为主。



(a)

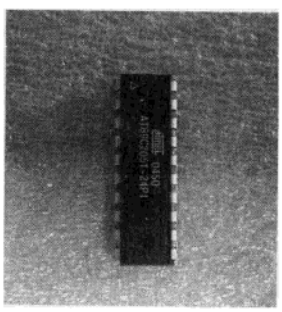


(b)

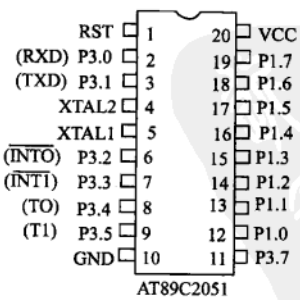


(c)

图 3-8　AT89S52 的外观与引脚图



(a)



(b)

图 3-9　AT89C2051 的外观与引脚图

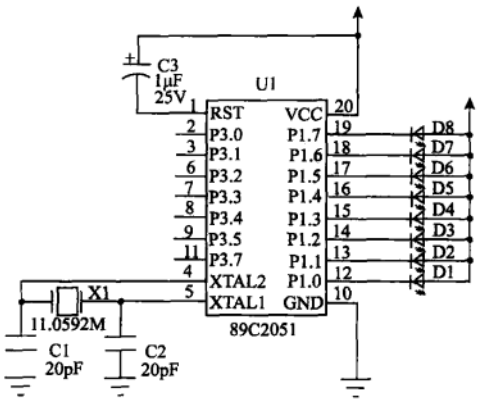


图 3-10　利用 AT89C2051 所设计的指示灯电路图

8051 单片机之所以吸引人，就是因其电路很简单，只要加上石英晶振和+5V 电源，在系统 RESET 引脚加上 1μF 的电解电容，再烧录你自己写的程序到单片机内，这个简易的控制系统就开始工作了，图 3-10 是以 AT89C2051 为 CPU，一个非常简单的指示灯显示电路。只要按照书上的章节研读，您也可以完成整个程序的编写与验证。

“DIY 从自己开始”是本书一直在灌输的观念，8051 的电路不难，参考电路上网找的话有一大堆，编写程序也不如想象中的难，难的是静下心来，在没有外在干扰的情况下，好好地在电脑前看一段本书的程序范例，再完成您自己设计的程序。

3.5　认识元件—XTAL 石英晶振

单片机系统中需要一个非常非常稳定的时钟来作为所有操作的时间依据，而这个稳定的时钟来源就是石英晶振，英文称为 XTAL。石英晶振内部是一小片经过仔细雕刻的石英片，石英两端用引线拉到外部，加上金属隔离罩后就成为常见的石英晶振外观。

石英晶振不会自己振荡，它要配合外部的反相放大电路，就可以让石英晶振开始振荡。只要一加上电源后，石英晶振会因电路不稳定的关系开始振荡，而且一起振后，就一直很稳定地保持在该振荡频率上，请注意石英晶振端上各有两个小电容是需要的，其值都在数 10pF 上下。一般商业用途的石英晶振的频率误差都可以保持在百万分之一百以内，即小于 100ppm，对于一般的控制用途已经足够了。

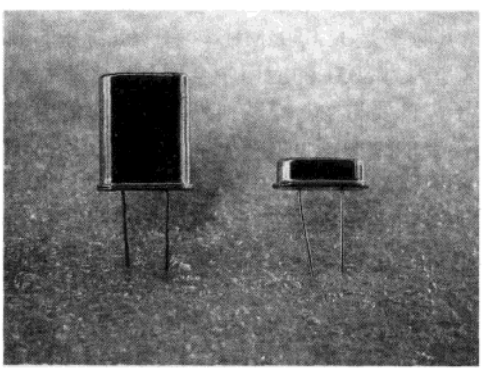


图 3-11　石英晶振的外观，左为 49U 的规格，右为 49US

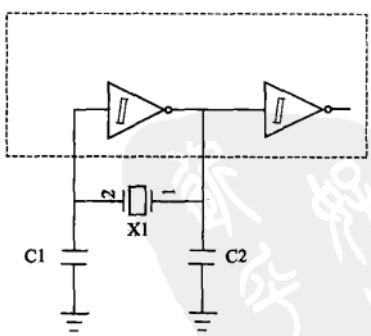


图 3-12　标准石英晶振的振荡电路范例

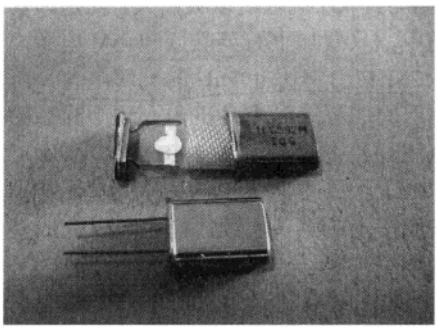


图 3-13　石英晶振的分解图

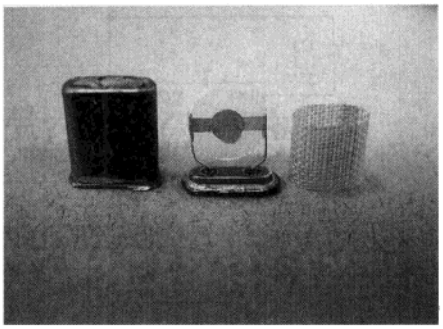


图 3-14　石英晶振的内部构造图

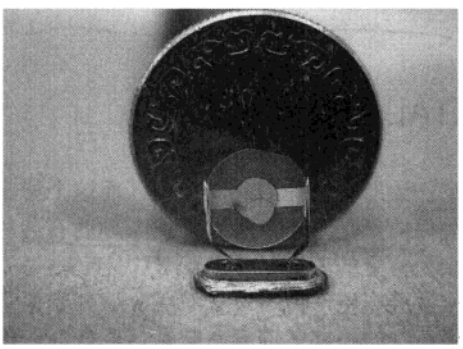


图 3-15　石英片的近拍照

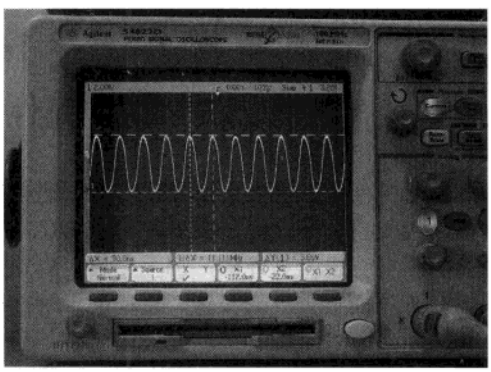


图 3-16　石英晶振的振荡波形

图 3-16 用示波器观看石英晶振产生的振荡波形，看起来不太像方波，单片机内部还要再做处理，将其整形成方波。

在单片机 8051 的电路中会有一颗石英晶振，其频率是 11.0592MHz，有一点奇怪的频率值，不过接触越多就不会觉得这个频率奇怪了。这是因为 8051 的应用中会用到许多串行通信的例子，若要与 PC 或其他外部设备通信连接，系统需要的频率经过换算后正好是 11.0592MHz 这个频率值。

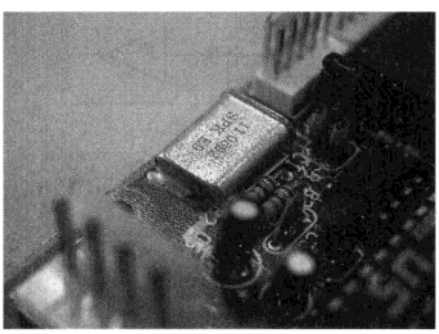


图 3-17　XTAL 使用范例

图 3-17 是 8051 控制板上使用 XTAL 的范例，石英晶振的金属外罩应该接地，以避免振荡时的 RF 噪声溢出。

DIY 时如何判断石英晶振的好坏呢？最快的方法就是加到电路上，直接用示波器或计频器直接观察其频率值了。请看图 3-18 的示范电路，AT89S52 不需要加任何程序，通电后直接检查 XTAL2 端点的频率或是 ALE 上的频率值。

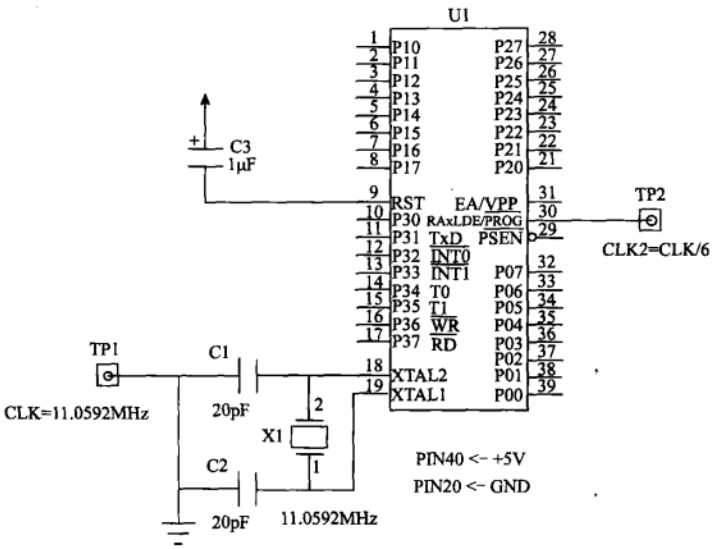


图 3-18　检查石英晶振好坏的示范电路，直接使用 AT89S52 内部的振荡放大电路，检查点分别是 XTAL2 与 ALE，ALE 的频率刚好是石英晶振振荡频率的 1/6

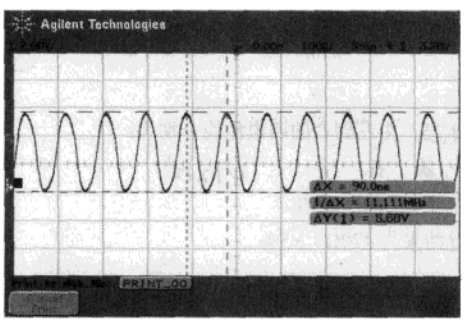


图 3-19　用示波器的计算功能看到的频率值为 11.11MHz

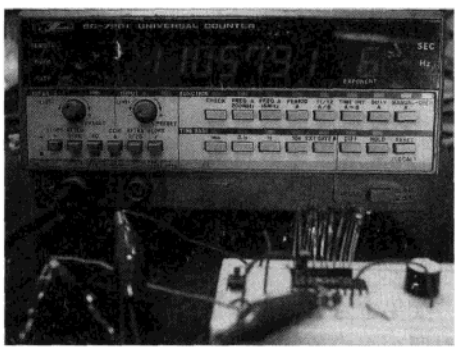


图 3-20　用计频器看到的频率值为 11.05791MHz，这个值会比示波器所计算的准确

3.6　认识元件——电阻

电阻器，按照 *V=IR* 的公式我们知道在电压固定时，电阻和电流是成反比的。电阻的主要功能是在电路中限制电流量，或提供分压、降压的功能。配合使用场合的不同，电阻有不同的样子和电阻体，而我们 DIY 最常用的是 5%误差的碳素电阻。

电阻值的单位是 Ω（欧姆），在图 3-21 中最上方和最下方的电阻值都已经标示在上面，而中间三种电阻因为体积关系无法进行阻值标识，而改以四圈的色码线来标示该电阻值的大小，该怎么用这一道道色线该来判断电阻大小呢？碳素皮膜和金属氧化皮膜一般会有四道色线，最

后一道和前三道色线会隔一小段距离，下面的色码识别表就是电阻值与色码间的相对关系表。

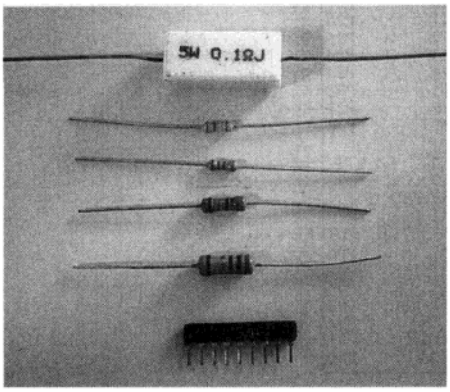


图 3-21　由上往下分别为水泥电阻，中间三个是碳素皮膜电阻，最后一个是排阻

色名	第 1 色带 第 1 数字	第 2 色带 第 2 数字	第 3 色带 第 3 数字	第 4 色带 标准电阻值的容许误差
黑	0	0	10^0	
棕	1	1	10^1	±1%
红	2	2	10^2	±2%
橙	3	3	10^3	
黄	4	4	10^4	
绿	5	5	10^5	
蓝	6	6	10^6	
紫	7	7	10^7	
灰	8	8	10^8	
白	9	9	10^9	
金			10^{-1}	±5%
银			10^{-2}	±10%

图 3-22　四组色码标示表对照后第一数字和第二数字乘上第三数字，最后再加上容许误差，就是该电阻值

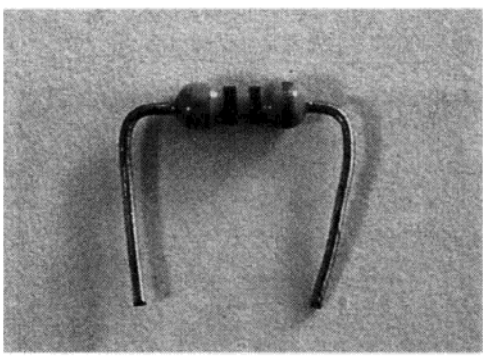


图 3-23　由左到右的四道色线分别为：

黄　紫　棕　金

4　7　10^1　±5%

计算结果为 470Ω　±5%

如果电阻第一道线是黄色，第二道线紫色，第三道线棕色，最后一道线金色。接下来只要对照列表中各颜色所代表的数字，再经过计算 47×10^1±5%=470Ω±5%，就知道该电阻值为 470Ω，容许误差为±5%。

再看另一个电阻，第一道线是棕色，第二道线是黑色，第三道线是橙色，最后一道线是金色，我们排列一下：

棕　黑　橙　金

10^1　10^0　10^3　±5%=10 000±5%Ω

上面电阻的值为 10 000Ω（10×10^3），由于这样的表示法很容易混淆，因此我们会用 k 来表示 10^3，上面这个电阻值就写成 10kΩ 而不是 10 000Ω。如果电阻值为 1×10^6Ω，而 10^6 可以用 M 来取代，我们就用 1MΩ 来表示。

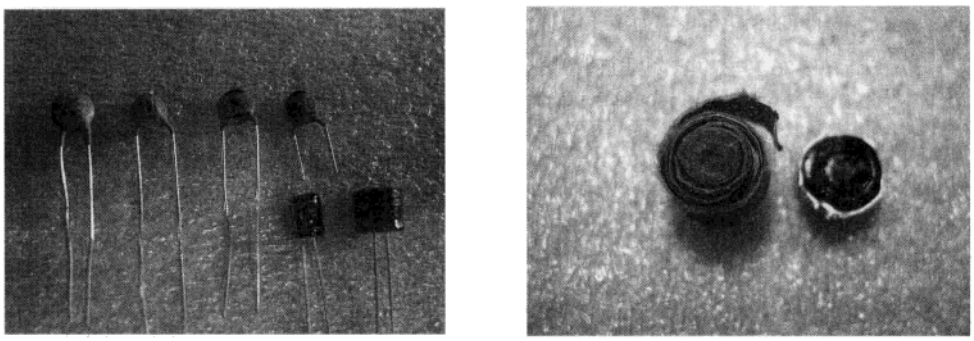
对初学者而言，电阻的色码辨认是要花一小段的时间来熟悉的。一拿到电阻后，首先找到金色码的位置，然后从金色码的对面开始读取色码，第三码若是橙色时就套上 k，绿色时为 MΩ，再默背“黑棕红橙黄绿蓝紫灰白”的排列就可以了。

其实认识电阻色码的规则，更好的方法是到电子元件行买数十个不同阻值的电阻，回家

一个一个进行解读，并且配合数字电表的电阻挡再做确认，只要一个下午的时间就可以让你完全弄懂电阻的色码了。你也可以随意找一片布满电阻的电路板，针对上面的电阻一一去判读，重复几次后就非常熟悉了。

3.7 认识元件——电容

两片平行金属片如果距离很接近的话，就形成所谓的电容，由于金属片很接近但又不短路，金属片上累积电荷后就变成一个带电的装置。为了使电容能累积更多的电荷，真正的电容的两片金属片是呈卷曲状的，以便有更大的面积来储存电荷，请看图 3-24 右边的电容解剖图。



(a)　　(b)

图 3-24　常见的铝质电解电容、陶质电容，右边为电解电容的解剖图

电容的单位是 F（法拉），常见的电容都没有到 F 如此大的电容值，一般的电容都在 μF 的等级，即 F 的百万分之一（10^{-6}）。像石英晶振旁的 20pF 电容，其容量更小了，小到只有 F 的 10^{-12}。电容量超过 1μF 以上的电容因为制造与材料的关系会有极性的关系，而小于 1μF 的电容就没有极性之分了。所谓的极性是指该电容是有正负极的，即正端的电压一定要比负端为高，如果反接的话，该电容是有爆炸的可能。电容内部的平行金属片与施加的电压也有关系，当加在电容两端的电压过高时，有可能造成金属片因耐压不够发生短路，所以电容有两个必要的规格：容量与耐压，标示耐压 25V 的电容，就不能在其两端施加 35V 的电压。

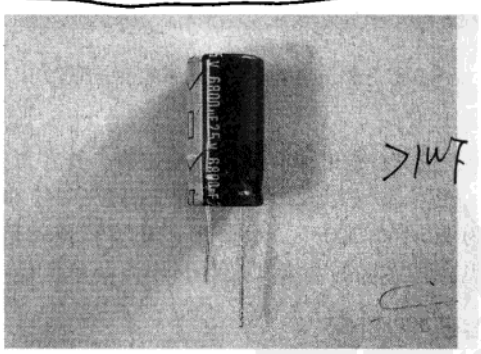
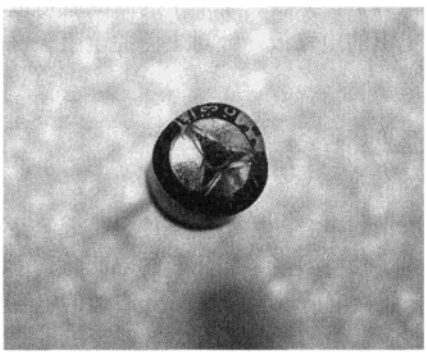
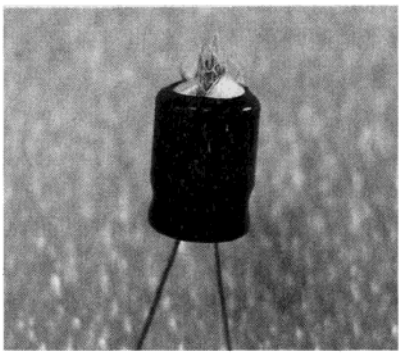


图 3-25　这是铝质电解电容，工厂在制造时会故意将正端的引脚加长。另外在电容器的负端增加特别的标示

电容器的基本功能就是充电和放电，而其充放电的速度跟电容周边的元器件有直接的关系。电容主要是依电介质材料、电极的构造和使用的目的等来分类，一般而言，铝质电解电容和陶质电容在 DIY 实验中是最常接触到。



(a) 被炸开的铝电解电容面貌



(b) 炸开后的侧面图

图 3-26　电解电容电源反接加上额定电压，并持续一段时间（10s）后，电容膨胀后爆开损坏，并喷出滚烫的电解液

电容的单位除了刚刚看到的 $\mu=10^{-6}$ 外，还有 $n=10^{-9}$ 和 $p=10^{-12}$，所以如果认错单位其容量可能就会差上 1000 倍，这是我们 DIY 时要一定要特别注意的。在实现上 n 级的单位反而很少看到，所以我们更要注意电容的单位。

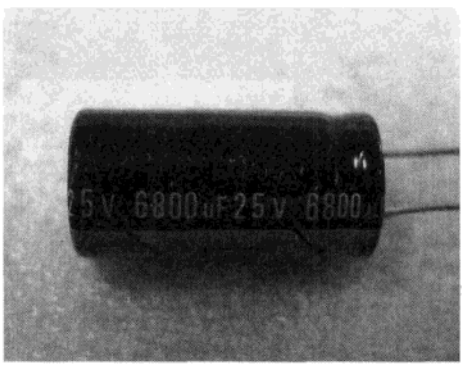


图 3-27　铝质电解电容，通常电容量都会标示在上面

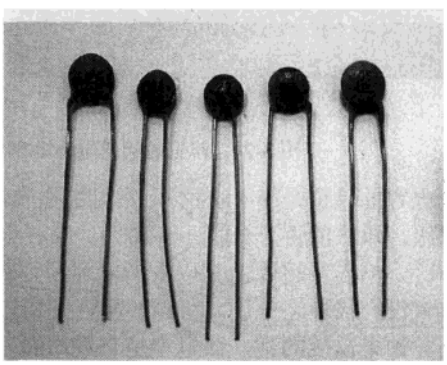


图 3-28　由左到右分别标示为 104/201/ 20/7/.05 的陶质电容

图 3-28 是几种常见的陶质电容，其容量是印刷印上去的，电容的标示并未像电阻如此统一，较容易产生混淆。若印有三码数字时，则第三码为其科学符号，所以标示 201 的电容容量为 $20\times10^{1}=200pF$。标示 20 的电容容量就是 20pF，标示 7 的电容容量就是 7pF，但是“.05”的电容其真正的容量却是 0.05μF，其单位是 μF。

电容值可以量吗

电容值是可以测量的，图 3-29 是专业用的 LCR 表，它可以直接设定频率后量出电容值来，其准确度都在 0.5%以内，但是这种仪器的价格接近几万元，这个价位很明显的不适合一般 DIY 级的人。退而求其次，我们可以用较高级的数字电表上的电容挡来进行测量，如图 3-29 与图 3-30 所示，此时的测量值仅供参考，因为数字电表的主要的强项还是在交直流电压

的测量，至于电容是可以量但准确度就差了一点，而且测量范围应该在 μF 上下。

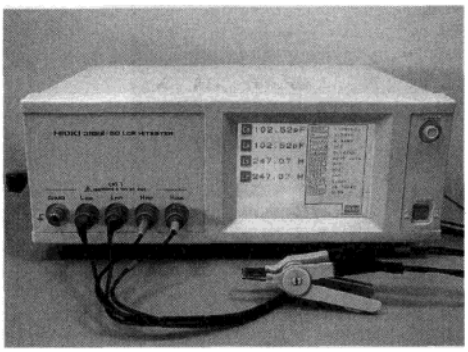


图 3-29　专业的 LCR 表，可度量电感 L、电容 C 与电阻 R，100pF 的电容测起来的值为 102.52pF。不过其价格不适合一般 DIY 的使用者

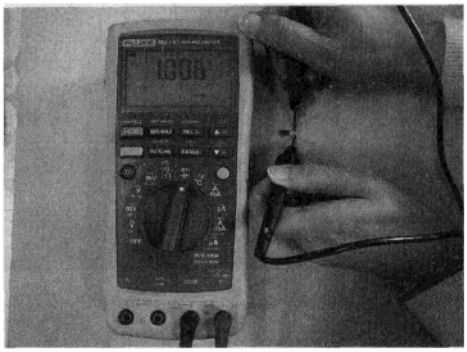


图 3-30　用数字电表的电容测量挡，去对 1μF 的电解电容进行检测，得到结果为 1.006μF

如果用数字电表的电阻挡对较大电容的电解电容测量时，测棒刚接触的瞬间，电阻值几乎是 0，随后阻值一直往上升，升到数百 k 甚至数 MΩ 为止。一般的数字电表只能对电容做简单的充放电特性判断，如果电阻挡测量时，发现电容两端的阻值接近 0 或固定在某个低阻值，那肯定是这个电容内部短路坏了，千万不要放在 DIY 的电路里，请直接就丢到垃圾桶里，以免后患无穷。

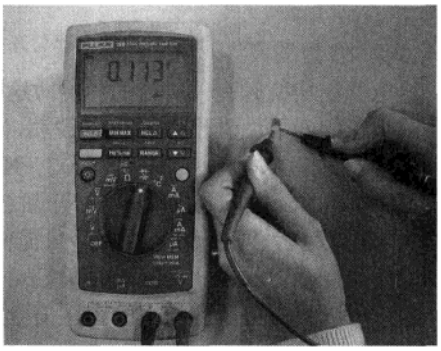


图 3-31　用数字电表的电容测量挡，去对 0.1μF 级的电容进行检测，得到结果为 0.113μF

在 DIY 自行购买电容时，有几点要铭记在心：

（1）μF 等级以上的电容大部分都是有极性的电解电容。

（2）由于制造的问题，电容的误差都在 10%以上，无法像电阻有 1%以内的误差。

（3）电容的特性跟工作频率有直接关系。

（4）电解电容是有极性的，焊接时绝对不可以接反。

3.8　认识元件——继电器

继电器（Relay）的作用跟手动开关（Switch）一样可切换信号，不过开关需要借用手来做切换，而继电器是用电的信号来进行切换。图 3-32 是继电器与开关的实物图与电气符号，从透明的继电器上看过去，其内部有一个相当明显的线圈，与线圈垂直的地方有一簧片，当我们对线圈施加额定的电压时，线圈会产生磁力，吸引簧片进而连动一机构。当施加的电压不见时，簧片被一组弹簧拉回原来的位置，达到类似 Switch 开和关的效果。

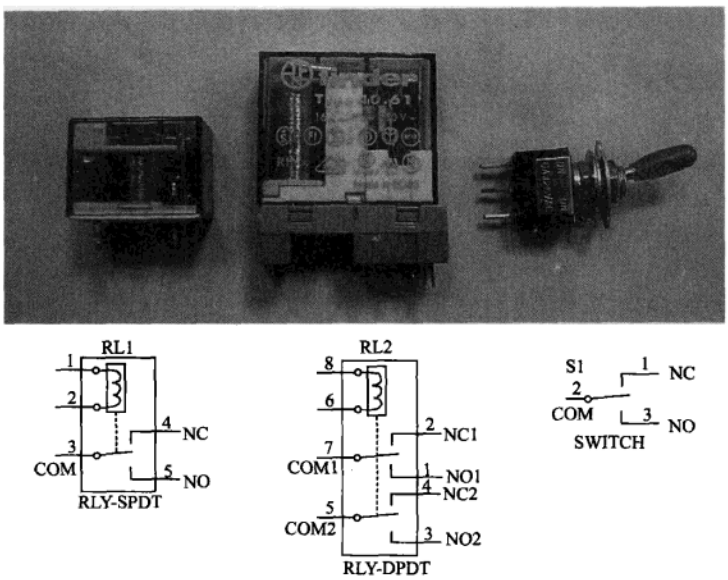
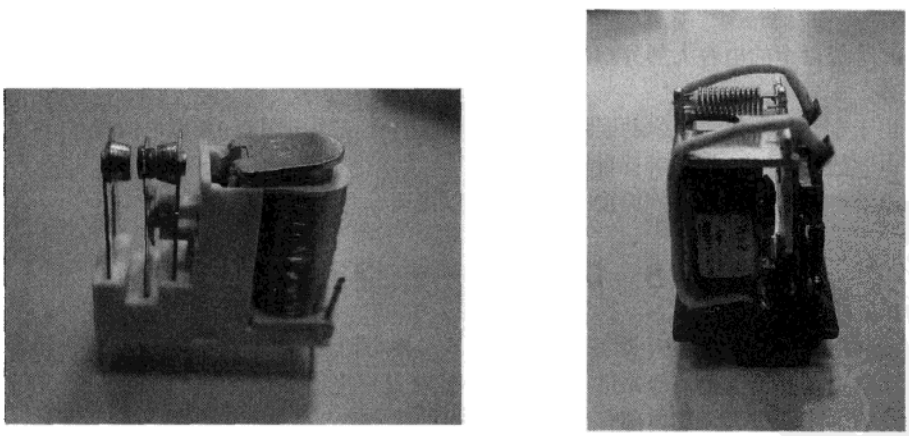


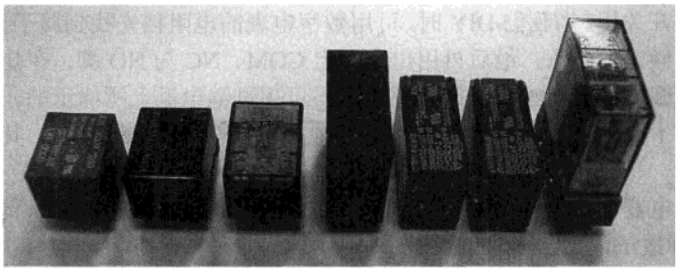
图 3-32　继电器与开关的实物图与其对应的电气符号

继电器在切换时有一清脆的声响，这是由簧片改变位置所造成的。而簧片所连动的机构上才是真正的开关，请看图 3-33 的说明，通常一个开关有三个触点（COM、NC 和 NO），当簧片不工作时，共同点 COM 点和 NC（Normal Close）常闭触点是相通的，线圈通电后，改变成 COM 和 NO（Nomal Open）常开触点接通，这也就是说 COM 是一个共同点，而开关在 NC 和 NO 点间切换。一般小 Relay 只提供一组开关，如图 3-34 所示。由以上的说明我们可以知道：继电器上的线圈与三个开关触点，在电气上是完全隔离开的，两方是利用机构连触在一起，所以我们可以用一小的电压信号加在线圈上，进而控制一个大灯泡的亮与灭。

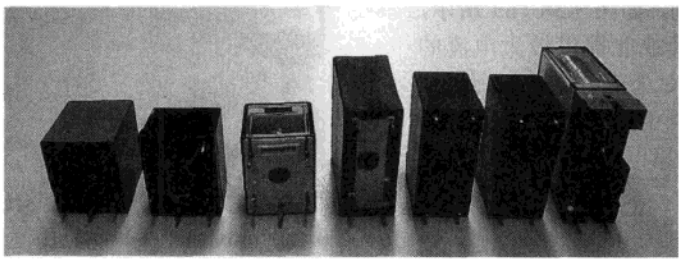


(a)　　(b)

图 3-33　继电器上的开关触点分别是 COM 共同点、NC 常闭触点和 NO 常开触点，请特别观察它的连动机构



(a)



(b)

图 3-34　常见的继电器，各有一组或二组开关

继电器内部最重要的组件是什么？答案是触点，请看图 3-35。继电器的触点都是镀银的，这样才能允许大电流的通过。如果是一般的铜触点时，使用久后会氧化，使得整体触点的阻值升高，如果再经常切换大电流的话，继电器就会因发烫而损坏。所以，每个继电器的外壳上都会标示该继电器可承受电流的值，这个值越大则其内部的触点就越大且越讲究。图 3-36 是几款常见的继电器外观，我们除了要确认其通过电流的大小外，还要知道继电器上的线圈要加多少电压才会工作，这个值在图 3-36 继电器的外观上都有标示。

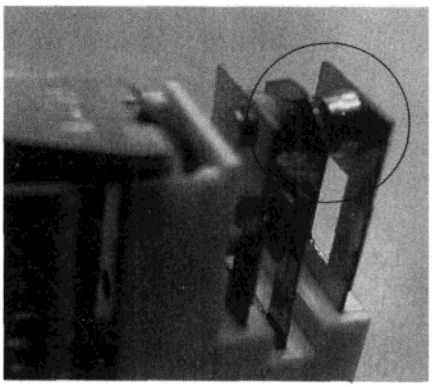


图 3-35　继电器内最重要的组件：开关上的镀银触点

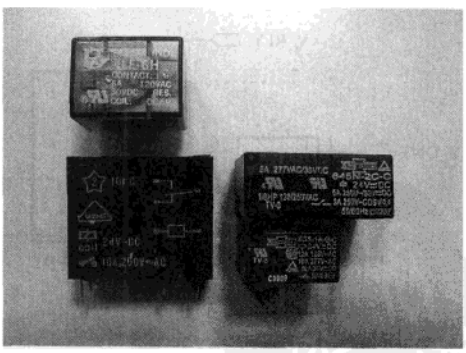


图 3-36　几种常见继电器的上视图，请注意上面一定有线圈的规格，与其容许的电流值，而且每个电压下允许通过的电流也不一样

图 3-37 是只有一组开关的继电器底视图，总共有 5 个引脚。其中两个是控制线圈的，另

外三个引脚是做开关用。当我们 DIY 时，可用数字电表的电阻挡先找到属于线圈的两个引脚，然后对线圈施加额定的电压，最后再用电表判定 COM、NC 与 NO 脚。在使用继电器时要注意两点：加到线圈的电压要够，还有切换电流不可超过继电器上所标示的。继电器的使用寿命都在可切换数十万次以上，但是若使用不当时，也可能几百次就毁了，其实就是切换的触点损毁。

在实际上继电器该如何驱动呢？请看图 3-38 的几种接法，当继电器的线圈只要 10mA 左右就会工作时，可以用 TTL IC 直接驱动，如图 3-38（a）所示，也可以用晶体管或能提供较大电流的 ULN2003 专用 IC 驱动。DIY 时还要注意到一点：当线圈由接通转变成不通时，会在线圈上产生反电动势（反电压），在电路上要用一枚二极管将这个电压吸收掉，若省略此元件经常会导致 Relay 驱动电路的损坏。

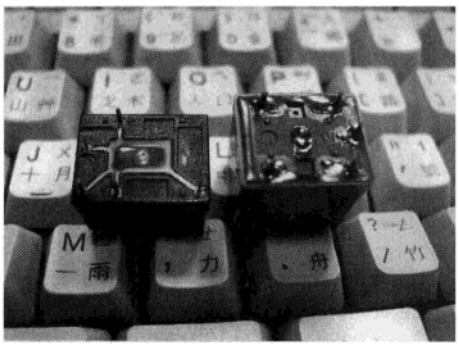
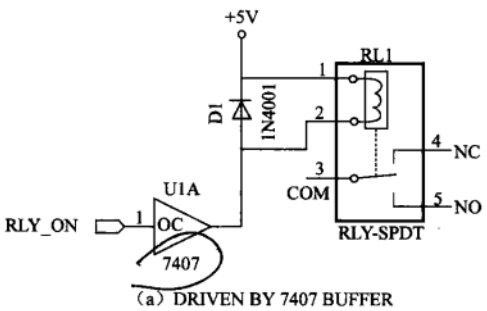
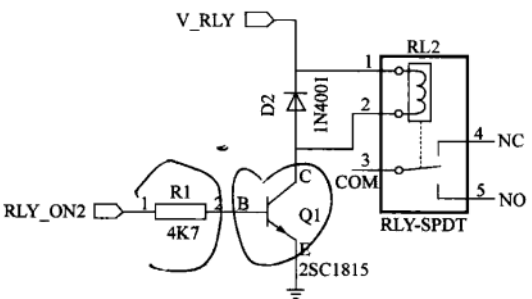


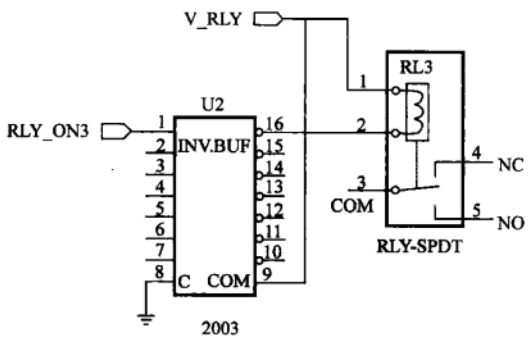
图 3-37　继电器的底视图，要先找出线圈的引脚，然后再找出 COM 和 NC 点，最后一点就是 NO 点



（a）DRIVEN BY 7407 BUFFER



（b）DRIVEN BY NPN TRANSIS TER



（c）DRIVEN BY ULN2003

图 3-38　几款继电器的驱动电路示范（a）7407 驱动；（b）三极管驱动；（c）ULN2003 驱动

注意：线圈旁的二极管是不可以省略的，ULN2003 内部已经有此二极管所以（c）并未外加保护二极管。

3.9 逻辑状态 1 和 0

在数字的世界里只有 1 和 0，而在数字硬件电路上也有 1 和 0 的区别，这代表着 0 和 1 是完全颠倒的，若是 0 就不是 1，而且不是 0 就是 1。

在真正的数字电路的定义上，若供应电压是+5V 的话，我们是以电压 2.0V 以上为数字 1，而电压 0.8V 以下为数字 0。这也就是说，所有的电路都要把它的输出维持在 2.0V 以上或 0.8V 以下。所以，2.0～5.0V 就是所谓的高电平 1，而 0.0～0.8V 就是低电平 0，硬件电路输出的电压值若落在 0.8V～2.0V 的地方，就是不对的，它会造成其他逻辑电路的判断错误。

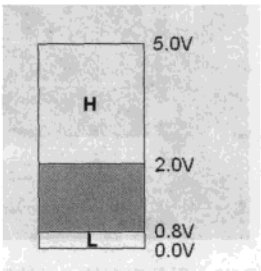


图 3-39 数字 0 和 1 的分界，其中有 1.2V 的灰色地带

recommended operating conditions (see Note 3)

		SN5404			SN7404			UNIT
		MIN	NOM	MAX	MIN	NOM	MAX	
V_{CC}	Supply voltage	4.5	5	5.5	4.75	5	5.25	V
V_{IH}	High-level input voltage	2			2			V
V_{IL}	Low-level input voltage			0.8			0.8	V
I_{OH}	High-level output current			−0.4			−0.4	mA
I_{OL}	Low-level output current			16			16	mA
T_A	Operating free-air temperature	−55		125	0		70	°C

NOTE 3: All unused inputs of the device must be held at V_{CC} or GND to ensure proper device operation. Refer to the TI application report, *Implications of Slow or Floating CMOS Inputs*, literature number SCBA004.

图 3-40 TTL IC 的输出规格资料（摘自 TI 网站）

数字状态 0 和 1 最佳的观察方法有两种。若是静态的 0 和 1，可以用数字电表 DCV 电压挡直接测量，除了可以判断状态外，还可以观察电平的电压值。如果是一直在变化的 0 和 1 时，就要用数字式示波器看了波形，它可以观看的信息就多了，第一可以分辨数字符状态，第二可以真实地看到波形变化及其电平，第三还可以换算出处于各个状态的时间或频率值，这些对我们 DIY 时的帮助是最正面的。

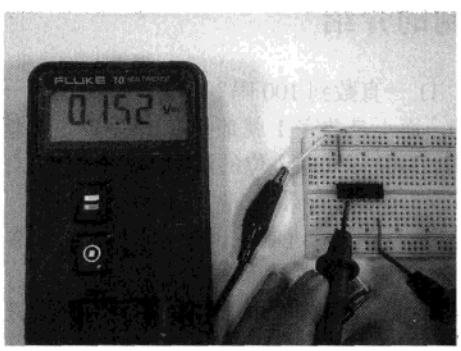


图 3-41 用数字电表观看数字状态 0

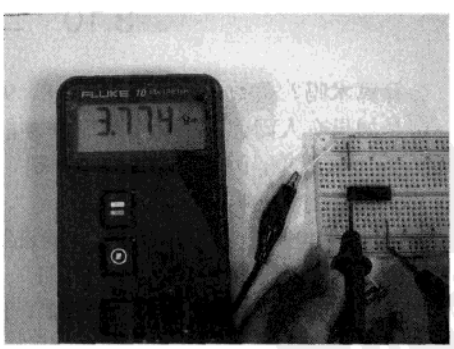


图 3-42 用数字电表观看数字状态 1

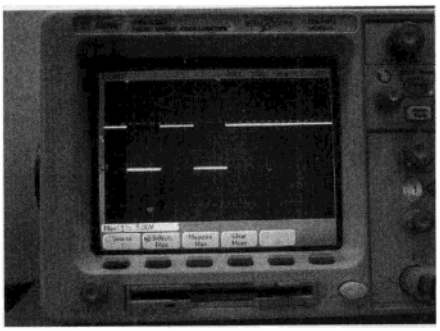


图 3-43　用数字存储式示波器看到完整的数字波形

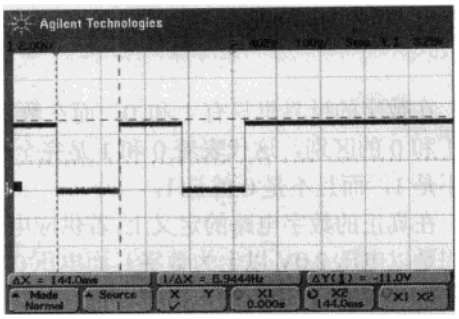


图 3-44　数字式示波器除了电平以外，还可以测量时间

使用逻辑笔（Logic Probe）来看数字状态也是可以的，它的优点是体积小且携带方便，它可以观看静态的 1 和 0，通常 1 以红 LED 灯代表，0 用绿 LED 代表。至于一连串的 01 变化时，则以闪烁的黄 LED 灯号做指示。不过，对于较复杂的单片机电路，其 0 和 1 的变化不完全是规律的，我们还是强烈建议用数字式示波器来检测，才是较为正确的作法。

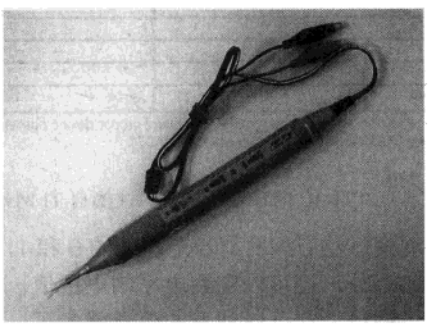


图 3-45　逻辑笔的外观

3.10　二进制的介绍

会算算术吗？会数 1、2、3、…、8、9、10、11 一直数到 100 吗？相信大多数的人一定会。可是如果有人问你，为什么数完 9 才能数 10，而不是数完 1 就数 10，或是数完 7 就数 10 呢？答案很简单，因为数是用“十进制”来数嘛！可是你知道进位的法则除了十进制以外，还有其他的进制方式吗？

在 8051 的世界里，有两种很常用的进制法则，一种叫“二进制”；另一种叫“十六进制”，这两种进位法则远比十进制受欢迎多了！

先谈二进制好了，所谓的“二进制”，顾名思义，就是数到 2 就进位，换句话说，在二进制里除了 0 跟 1，是看不到其他数字的。再问个问题，准备好了吗？如果要用二进制来表示 5，应该怎么做呢？

第一种方法，就是一个一个慢慢加：

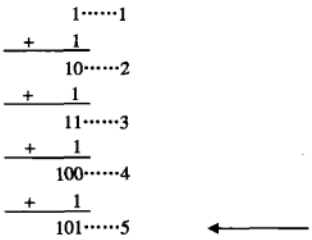


图 3-46　以二进制方式表现 5，答案是 101，数到 2 要记得进位

加出来了吗？如果用二进制来表示 5，答案就是 101。再请算算看 15 用二进制该如何表示呢？先别急着数，这边有一种用乘法就可以计算的方法：

$15=2^3\times\underline{1}+2^2\times\underline{1}+2^1\times\underline{1}+2^0\times\underline{1}=8+4+2+1$

看清楚了吗？15 用二进制的表现方式就是 1111。再给你一次机会，50 的二进制表现方式为何？

$50=2^5\times\underline{1}+2^4\times\underline{1}+2^3\times\underline{0}+2^2\times\underline{0}+2^1\times\underline{1}+2^0\times\underline{0}=32+16+0+0+2+0$

答案就是 110010，这个方法应该有比一个一个慢慢数快多了。那如果现在要用二进制来表现 99 呢？先别急着展现刚学到的功夫，下面还有用除法来计算的好方法。

用二进制表现 99，首先把 99 当被除数，进位数 2 当除数，计算后会得商数 49 余数 1，第一个得到的余数就是“最低位数”，再将商数 49 也除以 2，会得到商数 24 余数 1，继续把商数 24 除以 2，得到商数 12 余数 0，接下来重复相同的做法，把得到的商数当成被除数一直除以 2，直到商数为 1 才停止，最后一个得到的被除数（也可以称做商数）即为“最高位数”，往下来看看计算出来的结果吧。

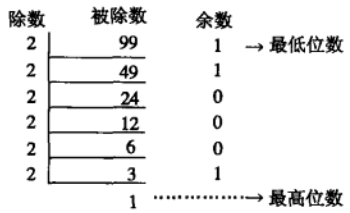


图 3-47　将所有的余数，由最高位写起，可以得到 99 二道位的表现方式为 1100011

答案就是 1100011。再出一个题目来测验刚刚学习的成果，数值 68 的二进位该如何表示？

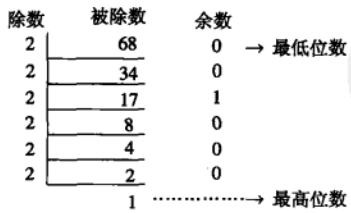


图 3-48　计算 68 的二进制表现方式，一样由最高位数写起得到的结果是 1000100

答案就是 1000100。事实上，方才学到的两种计算方式，都可以快速得到我们需要的结果，无论自己喜欢那一种方式都可以，只要计算时感到方便就行，剩下的熟练程度就需要靠自己多多练习了。

3.11 认识十六进制

了解了二进制，现在可以挑战十六进制了。同理，十六进制就是数到 16 就进位。动脑思考一下，当我从 10、11、12、13、14、15，接下来呢？先拿出纸笔来除除看好了，此时的进位制是十六进制，所以除数应为 16。

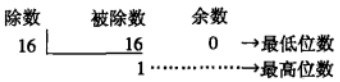


图 3-49 计算 16 用十六进制的表现方式，一样由最高位数排到最低位数得到 10

答案是 10，由此可知，16 用十六进制的表现方式就是 10。再出一个题目来测验自己是否真的理解，计算 129 用十六进制的表现方式为何？

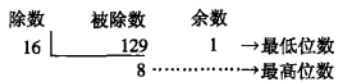


图 3-50 计算 129 在十六进制的表现方式，由最高位数排到最低位数得到 81

看到图 3-50 的算式，使用除法的计算技巧，很快就计算出结果来，答案是 81。在这时要注意：当被除数无法被 16 除，就可以停止计算了。再举个例子来加深自己的印象吧，数值 250 的十六进制表现方式为何？

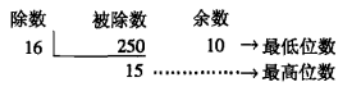


图 3-51 计算 250 在十六进制的表现方式，当被除数无法被除数 16 除时就停止计算，再由最高位数排到最低位数

一样用除法的计算方式得到 250 用十六进制的表现方式，答案是 1510？这样的写法似乎怪怪的？如果没有特别注明，我们可能明天就忘记，以为是一千五百一十吧。关于十六进制，到底该如何表示才适当呢？十进制里的 10～15 在十六进制里是否有不同的表示方式呢？

答案是有的。看到图 3-52 的表格：A 到 F 分别取代了十六进制的 10～15，为了避免产生误会，所以在十六进制的世界里，使用了 A 来代替十六进制的 10，用 B 来代替十六进制的 11，C 代替 12，D 代替 13，E 大替 14，最后用 F 来表示 15。现在来深入探讨一下十六进制的世界了！请计算十进制的 181、31 和 171 用十六进制的表现方式为何呢？

看到如图 3-52 和图 3-53 的计算结果，对于十六进制的了解又更进一步了。我们现在来回头看先前的计算式，用十六进制表现 250，答案还是 1510 吗？当然不是，正确的答案应该是 FA。

十进制	10	11	12	13	14	15
十六进制	A	B	C	D	E	F

图 3-52 十六进制 10～15 的表示方式为 A 到 F

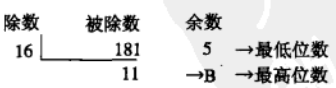


图 3-53 181 用十六进制表示的正确方式为 B5

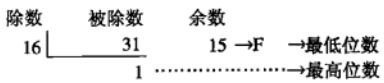


图 3-54　31 用十六进制表示的正确方式为 1F

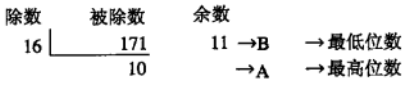


图 3-55　171 用十六进制表示的正确方式为 AB

看到这边再想一想：10 所代表的数值是多少呢？自己会开始产生疑问，现在我要表现的 10 是二进制、十进制还是十六进制呢？相同的道理，当我们在 8051 的程序里写 10，再送到编译器去判读，而且不给编译器任何提示，那会产生什么样的状况？结果是会被编译器一律当成十进制的 10 来看待处理。

所以，当我们需要编译器判读的是二进制（Binary）的 10，就用 10B 来表示；如果是十六进制（Hexadecimal）的 10，则用 10H 来表示，这样就不怕编译器看不懂跑出来质问我们了。

3.12　二进制与十六进制间的转换

在学习完二进制与十六进制后，再来要认识一个藏在电脑里的小工具。相信很多人家中都有电脑，也曾经用过接下来要操作的“计算器”这个工具，如果不熟悉的人，请到“开始”→“所有程序”→“附件”→“计算器”，先将这个工具打开，可不要小看它，它可是能快速计算二进制和十六进制的好帮手！

现在请将鼠标指到“查看”的位置，看到目前的小算盘是“标准型”的模式，接下来直接选择“科学型”的模式，点选后会出现下方“科学型”样式的计算器。

图 3-56　计算器“标准型”的样式

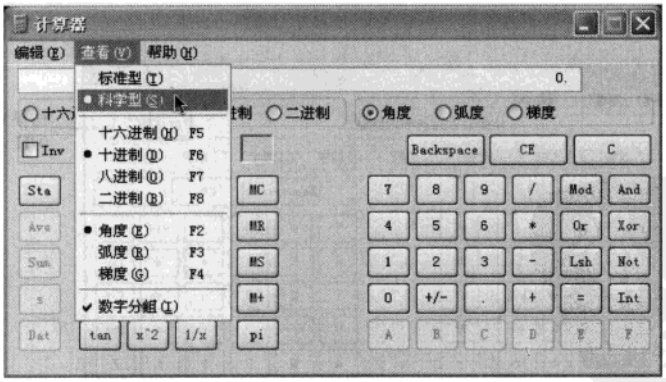


图 3-57　计算器“科学型”的操作模式

我们先看到计算器的左方，发现一些熟悉的内容，先前学习的二进制、十进制和十六进制都可以在这边实际计算，而且还多了一项八进制。现在就来了解如何使用计算器计算二进

制、十进制和十六进制。先来尝试用计算器计算 215 二进制十六进制的表示方式吧。

图 3-58　使用计算器“科学型”，单击十进制并且键入数值 215

图 3-59　使用计算器“科学型”，单击十进制并且键入 215 后，再单击二进制，得到 11010111B 的结果

操作完前面三个步骤后，很快就得到 215 的二进制表示方式为 11010111B，接下来再用鼠标单击十六进制。

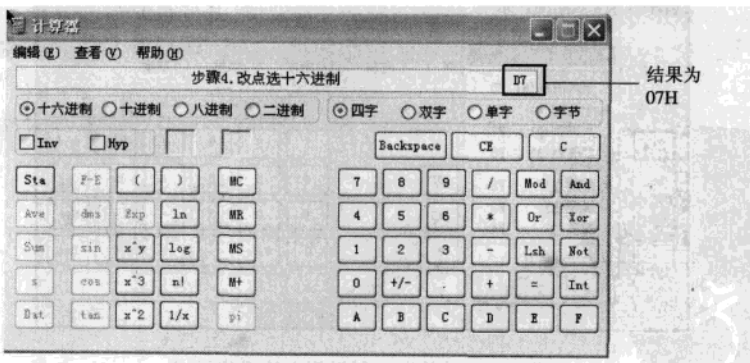


图 3-60　在得到 215 二进制的表示方式后，再用鼠标单击选十六进制，
我们很快看到数值 215 十六进制的表示方式为 D7H

一样很快就得到 215 十六进制的表示方式为 D7H。那如果我想将十六进制用十进制来表

示，是否也可以用计算器来计算呢？我们尝试用计算器，将十六进制的 8AH 用十进制来表示，仔细看下面的操作步骤吧。

图 3-61　使用计算器“科学型”，单击十六进制并且键入 8A

图 3-62　使用“科学型”计算器，单击十六进制并且键入 8A 后，将鼠标移到十进制同时单击十进制

用计算器操作上面三个步骤后，所得到结果是 138。现在无论是十进制转二进制或十六进制，或是二进制和十六进制转回十进制，甚至二进制和十六进制互相转换，都可以使用计算器这个好帮手来帮助我们得到答案。当我们想验证自己计算的答案是否正确时，计算器更是很好的辅助工具，不过千万不要过度依赖工具的使用，先前乘法和除法的计算方式也要好好的练习，因为理解二进制和十六进制的原理，在 8051 的学习过程中是相当基础且重要的一项功课，所以我们一定要认真学习。

先不要急着结束，计算器的二进制、十进制和十六进制都已经介绍完了，可是整个进位制度中，有一个八进制，到现在都还有没提到，现在该是发挥想象力的时候了，到底八进制应该用在何处呢？

您可以从下列公司的网站取得更进一步的信息：

（1）www.chipware.com.tw：查询单片机相关应用的参考资料。

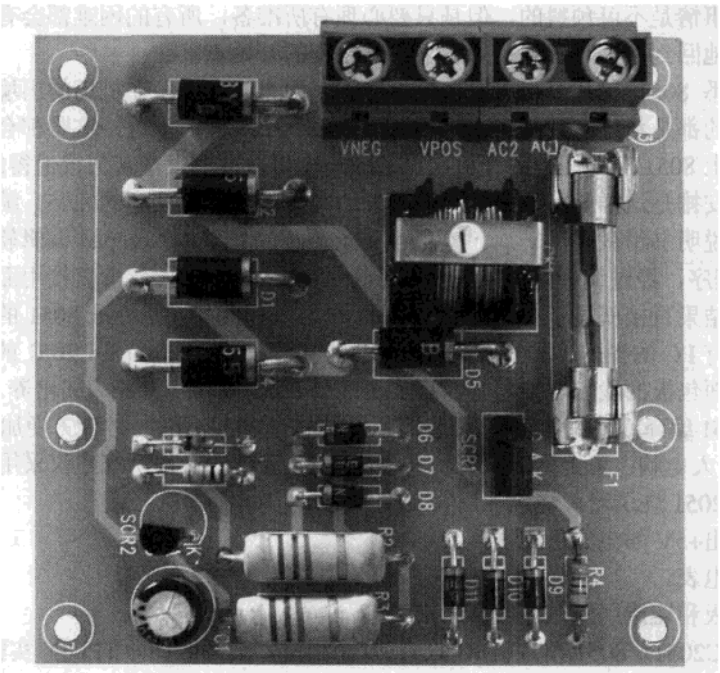
（2）www.fluke.com：查询掌上型数字电表的相关资料。

（3）www.agilent.com.tw：查询示波器的相关资料。

（4）www.topward.com：查询电源供应器的相关资料。

（5）www.iwatsu.co.jp：查询计频器（counter）的相关资料。
（6）www.hioki.co.jp：查询 LCR Meter 的相关资料。
（7）www.atmel.com：制造 8051 的厂商，有很多 MCU 的相关参考资料。
（8）www.intel.com：查询 8051（MCS51）的相关介绍。

4 8051 ASSEMBLER 的认识



所谓的固件，是介于硬件与软件之间的沟通桥梁，而 8051 所使用的程序，就是一种固件。然而在编写 8051 的固件程序时，我们必须通过一些应用程序的协助，把我们所编写的程序码转变成 8051 才看得懂的机器码（machine code），这种应用程序，就是我们接下来所要介绍的 Assembler（编译器）。通过本章的介绍，可以让您了解 Assembler 的编码原理，并学会如何将程序转变成机器码，并烧录到 8051 单片机里的完整过程。

第 4 章　8051 ASSEMBLER 的认识

4.1　学习就像旅行

有人说：学习一项新的东西，就好像旅行一样，随时你都会碰到喜怒哀乐各种情形。在旅途中有许多事情是不可预料的，但是只要心理有所准备，所有的困难都会有办法解决的。当一切都平安地回到家后，正好是筹备下次新旅程的开始点。

学习与熟悉 8051 单片机也可以像旅行一样，而本章就好像在机场的出境大厅一样，所有看到和听到的都是新鲜的题材，或许不久就会碰到小难题，不过，这些都会过去的，欢迎跟我们一起登上 8051 汇编语言的登机门，进入一个可以预期很辛苦但很值得的学习历程。

在本章的安排是这样的，你要学习从国外的网站上找到可共享的 8051 编译程序，并将此程序软件的说明书稍微研读一次。接着用 Windows 的记事本 Notepad 编辑软件写一个非常简单的 8051 程序，经过编译后烧录到 20Pin 的 AT89C2051 上，然后直接用面包板做程序操作的确认，把结果直接就显示在 LED 上。整个学习流程就如同是标准 8051 的开发一样，在这里面的技巧有 PC 的操作、8051 汇编语言程序的编写与编译、AT89C2051 的烧录程序、看电路图施工与面包板的线路连接等等，再加上简易 8051 硬件除错能力的培养。

要进行 8051 编译程序的学习前，请先准备妥以下的器材与元件，才能方便加速学习的进度。

（1）PC 个人电脑：可上网下载资料与程序编写，PC 的操作系统建议采用 Windows XP。

（2）PGM2051 烧录器：烧录 AT89C2051 专用。

（3）可输出+5V 的电源供应器。

（4）数字电表。

（5）面包板和连接用的单芯线（十来条）。

（6）AT89C2051：8051 单片机 IC 只有 20 个引脚，最适合简单的 DIY 实验。

（7）石英晶振（11.0592MHz）一枚与 LED 八颗，20pF 电容两个与 1μF 电解电容一个。

（8）激光或喷墨打印机：打印 8051 编译程序说明文件与自己写的程序。

图 4-1　准备工具一：个人电脑 PC 或 NB，还有一台 PGM2051 烧录器

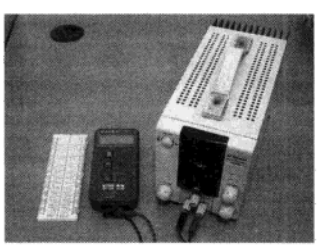


图 4-2　准备工具二：数字电表、电源供应器和面包板

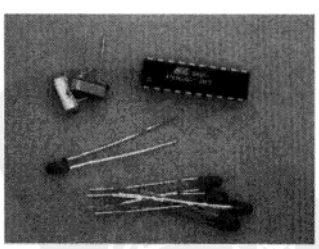


图 4-3　AT89C2051、石英晶振和 LED

4.2　编译器的下载（ASSEMBLER）

学习 8051 单片机的第一步为写汇编语言，汇编语言为最低级但最强的程序语言，我们

所写的程序动作将直接控制 CPU 内所有的工作。基本上，我们所写的汇编语言程序只有指令的代码而已，必须要经过一个特定的翻译程序把指令的代码转译成 8051 单片机看得懂的机器码，这样安排后才能让 8051 单片机开始工作，而这个特定的翻译程序我们称之为 ASSEMBLER，即编译器。

早期的 ASSEMBLER 是要用钱买的，而且价格也不便宜。但现在这类的程序可以在 8051 单片机制造厂的网站上免费取得，而且包括所有的说明文件和程序操作说明。情况会变成这样是因为单片机的市场风起云涌，大家都想扩大市场占有率，如果连 ASSEMBLER 都还收费的话，用户就改用其他厂家的单片机了。

我们这里的 8051 ASSEMBLER 下载自 ATMEL 的网站，本书示范的 AT89C2051 也是 ATMEL 公司的产品之一。以下是从搜索到下载的整个过程：

步骤 1：利用网络连至 ATMEL 的网站 http://www.atmel.com/。

步骤 2：在 ATMEL 的搜索（SEARCH）处，输入“ASSEMBLER”。

步骤 3：在搜索结果中有一个“Technical Documents”的项目，在这个项目下方的会出现“See more results”，点选会将整个 TechnicalDocuments 项目展开。

步骤 4：展开后请找到“MLASM51.EXE Software”。

步骤 5：找到后请按保存直接将文件下载，文件只有 136KB。

图 4-4　在 ATMEL 的网站搜索区（SEARCH）搜索“ASSEMBLER”

步骤 6：我们将文件下载至电脑的 C 盘，为了避免解压缩后找不到文件，所以在下载前先建立一个新文件夹，并且将文件夹命名为“ASSEMBLER”。

步骤 7：建立好 ASSEMBLER 文件夹后，就直接进入新建的 ASSEMBLER 文件夹并且按下保存按钮来保存。

文件下载保存后我们就开始来安装。

步骤 8：下载保存后，请打开 C 盘内的 ASSEMBLER 文件夹，打开后会发现只有 MLASM51.EXE 一个文件，直接用将这个可执行文件打开。

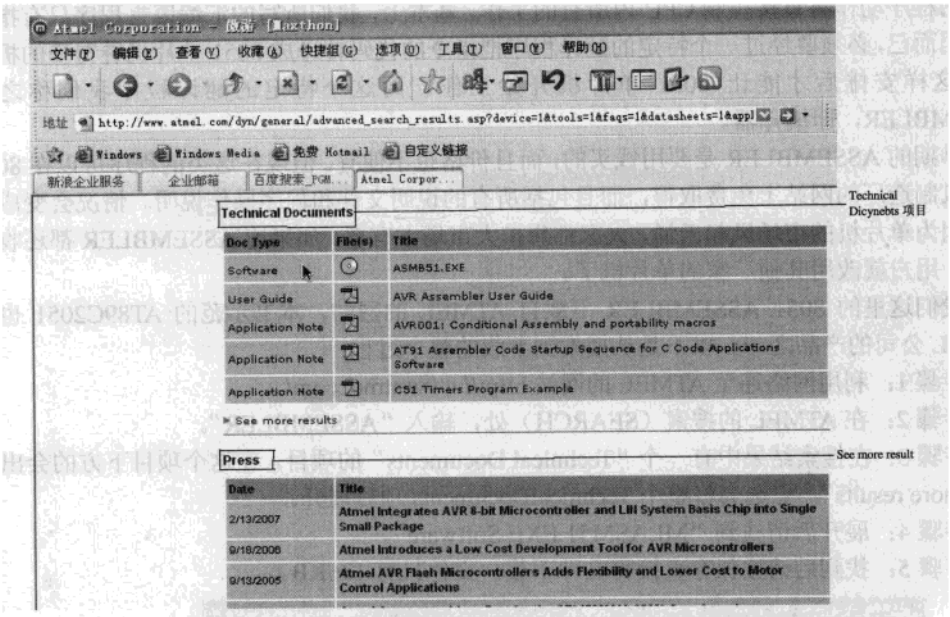


图 4-5　利用 ATMEL 搜索 ASSEMBLER 的结果

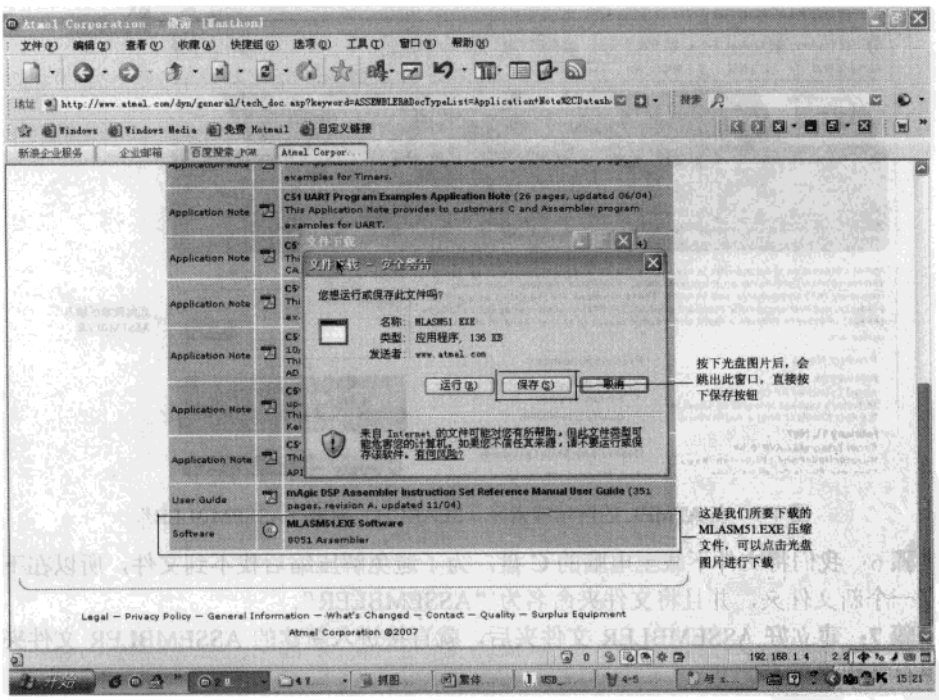


图 4-6　搜索到 MLASM51 后直接点击下载

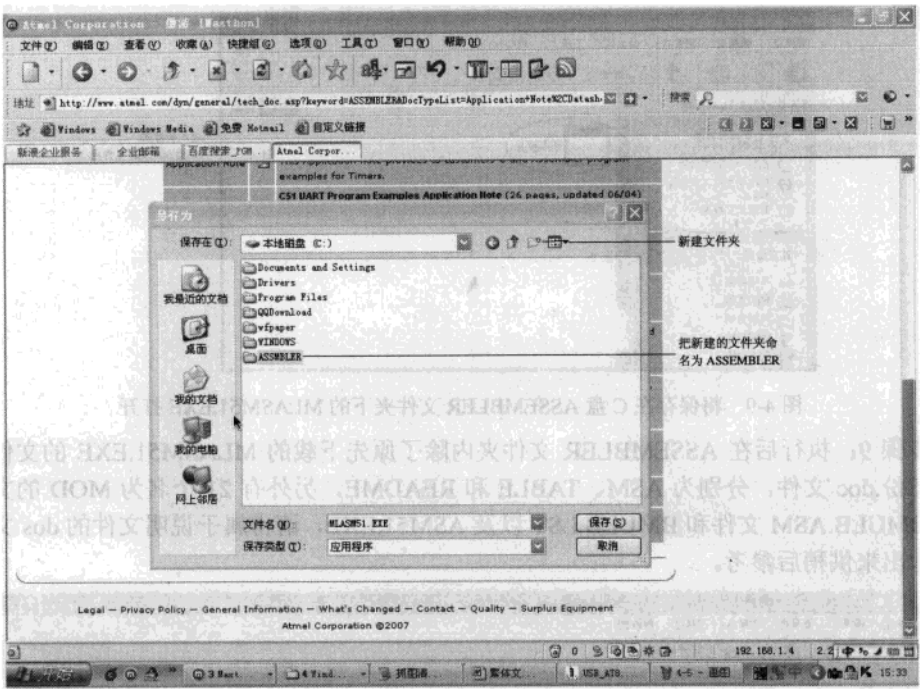


图 4-7　将下载的文件保存在 C 盘下，并且新建一个“ASSEMBLER”文件夹

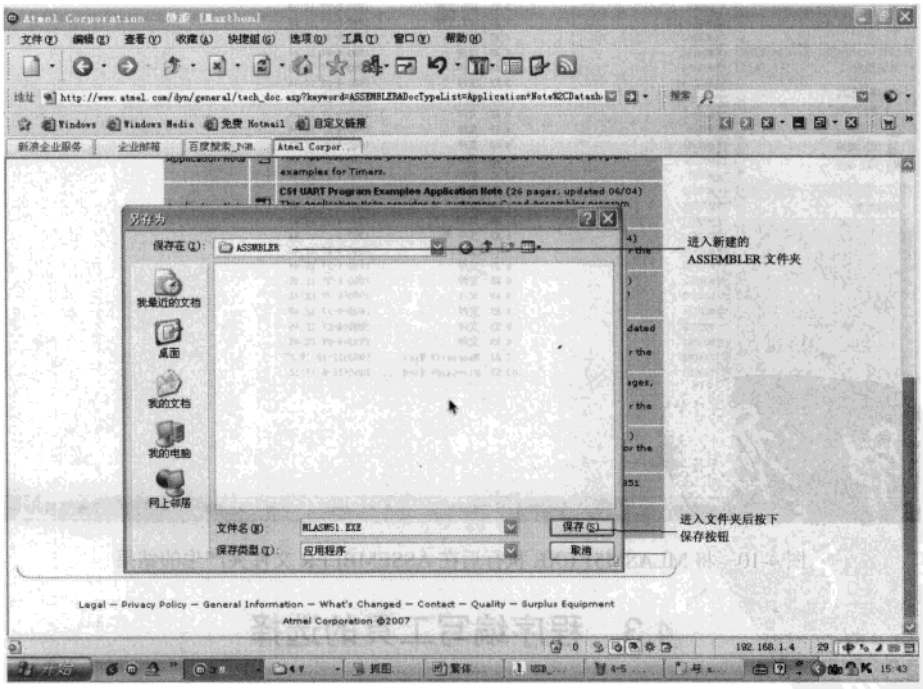


图 4-8　进入在 C 盘下新建的 ASSEMBLER 文件夹，并且将 MLASM51.EXE 文件保存

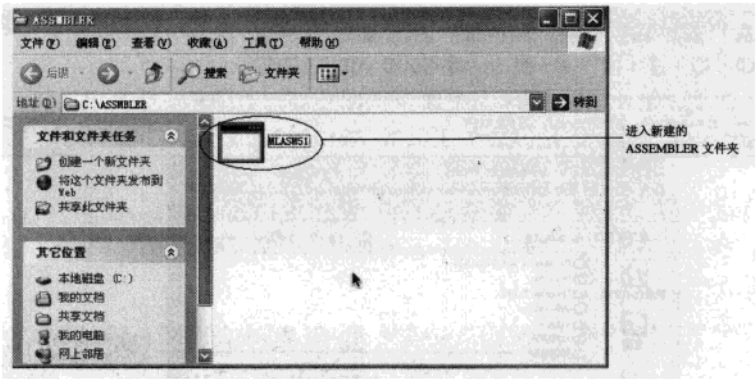


图 4-9　将保存在 C 盘 ASSEMBLER 文件夹下的 MLASM51.EXE 打开

步骤 9：执行后在 ASSEMBLER 文件夹内除了原先下载的 MLASM51.EXE 的文件外，还有三份.doc 文件，分别为 ASM、TABLE 和 README，另外有 21 个名为 MOD 的文件，还有 BMULB.ASM 文件和 BMULB.LST 以及 ASM51.EXE，请将属于说明文件的 dos 文件全部打印出来供稍后参考。

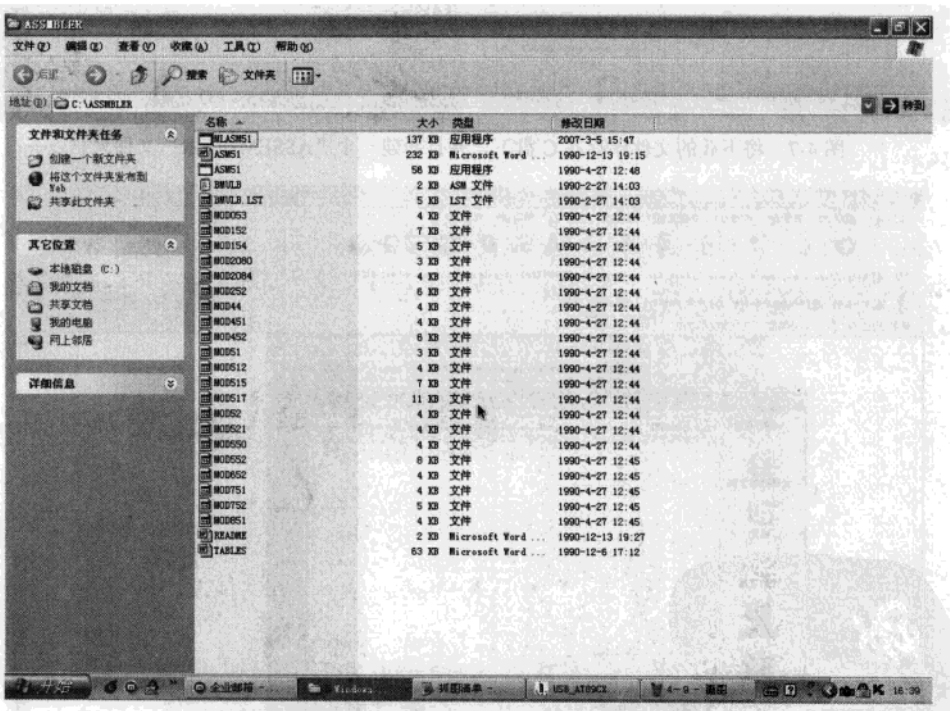


图 4-10　将 MLASM51.EXE 执行后在 ASSEMBLER 文件夹产生的结果

4.3　程序编写工具的选择

编写 8051 程序的工具不难找到，只需要用电脑内 Windows 提供的记事本来编辑。记事

本 Notepad 打开方式是从电脑的“开始”→“所有程序”→“附件”，就可以点击并在桌面上打开一个新的记事本，打开后直接就可以编写 8051 的程序代码。另外，还有一个方法也可以打开记事本，就是直接在 Windows 的桌面按鼠标右键→“新建”→文本文档，点选后也一样可以打开 Notepad 记事本。其他类似的工具像 WordPad（写字板）也可以拿来做编写 8051 程序的工具。

写 8051 程序更专业的工具软件还有许多，如 Ultra-Editor 软件都是许多程序设计师的最爱，但是这是要花钱买的，价格约在几十元美金左右，稍后你也可以下载 Ultra-Editor 的试用版本来试用看看。

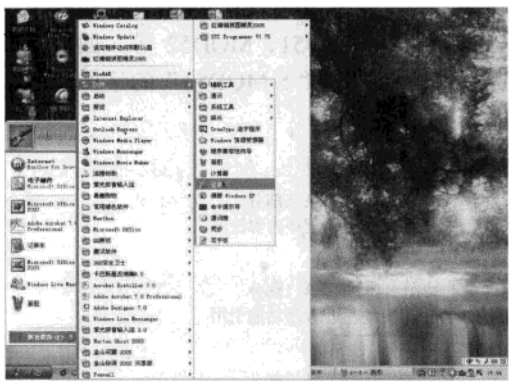


图 4-11　从电脑的开始→所有程序→附件，可以找到记事本工具来编辑程序码

图 4-12　由桌面按下鼠标右键，也可以新建文本文档

在程序编辑完后时有个应该要注意的问题，就是文件的保存类型，原先由 WordPad 或记事本保存的扩展名为.txt，这边需要自行更改成.asm 的文件名。更改的操作不困难，只要选取需更改的文件后，按下鼠标右键，就会出现重命名的选项，直接用键盘键入文件名和扩展名即可完成更改的操作。

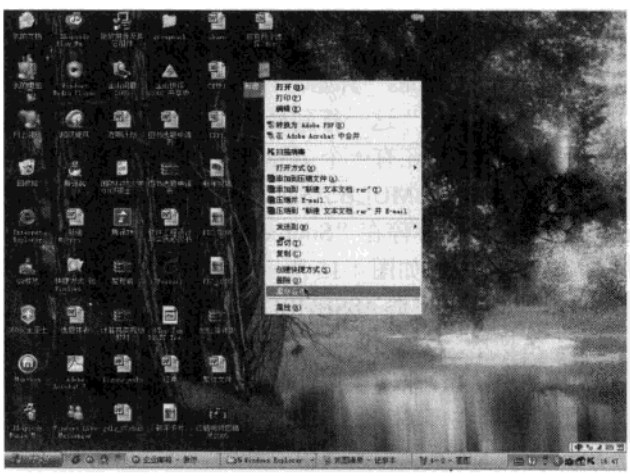


图 4-13　更改记事本默认的文件名，先选取文件，按下鼠标右键，选择“重命名”

4.4 MLASM51.EXE 的学习

MLASM51.EXE 解压缩后第一个操作是先打开所有的文件来浏览一下，除了.EXE 文件是可执行文件无法直接打开外，其他文件也都可以用记事本来打开。我们先看到 README.DOC 文件，内容大约是告诉使用者，使用的手册分为 ASM51.DOC 和 TABLE.DOC 两部分，以及两份文件相对应的页数，这两份文件应该就是学习 MLASM 编译器的说明书了，建议您全部打印出来，方便以后查询使用。

而 BMULB.ASM 文件和 BMULB.LST 文件，ASM 文件是 8051 的汇编语言范例程序，接下来的 LST 文件应该就是编译后的相关资料。其他还有 MOD51、MOD52 等，如果打开 BMULB.ASM 来看，会发现在一串的程序注释，在第 17 行出现“$MOD51”的字样，所以这些 MOD 的文件应该有特殊用途。

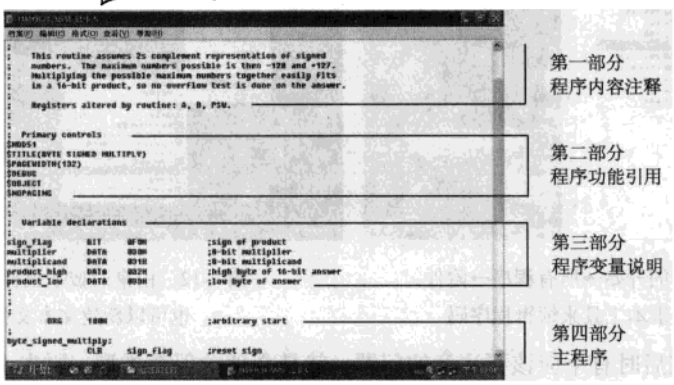


图 4-14 打开下载后的文件 BMUBLB.ASM

我们继续看 BMULB.ASM 这份范例程序，可以将这个程序大约分成四个阶段：

（1）最上方是程序功能的注释，也就是分号后面的内容。

（2）从“Primary controls”开始，所有程序需要的功能都会在这里声明。

（3）这部分从“Variable declarations”开始，程序内用到的变量要在这部分列出。

（4）从“ORG 100H;arbitrary start”这一行到最后一行的“END”，就是程序主要的部分。

了解程序的编写流程后，接下来还有一个 ASM51.EXE 的执行文件，这个一定就是用来转文件用的编译器了，我们直接使用 BMULB.ASM 这个程序来测试一下。在打开 ASM51.EXE 后会出现 DOS 模式的一个窗口，光标停在“Source file drive and name[.ASM]：__”这一行，直接将程序的文件名 BMULB 输入，如图 4-15 所示，输入后按下“Enter”键。

执行完后 DOS 窗口会自动关闭，可是 ASSEMBLER 的文件夹同时多出 BMULB.HEX 文件和 BMULB.DBG 文件，如图 4-16 所示，以后如果需要编译 8051 程序就可以直接使用 ASM51.EXE 了。

除了直接点击 ASM51.EXE 文件来转换文件之外，我们也可以从“命令提示符”打开编译程序。下面来看看两者的差别在哪里？

步骤 1：我们以 Windows XP 系统为例，由“开始”→“所有程序”→“附件”→“命令提示符”，找到后并将它打开。

步骤 2：打开后的位置都不太一样，这时我们要先找到 ASM51.EXE 文件在计算机内的

位置，下载后我们是放在“C:\ASSEMBLER”文件夹下，如图 4-17 所示。

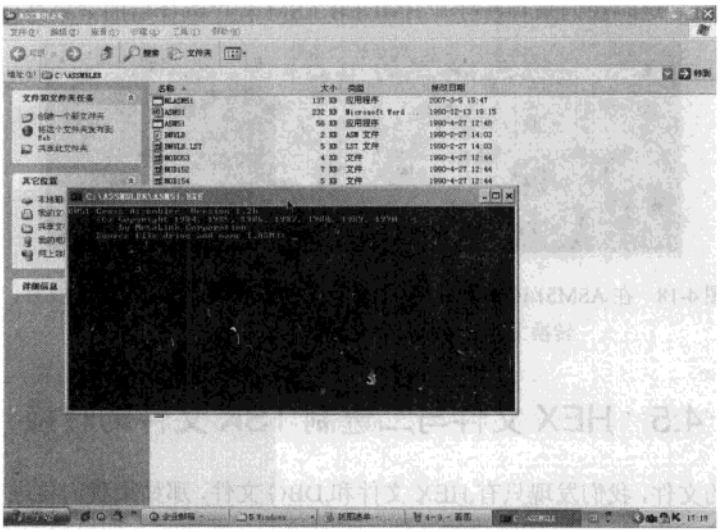


图 4-15　ASM51.EXE 打开后的样子，直接输入 BMULB

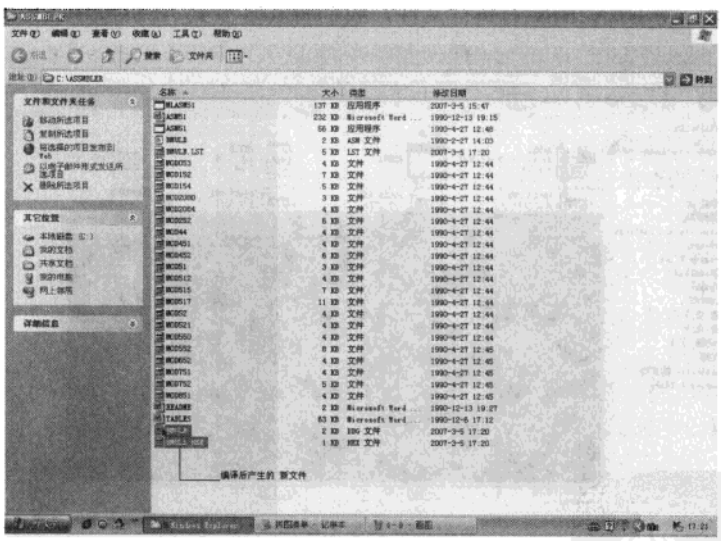


图 4-16　在 ASSEMBLER 文件夹多出 BMULB.HEX 和 BMULB.DBG 两个文件

图 4-17　先进入到 C 盘下后，我们再进入 ASSEMBLER 文件夹，最后再打开 ASM51.EXE

步骤 3：在进入存放 ASM51.EXE 的 ASSEMBLER 文件夹后，就可以直接从“命令提示符”打开 ASM51，这时我们来看看打开 BMULB.ASM 程序编译后的结果是什么？

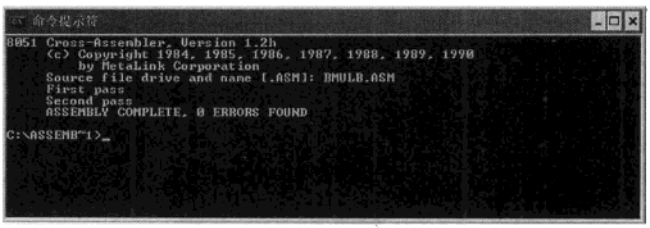


图 4-18　在 ASM51.EXE 运行时，我们同样打开 BMULB.ASM 转换文件，转换文件后屏幕上会显示程序是否编译成功

4.5　HEX 文件与二进制 TSK 文件的转换

编译出来的文件，我们发现只有.HEX 文件和.DBG 文件，那如果我们要进行烧录的操作，似乎还少了一个.TSK 或.BIN 的二进制文件。这时该怎么解决呢？如果您的烧录器真的只支持二进制的烧录文件，那接下来的工具程序，很快就可以解决您的困扰了。请打开随书附的光盘，里面收录一份 HEX2BIN 的执行文件，我们将其复制到 ASSEMBLER 的文件夹内。

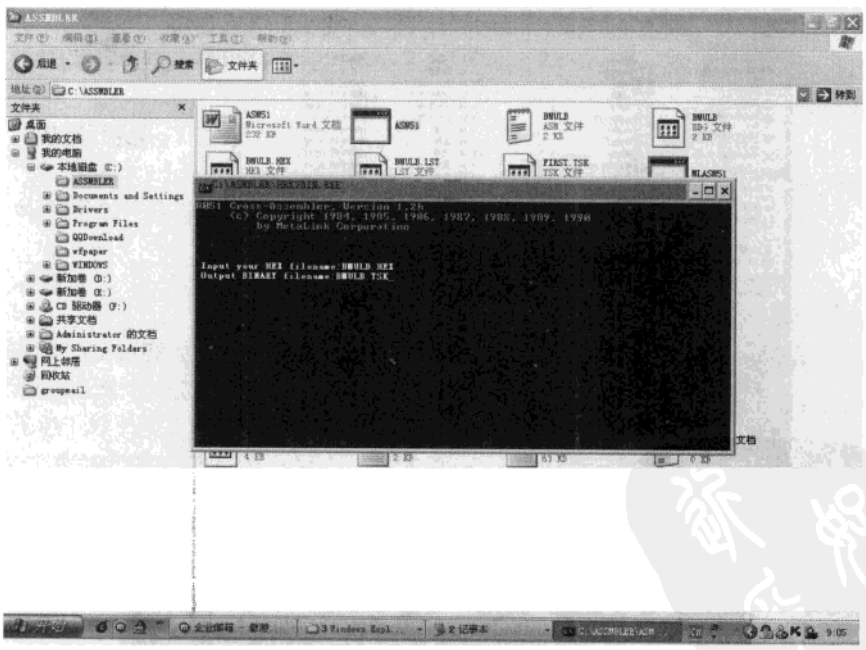


图 4-19　HEX 文件转二进制文件（HEX2BIN）的执行后所看到的画面

我们将 HEX2BIN 执行文件打开后会看到上面的 DOS 画面，这时光标会停“Input your HEX filename:”，在这里直接键入要转换的 HEX 文件的全文件名，我们继续用 BMULB.HEX 文件来测试，输入后按下 Enter 键，这时又会出现第二行“Output BINARY filename:”，这行

要键入转成二进制文件的全文件名，这边我们一样命名为 BMULB.TSK 或是 BMULB.BIN 都可以。输入之后一样按下 Enter 键，再回到 ASSEMBLER 文件夹底下，我们会发现又多了一个 BMULB.TSK 文件。

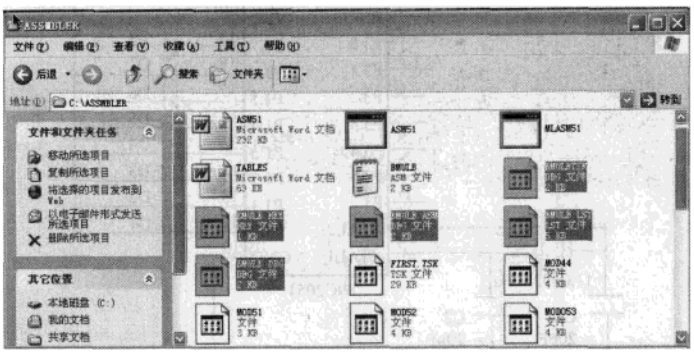


图 4-20　从下载到编译再到转换二进制文件，ASSEMBLER 文件夹里一共有 5 个 BMULB 的相关文件

现在 ASSEMBLER 的文件夹一共出现了 5 个 BLUMB 相关的文件，除了.ASM 文件是下载时就存在的，其他 LST 文件、HEX 文件和 DBG 文件都是编译后所产生的，TSK 文件则是我们再次转换后所产生的，现在就简单整理一下这些文件：

（1）.ASM 写 8051 程序时文件的存储格式。

（2）.LST 编译后会产生的文件，该文件会显示许多有用的信息。

（3）.HEX 由 Intel 公司制定的标准格式，它会在编译后产生，主要功能是程序码以 ASCII 码的方式显示。

（4）.DBG 除错文件，当程序发生错误可以利用该文件配合模拟器进行除错。这个文件的详细用法请直接参考原手册上的说明。

（5）.TSK 二进制文件，和.BIN 文件一样，通常是转换给烧录器专用的程序码。

4.6　第一个 8051 程序

现在我们要踏出写程序的第一步了，准备写一个小程序，经过 ASSEMBLER 编译、转成二进制文件 HEX2BIN，烧录到 AT89C2051 并且放到面包板上来完成一个软件和硬件的结合。现在就打开记事本来编辑第一个 8051 程序，并且将记事本的文件名更改成 FIRSTASM，这时要注意，第二行开始前面的空白和 MOV 后面的空白处是按键盘上的 Tab 键，也可以用空格键 Space 来做。

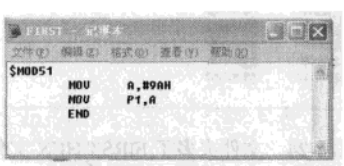


图 4-21　用“记事本”编辑出 FIRST.ASM 的程序

这段程序是参考 BLUMB.ASM 原程序后，修改并简化而成的，它要让八个 LED 灯显示某一个特定的状态。在线路上我们采用负逻辑接法，所以数字状态 0 代表亮灯，1 代表 LED 不亮。

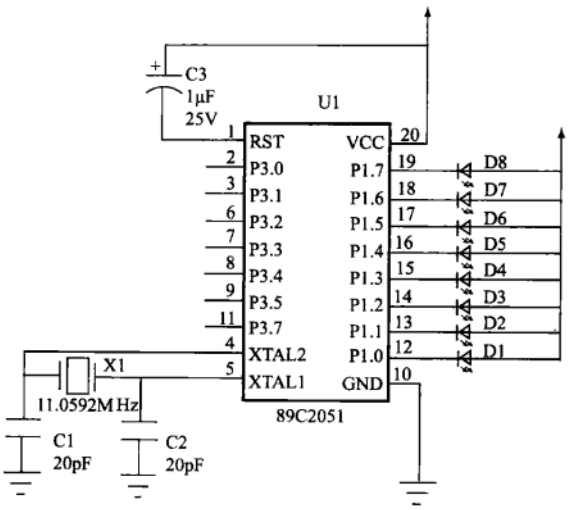


图 4-22　配合 FIRST.ASM 的电路图

程序编写好后，运行 ASM51.EXE 并输入 FIRST 文件名称，产生 FIRST.HEX 文件。

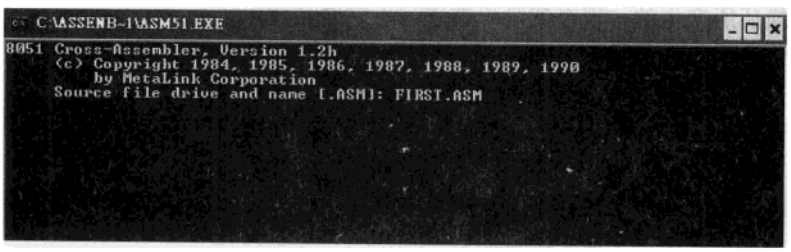


图 4-23　ASM51 的运行画面

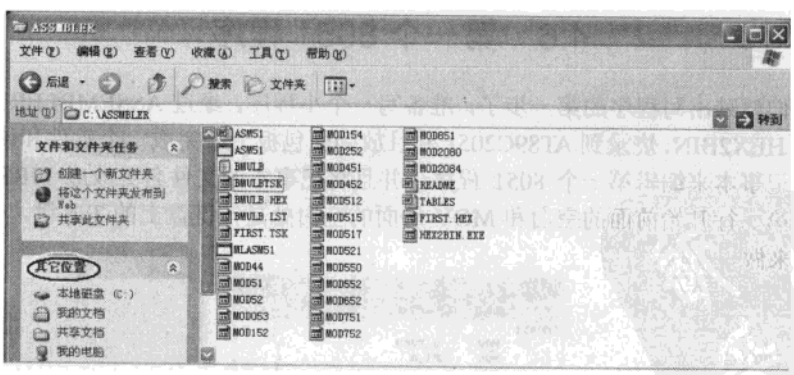


图 4-24　文件夹多了 FIRST.HEX 文件

接下来运行 HEX2BIN.EXE 将 FIRSTREX 文件转成 FIRST.TSK 二进制文件，此文件的内容将直接烧录到 AT89C2051 当中。所以此时要接上有 USB 接口的 PGM2051 烧录器，再运行 PGM2051 烧录程序。

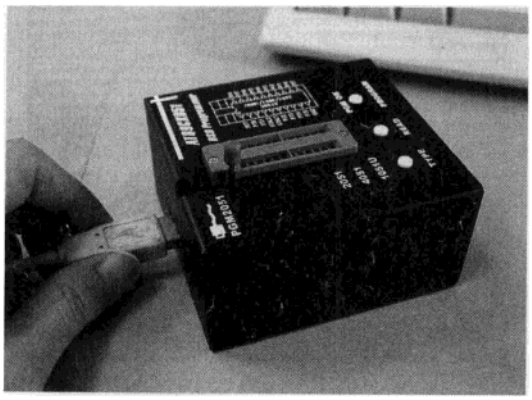


图 4-25　用 USB 线连接 PGM2051

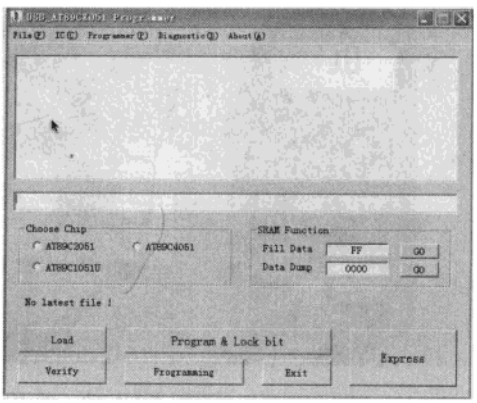


图 4-26　单击 PGM2051，进入烧录程序

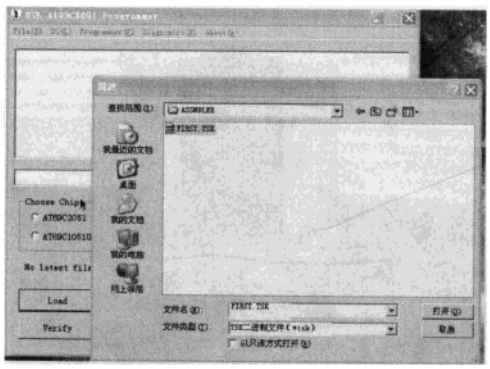


图 4-27　选择 FIRST.TSK 文件，将数据传至烧录器上

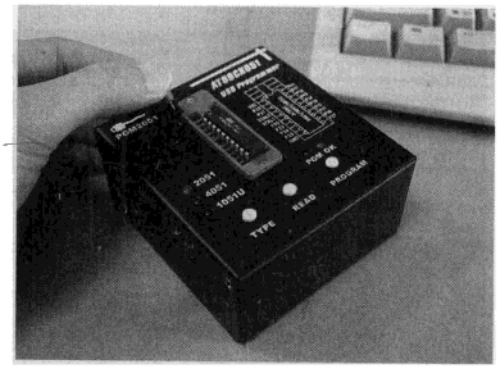


图 4-28　将 AT89C2051 放到烧录夹上，运行 PROGRAM 进行烧录

接下来是 DIY 硬件的组装，请参考图 4-22 用面包板完成所有的硬件接线，通+5V 电源前一定要先用数字电表检查电源端没有短路。

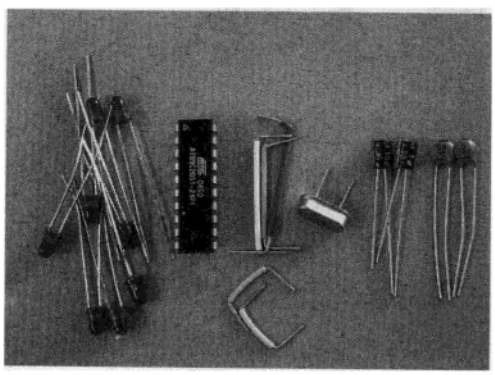


图 4-29　检查所有的电子元件是否到齐

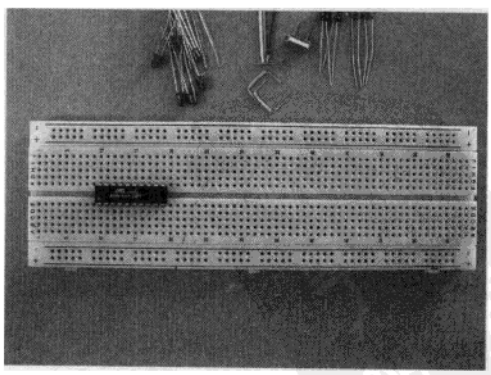


图 4-30　AT89C2051 先放到面包板的中央位置，其他元件再一一加上

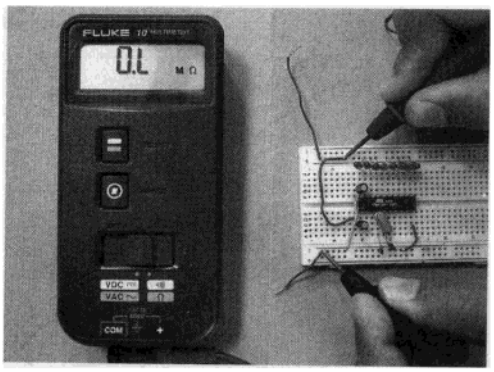


图 4-31　用欧姆挡确定电源端没有短路。图中电表的 O.L MΩ 代表真的没有短路

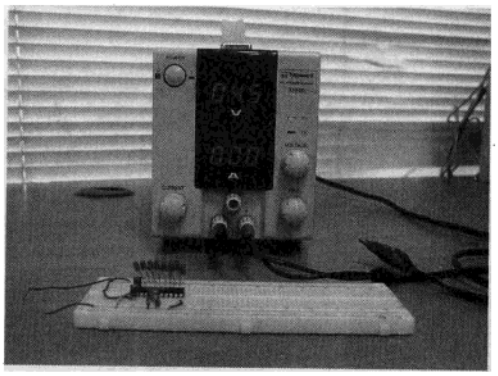


图 4-32　已经完成的电路板尚未通电，电源供应器已经在旁边待命

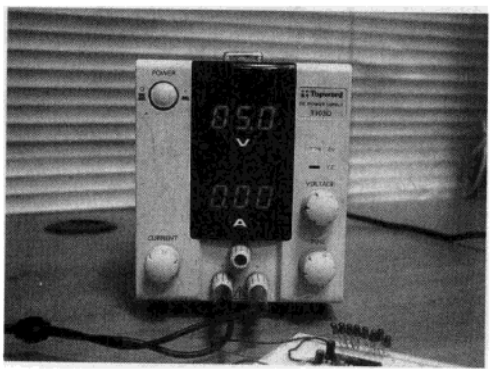


图 4-33　电源供应器的电压调整到+5V 输出

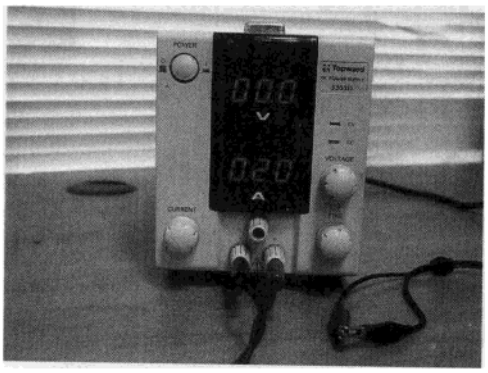


图 4-34　电源供应器的电流要限制在 0.5A 以防止面包板上的不正常短路

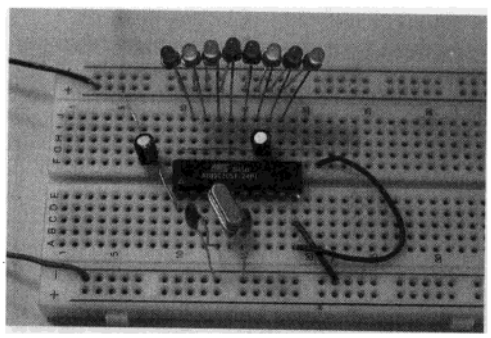


图 4-35　图中 P1.0 为最右边的 LED 灯，往左分别为 P1.1 到 P1.7 端口，通电后其中的 P1.0、P1.2、P1.5、P1.6 引脚的 LED 会亮。重新打开电源几次，结果都一样

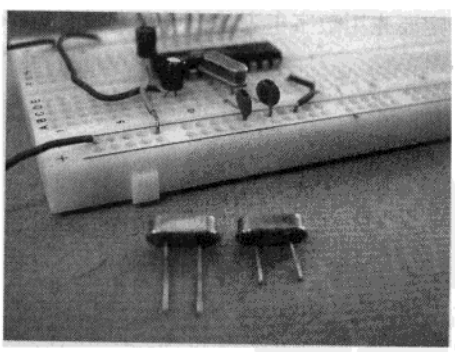


图 4-36　可以更好的配线范例一：XTAL 石英晶振的引脚尽量短

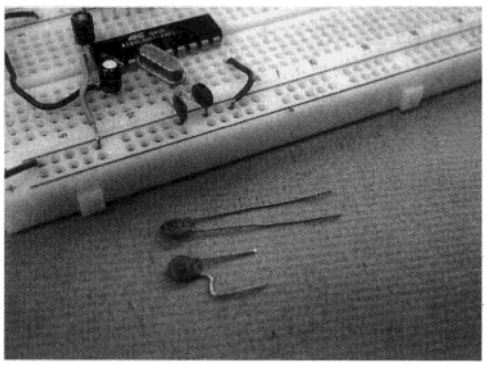


图 4-37　可以更好的配线范例二：20pF 电容的引脚不可过长

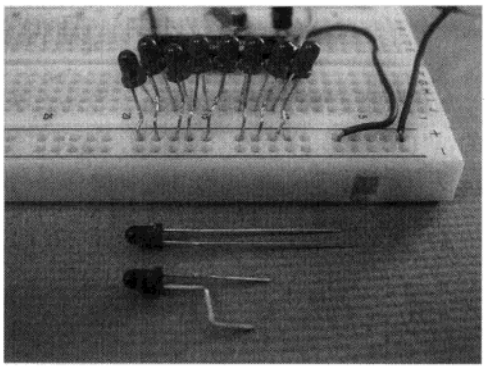


图 4-38　可以更好的配线范例三：LED 的引脚经过调整后，比较整齐也不容易短路

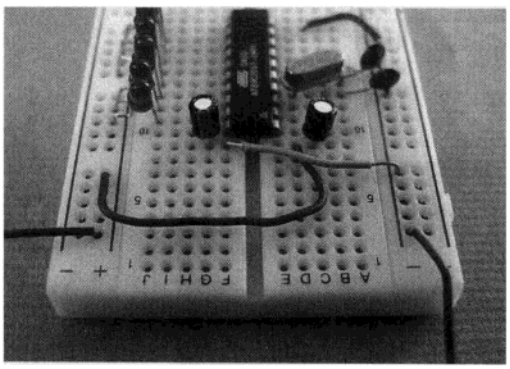


图 4-39　可以更好的配线范例四：重置 RESET 用的电容再做调整

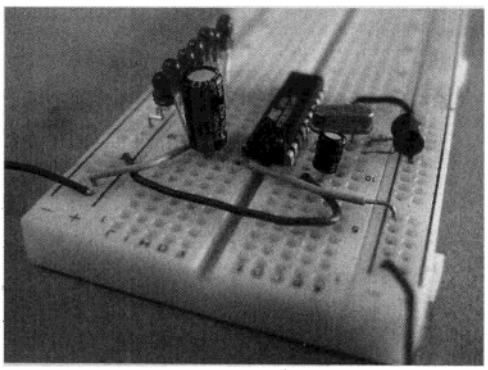


图 4-40　可以更好的配线范例五：在电源输入端加入一额外的 100μF/25V 电解电容

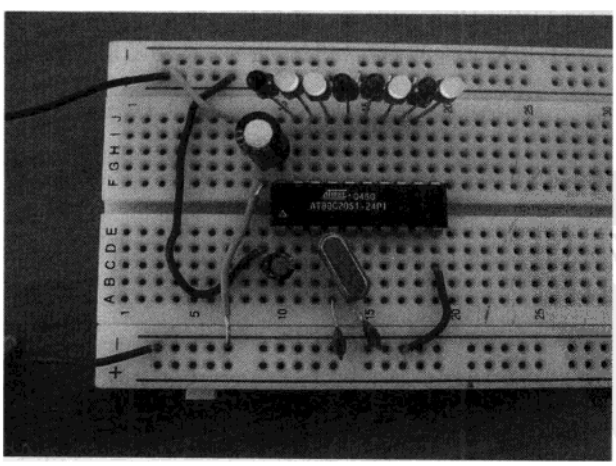


图 4-41　完成后通上+5V 电源

4.7 硬件失败的检查点

如果您通电后 LED 灯不亮，自我检查一次还是不亮！程序完全照抄还出错？这是很有可能的，“DIY 累积的经验不够的话，这种可能性会一直存在的”。

以下是我们归纳出的几个重要检查点，请逐条进行确认。

Check Point1　您写的程序对吗？

请仔细检查标点符号和英文字母，程序最后是 END。

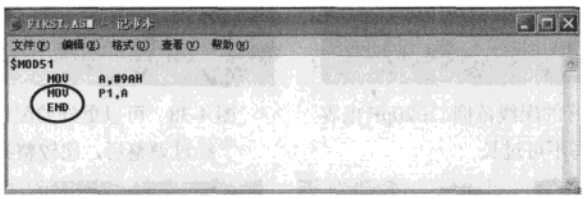


图 4-42　程序文件图

Check Point2　ASM51 ASSEMBLE 编译过关吗？

ASSEMBLER 过关的标准画面如图 4-43 所示。

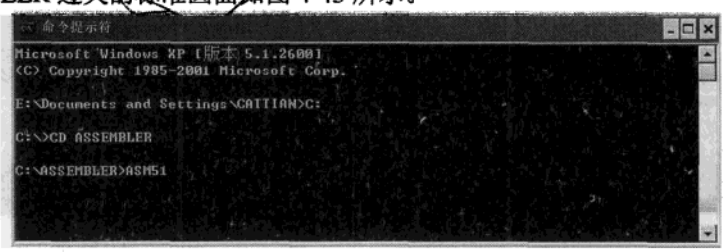


图 4-43　ASSEMBLER　过关画面

Check Point3　检查 FIRST.HEX 的内容，完全相同吗？

用记事本查看 FIRSTHEX 文件的内容要完全相同，请勿随意更改其中的内容，如图 4-44 所示。

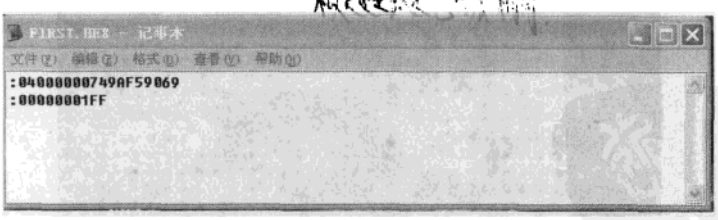


图 4-44　FISTHEX 文件内容

Check Point4　HEX2BIN 转换文件执行后，FIRST.TSK 文件大小正确吗？用文件“属性”查看 FIRST.TSK 的文件大小，如图 4-45 所示。

Check Point5　PGM2051 烧录器连接正确吗？

执行时应该不会出现以下连接失败的画面，如图 4-46 所示。

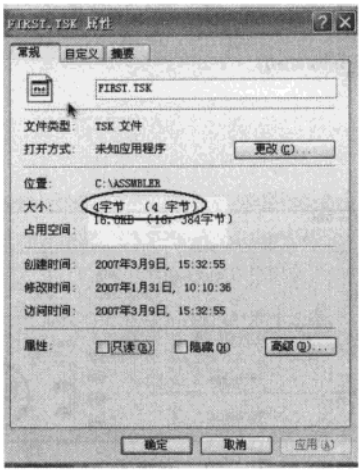


图 4-45　文件“属性”

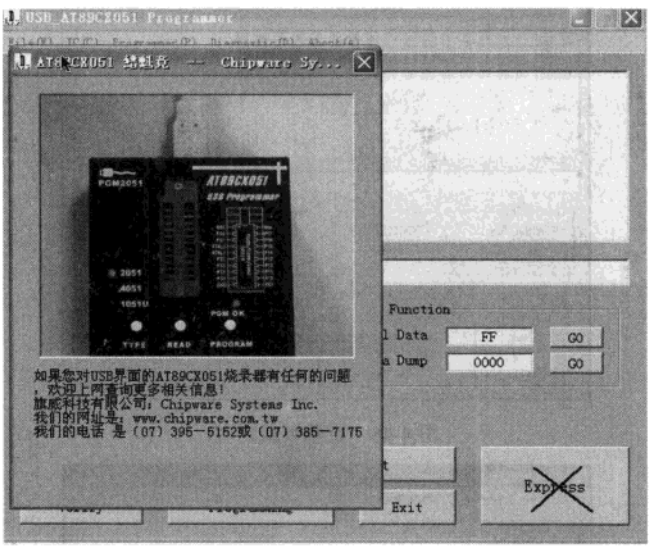


图 4-46　连接失败

正常连接成功后的画面如图 4-47 所示。

Check Point6　下载到 PGM2051 的数据对吗？

用 PGM2051 上的 Data Dump 查看下载的内容，从 0000H 开始至少要有 4 个字节是正确的，如图 4-48 所示。

Check Point7　PGM2051 烧录 AT89C2051 是否成功？

是否选对 IC 的编号？烧录成功的画面，如图 4-49 所示。

Check Point8　线路与手工所接的电路完成相同吗？

AT89C2051 的引脚顺序对吗？电解电容的极性是否对？LED 的较长引脚接+5V 端？如图 4-50 所示。

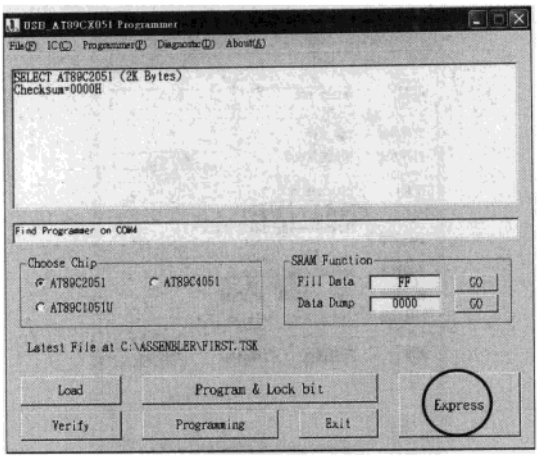


图 4-47　连接成功

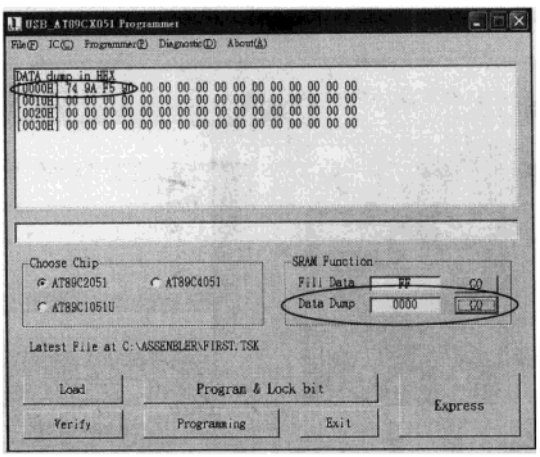


图 4-48　查看下载内容

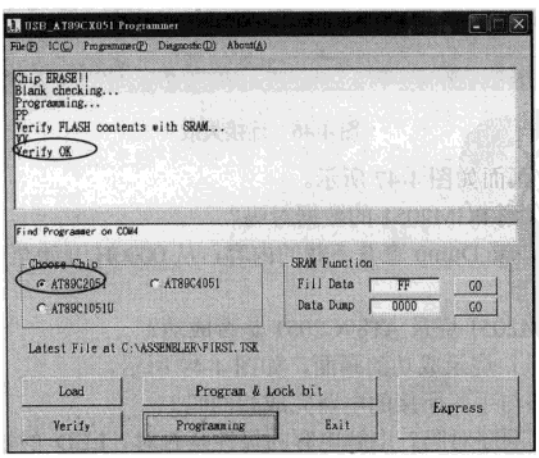


图 4-49　烧录成功画面

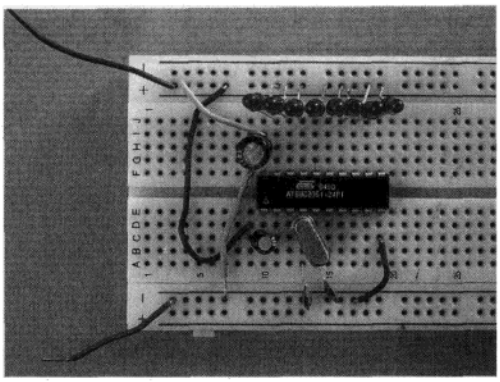


图 4-50　电源连接图

Check Point9　电源供应正确吗？

请用数字电表检查电源供应器的输出、面包板的电源输入端和 AT89C2051 的第 10 引脚与第 20 引脚间的电压值应该是+5V，误差应该要在 0.1V 以内，如图 4-51 所示。

Check Point10　石英晶体接法正确吗？

石英晶体没有极性，可以的话请用示波器看 XTAL2 引脚的波形，波形可能不是漂亮的方波，但是频率应该在 11MHz 上下，如图 4-52 所示。

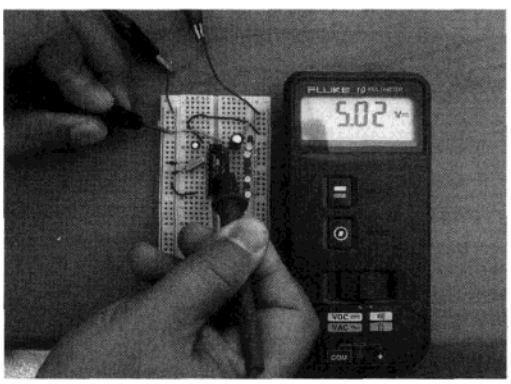


图 4-51　检查电源供应图

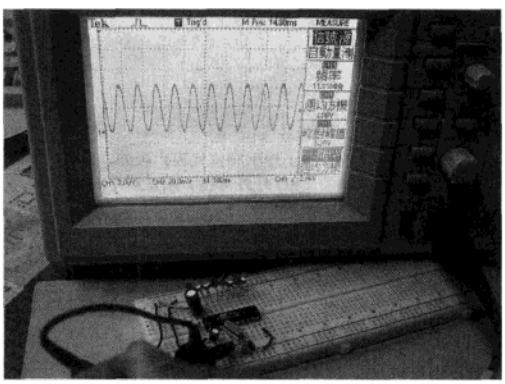


图 4-52　石英晶振连接及波形图

Check Point11　RESET 脚应该接近+0V。

用数字电表对 RESET 引脚进行电压测量，红色测棒接 RESET 引脚（Pin 1），黑色测棒接到面包板的 0V 地端，如图 4-53 所示。

Check Point12　LED 的极性接对了吗？

LED 的正负端再进行确认，其实 LED 的塑胶外壳就有极性标示。可以用数字电表的二极管挡做 LED 的点亮试验，如图 4-54 所示。

Check Point13　以上的检查点都对了吗？

如果都对的话，您 DIY 的电路板应该能工作才对。请关电后再次重新送电给面包板，LED 还是那几个亮，亮的继续亮，不亮的还是不亮“这意味着 AT89C2051 内部在电源打开后，会执行一个特定的程序让 LED 点亮，而这个程序就是我们上一节所写的 FIRSTASM 程序。

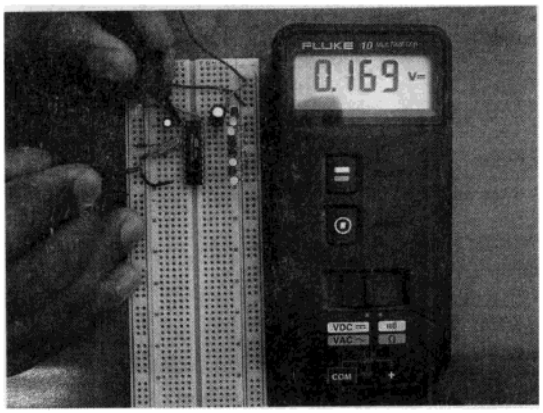
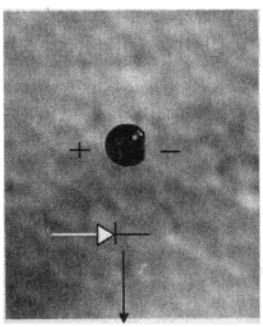
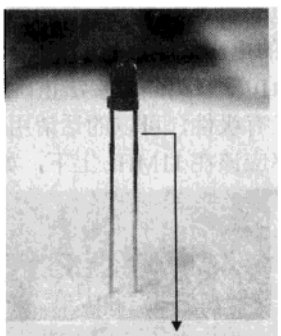


图 4-53　RESET 引脚测量图



(a) 除了短引脚位代表负极外，仔细看 LED 灯外可在负端是扁平的



(b) 左边是正极，右边是负极，可以看到负极部分比较扁平

图 4-54　LED 的接法

4.8　ASM51 的进一步认识 1

FIRSTASM 程序是我们所写的第一个汇编语言程序，程序只有四行经过 ASM51 编译后，产生了 FIRST.HEX 文件与 FIRST.LST 文件，后者是一个打印文件，上面有编译过的信息存档。

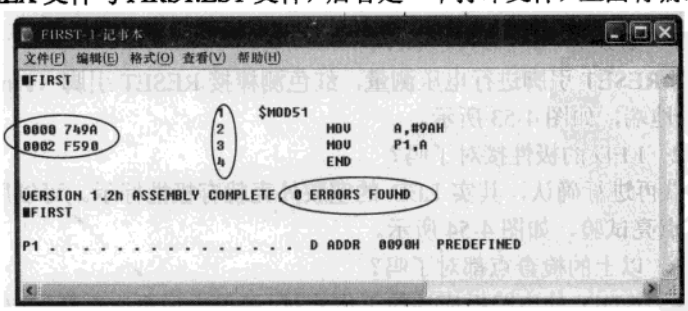


图 4-55　FIRST.LST 的全部内容

这个 FIRST.LST 文件内其实有许多信息的，当编译 FIRSTASM 没有任何错误时，就会在

LST 文件上留下“0 ERROR FOUND”的字样，告诉使用者这些写法是对的。

请看图 4-55 中央有一排数字，由 1 排到 4，代表原始程序（Source Program）共有四行。第 1 行与第 4 行左边都没有其他文字，代表这两行是程序的某种叙述，但不产生 8051 单片机的机器码。第 2 行与第 3 行的左边就有意思了，最左边的是程序的计数值（Program Counter），接下来是 4 个字节的十六进制值 749AH 与 F590H。这些码就是 ASM51 编译程序针对 FIRST.ASM 的内容所产生的最重要的结果。你对这几个数字觉得似曾相识吗？其实这些在前几节中已经有出现了，请再看看图 4-56 中 FIRST.HEX 的内容。

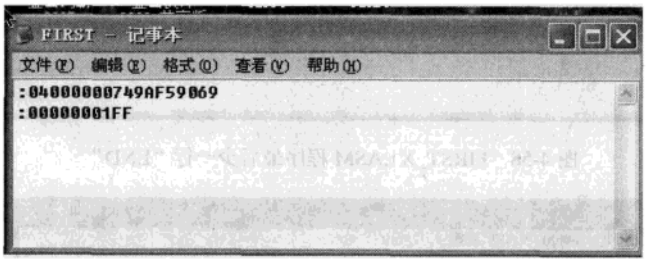


图 4-56　FIRST.HEX 内部也有 74 9A F5 90 这几个十六进制值

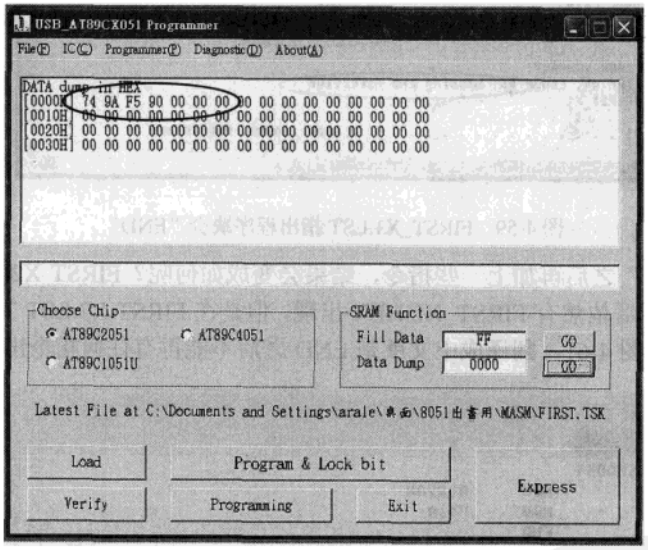


图 4-57　PGM2051 烧录程序下载完 FIRST.TSK 文件后，检查待烧录到 AT89C2051 的内容也有 74 9A F5 90 这些数据，刚好是占 4 字节，FIRST.TSK 文件的长度也是 4

经以上的说明，你是否对 ASM51 有进一步的了解？FIRST.ASM 经过 ASM51 编译成功后，最终目的就是产生 74 9A F5 90 这些十六进制码，只要这些码能够顺利地烧录到 AT89C2051 内，LED 就能够工作了。

4.9　ASM51 的进一步认识 2

FIRSTASM 程序最后面是一行“END”，这代表程序已经结束了。这一行程序一定需要吗？

我们试着把这行取消掉，然后用 ASM51 再编译一次，看看结果如何？经过修改后的程序名称为 FIRST_X1.ASM，请看图 4-58。结果是有一个错误信息出现，在 FIRST_X1.LST 文件中出现“MISSING END DIRECTIVE”等字样，编译器认为你写的程序尚未结束，这样是不对的。

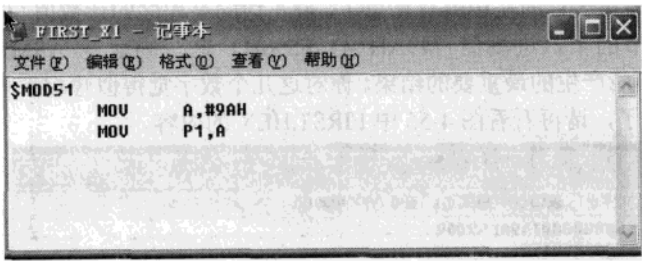


图 4-58 FIRST_X1.ASM 程序最后少一行“END”

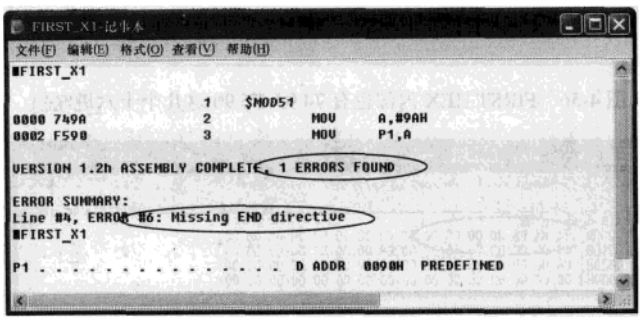


图 4-59 FIRST_X1.LST 指出程序缺少“END”

如果在“END”之后再加上一些指令，结果会变成如何呢？FIRST X2.ASM 的写法就是这样。ASM51 编译后依然有 FIRST_X2.HEX 出现，但是在 FIRST X2.LST 文件中却有一个错误信息出现，请看图 4-61，翻译成中文就是 END 之后不能再有任何指令出现。

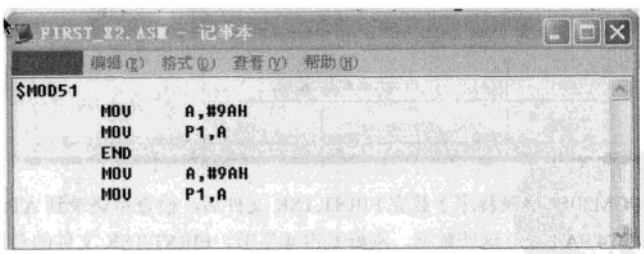


图 4-60 FIRST_X2.ASM 程序在“END”之后又多了两行

ASM51 编译程序是一套非常严谨的 8051 编译程序，除了程序最后一定要加上 END 字眼以外，它还有一大堆的规定和限制。就好像要开车上路前，要先有驾驶执照才行，这些规定不是故意找你麻烦，而是通过这些限制让你写的程序不容易出轨，因为你写的程序是汇编语言程序，直接就掌握整个单片机的操作，稍一不慎就会使整个系统崩溃，所以小心一点绝对是对的。

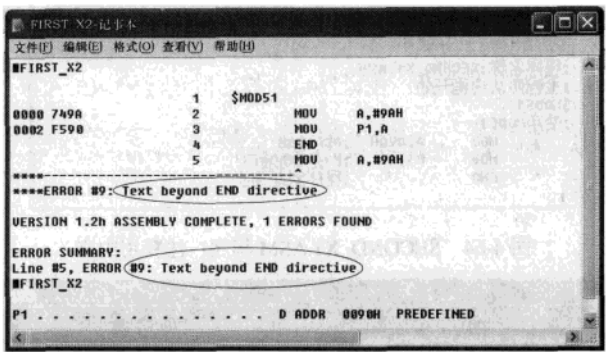


图 4-61 FIRST_X2.LST 指出程序缺少“END”

4.10 ASM51 的进一步认识 3

在写 8051 汇编语言程序时，我们常常会在程序当中加上一些注释，说明程序的用法或是重点说明，注释的英文为 COMMENTS，这些注释可以加在程序的任何地方，但是同时要在该注释前加上一个特别记号“;”（分号）即可。我们第二个程序 SECOND.ASM 就是在示范注释的写法。经 ASM51 编译后没有错误产生，也顺利地产生 SECOND.HEX 文件。

注释通常是用英文比较方便，中文的注释可行吗？本节的第二个示范程序 SECOND_X1.ASM 内的注释是中英文混用的，编译后却没有产生.HEX 文件及.LST 文件。这是因为文件名长度超过了八个字符。早期的操作系统所支持的文件名长度只有八个字符，所以把文件名改成 SECOND_1.ASM 之后就可顺利完成编译了，因此我们也可以利用中文为程序加上注释。

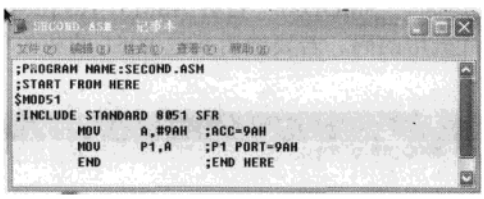


图 4-62 SECOND.ASM 程序的内容

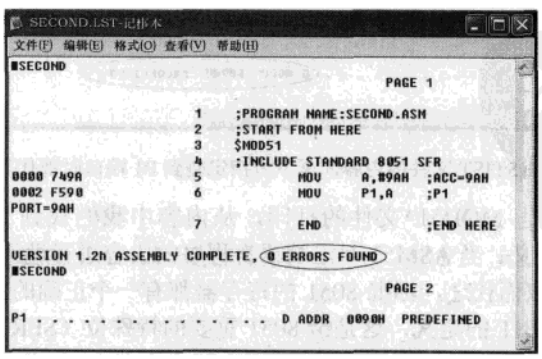


图 4-63 SECOND.LST 说明没有任何错误出现

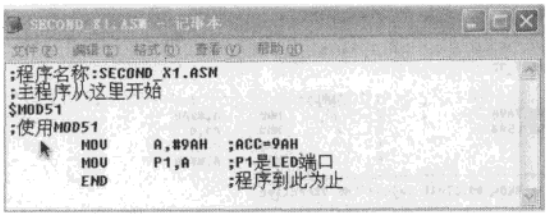


图 4-64　SECOND_X1.ASM 程序：注释用中文

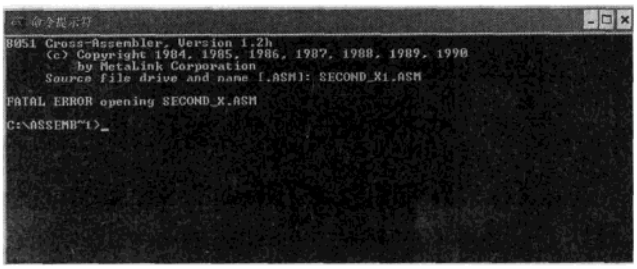


图 4-65　SECOND_X1.ASM 编译时，报告的错误为“FATAL ERROR opening SECOND_X.ASM”

4.11　ASM51 的进一步认识 4

在 ASM51 所产生的 LST 文件中还有最后一小段信息我们尚未说明，这段叙述在指明特殊功能寄存器的确实地址值。因为 8051 单片机衍生的族群众多，本身还是有点差异的，但是 ASM51 还是按型号要产生正确的对应值来。以前我们写的程序最前面都要加上 $MOD51 这一行就是这个原因。“$MOD51”告诉 ASM51 ASSEMBLER 编译程序：我要产生的码是标准的 8051 单片机用的码。若要选用其他 8051 的相关单片机时，就要选用其他的 MOD 文件。

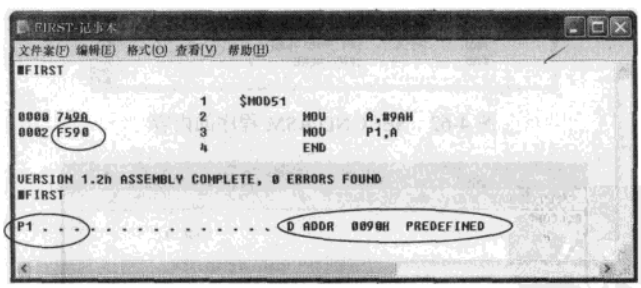


图 4-66　FIRSTIST 文件的内容，最下方指定碰到 P1 端口时都代入 90H 的值

请看图 4-67，这是 MOD51 文件的打印，从内容中我们找到“Pl.....DADDR 0090H PREDEIFNED”这段定义。当 ASM51 对汇编语言程序的内容进行编译时，只要一碰到 P1 的字样时，就会以 90H 取而代之，标准 8051 的寄存器都有一个正确的对应的十六进制值。在图 4-67 的最下方还有 BIT 的定义，这是给 8051 重要的特殊位（SFR）用的，更进一步的用法将在稍后的章节再做说明。如果我们的汇编语言程序少了 $MOD51 这行会有何结果？请

看图 4-68。在编译过程就会指出 P1 的值未定义，而对应的 LST 文件上也会有类似的错误信息出现。

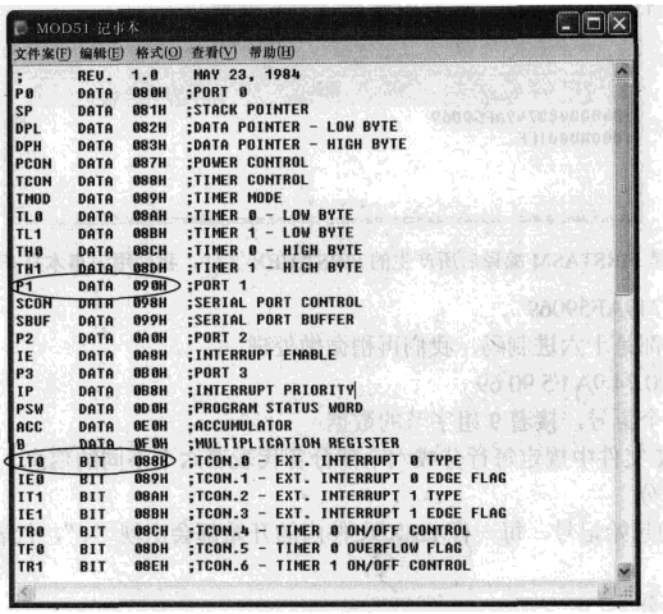


图 4-67　MOD51 文件的部分内容

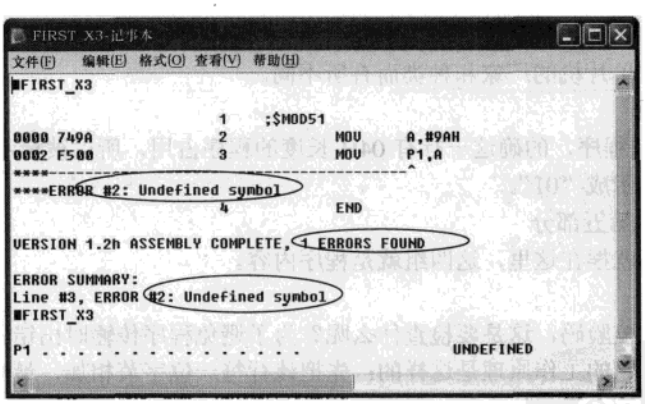


图 4-68　省略$MOD51 是不行的，在 ASM51 阶段就有错误出现

4.12　HEX 文件的认识

我们用 FIRST.HEX 这个 HEX 文件来说明。有关于 HEX 文件格式的更详细说明请参考旗威交流网（www.chipware.com.tw）内的“8051 讲座——认识 HEX 文件”，或是到 www.google.com 用“Intel HEX Format”的关键字搜索。

这个格式最初是由 Intel 公司定义的，规定 HEX 文件内所有的内容都是以 ASCII 格式存放，所以可以直接用记事本打开观察，这种内容格式方便分析但比较占空间。我们先把 FIRST.HEX 第一行拿来做说明，所有内容都是十六进制：

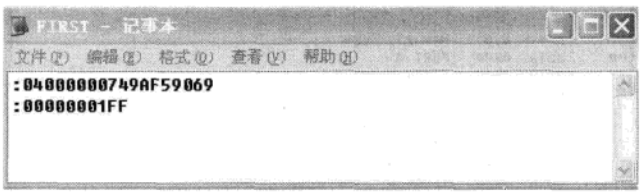


图 4-69　这是 FIRSTASM 编译后所产生的 FIRST.HEX 文件，我们用记事本打开所看到的内容

：04000000749AF59069

第一行这些都是十六进制码，我们再稍微做处理。

：04 00 00 00 74 9A F5 90 69

最前面有一个冒号，接着 9 组字节的数据

Intel 的 HEX 文件中规定每行分成六个部分各代表着六个不同的信息。

[：] 第一部分

HEX 文件的起始记号，每一行 HEX 文件内的开始都会出现“：”，代表这些数据是以十六进制表示。

[04] 第二部分

代表该行程序码所要占的长度，而 04 就是 04H，所以第一行程序占 4 字节大小的空间。

[00 00] 第三部分

标示该行程序码的起始位置，所以 FIRST.HEX 是由 0000H 开始存放的，共存放 4 个位置。这个位置会按单片机的厂家和种类而有所不同。

[00] 第四部分

00 代表该行是程序，的确这一行有 04H 长度的程序占用，所以被标示为 00，如果没有程序占用，会被标示成“01”。

[74 9A F5 90] 第五部分

真正的程序码就摆在这里，这四组就是程序内容。

[69] 第六部分

这一行数据的校验码，这是要检查什么呢？为了避免程序传输时出错，会在每行数据最后加上此校验码。它的工作原理是这样的：先把该行每一位字节相加，最后再加上校验码得到一累加值，不论这个累加值为何，但累加值除上 256（100H）后的余数一定要为 00H，所以此校验码是经过倒推后填入的。

我们试着把第一行的每个字节相加起来：

04H+00H+00H+00H+74H+9AH+F5H+90H=297H

最后 297H 再加上校验码 69H

297H+69H=300H

计算后我们得到 300H，果然末两位是 00H，300H/100H 的余数是 00H。

接下来继续分析第二行的数据：

[：] 第一部分

[00] 第二部分

00H，代表这行并没有任何程序码

[00 00] 第三部分

若有数据时要填入的位置，但此行并未有数据要放

[01] 第四部分

这行并没有占用程序码，所以放 01H

[ ] 第五部分

没有程序码，所以这部分完全是空的

[FF] 第六部分

校验码 FFH 加上第四部分的 FFH+01H=100H，其他都是 00H。最后的结果还是 100H，100H/ 100H 的余数还是 00H，这代表此行的所有数据都是正确的。

经过一连串的分析，让我们对 Intel 的 HEX 文件有进一步的理解，懂得越多就会对 8051 单片机的学习越有信心。

您可以从下列公司的网站取得更进一步的信息：

（1）www.chipware.com.tw：查询 PGM2051 烧录器的参考资料。

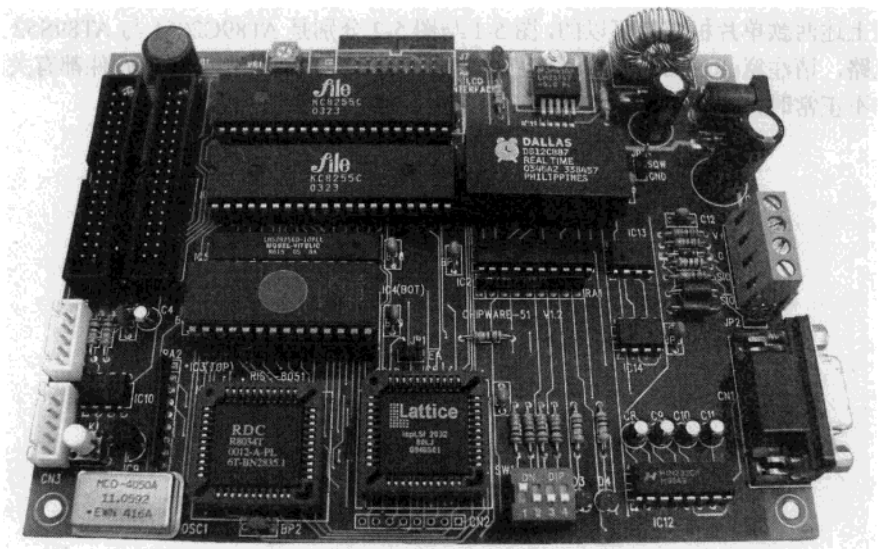
（2）www.topward.com：查询电源供应器的相关资料。

（3）www.fluke.com：查询掌上型电表的相关资料。

（4）www.atmel.com：查询 ASM51 的使用方法。

（5）www.intel.com：查询 8051（MCS51）的相关介绍。

5 ASM51 的操作和熟悉



上一章我们只提到非常简单的 8051 汇编程序，其中最主要的用意，是让初学者熟悉整个学习过程的程序。第 4 章就像是进入了学习的登机门一样，而接下来的这一章就更精彩了！我们要带领您深入 ASM51 的操作环境，为您打通编写 8051 汇编程序的任督二脉，通过本章所介绍的内容，让您轻松学会 8051 的程序格式和软件除错的技巧。

第 5 章　ASM51 的操作和熟悉

5.1　学习 AT89C2051

在本章中，我们还是以 AT89C2051 为学习中心，所有的范例也尽量以 P1 端口和 P3 端口为主。你也可以用容量与引脚较多的 AT89S52，然后用上网的方式下载程序。本章所示范的程序用上述两款单片机都是可以的，图 5-1 与图 5-2 分别是 AT89C2051 与 AT89S52 工作最基本的电路，请注意 RESET 要接对单片机才会动！本章所有的例子跟软硬件都有关系，当发现工作不正常时，请先确定硬件是否正确，然后再检查程序的内容。

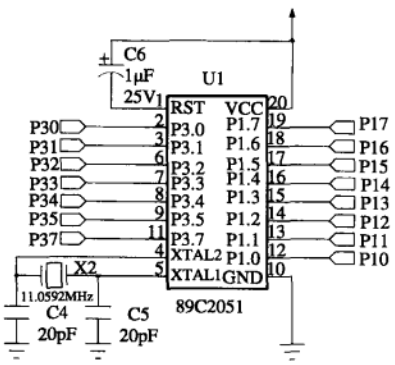


图 5-1　AT89C2051 的实验电路图

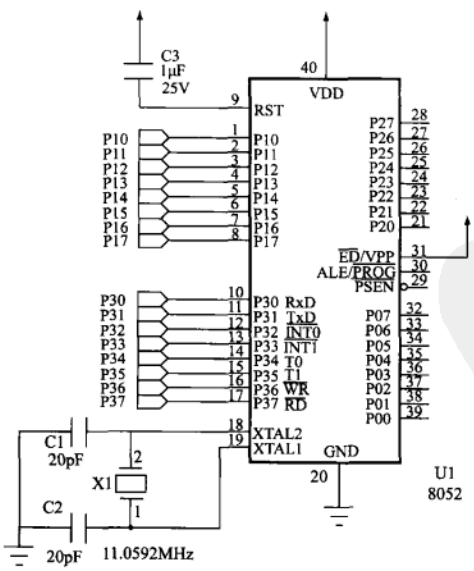


图 5-2　AT89S52 的实验电路图，EA 引脚要接到+5V 即执行内部程序

经验告诉我们：硬件的问题比较好理清，不对的话程序完全不动，这方面反而有许多参考电路可以观摩。软件就比较麻烦了，写法对不一定保证操作对，往往要经过反反复复的修正后，才能得到一个较明确的结果。

5.2 从简单的例子开始

前一章我们已经实际示范整个软硬件的开发过程，同时也包括烧录与面包板的实际接线试验，在真正进入 8051 汇编程序前，我们再把程序多加一行加法指令，使系统变得稍为复杂一些。

【例 5-1】 5-1-1.ASM

```
$MOD51
            MOV         A,#12H      ;A=12H
            ADD         A,#34H      ;A=12H+34H=46H
            MOV         P1,A        ;P1=46H=01000110B
            END
```

这个程序是一个非常典型的 8051 控制程序，程序中只用到两个寄存器（ACC 累加器和 P1 端口），首先 8051 最重要的累加器 ACC（或称为 A）被存入十六进制值 12H，经过 ADD 加法的运算加上十六进制 34H，这时累加器 ACC 的内容已经变成 46H 了，最后把 46H 值传到输出的 P1 端口上，而 P1 端口上又接着 LED 所以输出为 1 的引脚 LED 不亮，而只有输出为状态 0 的 LED 亮。

当程序正确地烧录入 AT89C2051 时，只要一接上+5V 电源的同时，就发现某几个 LED 灯会被点亮，怎么试结果都相同。如果您 DIY 的结果不一样时，请一定要参考前一章节的说明，一步一步地自行找出问题的症结所在。

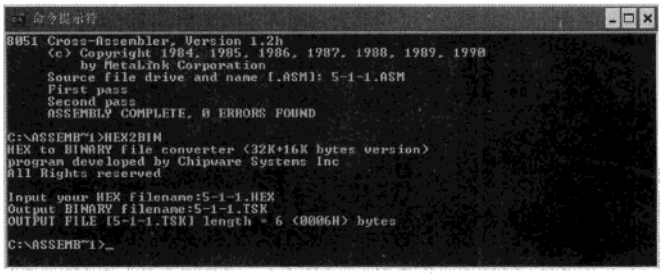


图 5-3 ASM51 编译成功并经过 HEX2BIN 转换文件

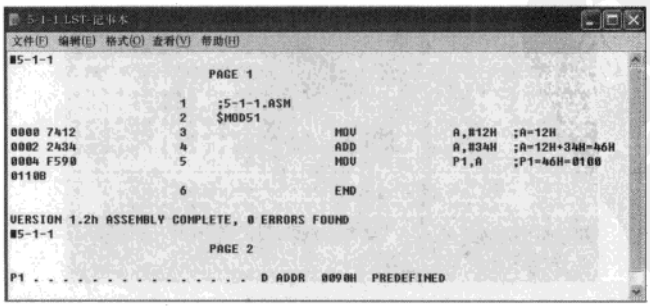


图 5-4 程序 5-1-1.ASM 编译后的 LST 文件

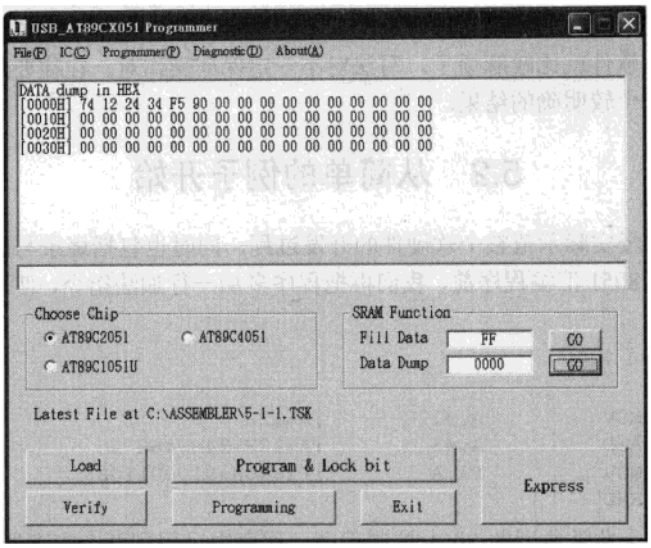


图 5-5　PGM2051 烧录前检查烧录内容是否正确

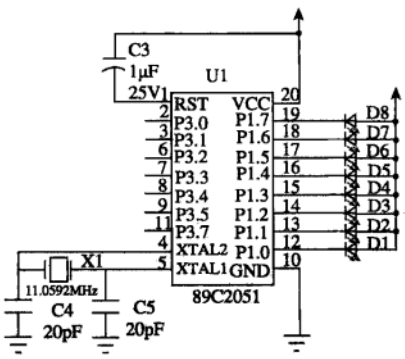


图 5-6　AT89C2051 测试电路图

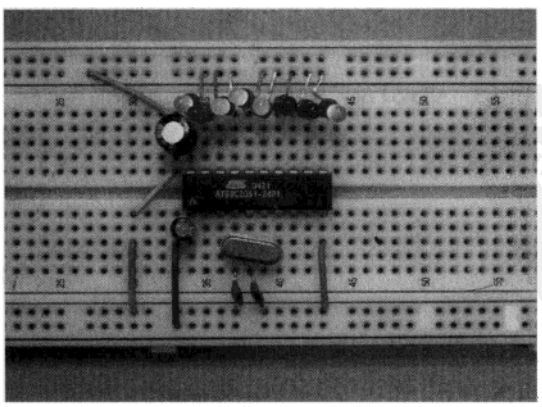


图 5-7　通电测试的结果：有 5 个 LED 灯亮

5.3 LIST 文件的再度观察

【例 5-2】5-2-1.ASM

```
$MOD51
        ORG     0000H
        MOV     A,#12H      ;A=12H
        ADD     A,#34H      ;A=12H+34H=46H
        MOV     P1,A        ;P1=46H=01000110B
        END
```

图 5-8 是 5-2-1.LST 的编译结果文件，真正转出十六进制文件的只有中间的三行，经过上一章的磨练后，我们很容易就归纳出 74H 12H 对应的是 MOV A，#12H 这行，而 24H 34H 就是 ADD A,#34H 加法指令转换后的代码，最后的 F5H 90H 则是 MOV P1，A。这些代码看起来好像是无字天书一样，但仔细分析起来还是有一点规则存在。

例如一看到“MOV A，#”，ASM51 一定编译是成 74H，接下来的值一定是给累加器 ACC 的值。这些十六进制值对人似乎很难理解，但是对 8051 单片机而言是有特定的含义，所以最后才会有正确的 LED 显示结果出现。

当这些有意义的十六进制码被有顺序地存在 8051 的程序存储区内后，一旦使用者对 8051 单片机供电，8051 内部会自动按序找到这些代码，并且依照这些代码所指定的操作来执行，而 LED 才会有所操作。

图 5-8 左边的 0000H 值是程序代码存入的真正地址，我们称之为 Program Counter 程序计数器（简称 PC 值），亦向 PC=0000H 内放入 74H，0001H 内放入 12H，其余的值依次类推，8051 单片机从何处抓取指令码都是由 PC 值决定的。

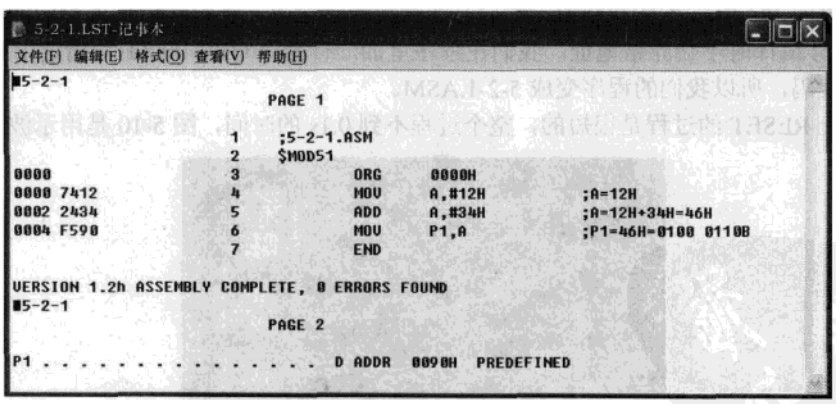


图 5-8 5-2-1.LST 的编译结果文件

但是 8051 怎么知道从哪里拿，怎么拿才不会出错？以前我们曾说过：每次通电后的结果都一样，这意味着 8051 单片机都取得相同且正确的程序代码，因此才会有正确的结果。8051 单片机之所以会这么厉害，还要从 8051 的 RESET 引脚开始说起。

ADDR	DATA	
0000H	74H	MOV A, #12H
0001H	12H	
0002H	24H	ADD A, #34H
0003H	34H	
0004H	F5H	MOV P1, A
0005H	90H	
0006H		
0007H		
程序存储区		AFTER RESET，PC=0000H

图 5-9　在 8051 的程序区中，这六个十六进制数据是由 PC=0000H 开始摆放，这些数据是由 PGM2051 烧录到 AT89C2051 内部的

5.4　RESET 后程序是由这里开始

8051 单片机的系统中有一个重要的信号 RESET，中文称为重置点。每次只要系统收到此重置信号时，8051 内部会进行一连串的重置操作。其中最重要的操作之一是将 PC 值清置成 0000H，亦即强迫 8051 从 0000H 开始抓取指令，程序执行时，最开始是把累加器 ACC 填成 12H，然后加上 34H，最后将结果放到 P1 端口上。除非烧录过程有错误，否则程序执行的结果应该都是相同的。

8051 单片机 RESET 的过程是很重要的，通常当电源一接上时，+5V 的电压不是马上就到达 5V，所以 8051 会保持在 RESET 的状态，等到电压稳定后，单片机 8051 解除 RESET 状态，此时的 PC 值已经是 0000H，然后开始读取程序代码，进行我们预先设定的程序。为了使 ASM51 编译时不会弄错地址，我们在程序里加一行命令 ORG 0000H，指定由 0000H 开始放程序代码，所以我们的程序变成 5-2-1.ASM。

事实上 RESET 的过程是很短的，整个过程不到 0.1s 的时间，图 5-10 是用示波器观察电

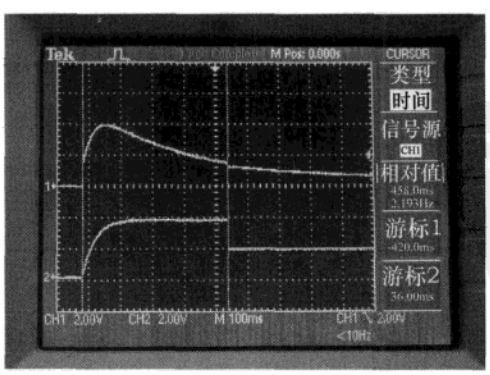


图 5-10　单片机从开始通电到 LED 状态确定的时间。（CH1 是 RESET 脚的信号，CH2 是 P1.0 的信号）由示波器观察到这段时间大约是 456ms，这是因为系统必须等待 RESET 完成后才能开始工作的缘故

源加入到 LED 被点亮的时间。对单片机 8051 而言，RESET 信号是不可少的，如果少了 RESET 的话，我们就无法指定它从哪个 PC 值开始执行。在硬件方面我们还会将整个 RESET 时间保持在 0.1 秒（或更少）以内，以便单片机通电后立即工作。

另外我们可以从 AT89C2051 的 DATA SHEET 上查到，在 RESET 阶段，P1 和 P3 都会被设成数字 1，所以此时 P1 端口上的 LED 是不会亮的。

F8H								FFH
F0H	B 00000000							F7H
E8H								EFH
E0H	ACC 00000000							E7H
D8H								DFH
D0H	PSW 00000000							D7H
C8H								CFH
C0H								C7H
B8H	IP XX000000							BFH
B0H	P3 11111111							B7H
A8H	IE 0X000000							AFH
A0H								A7H
98H	SCON 00000000	SBUF XXXXXXXX						9FH
90H	P1 11111111							97H
88H	TCON 00000000	TMOD 00000000	TL0 00000000	TL1 00000000	TH0 00000000	TH1 00000000		8FH
80H		SP 00000111	DP0L 00000000	DP0H 00000000				PCON 0XXX0000	87H

图 5-11　AT89C2051 RESET 时，所有寄存器的重置值

5.5　8051 开机操作细节解析

单片机 8051 实际工作时的速度是很快的，若其石英晶体的频率为 12MHz 时，执行速度为 1 指令 / μs，即百万之一秒执行一个标准的指令，所以 5-1.ASM 的程序只花了 3μs（百万分之三秒）就做完了，很难想象的快！前一节用示波器观察的结果也是如此。下面我们把 8051 开机所做的事情再细分如下：

8051 不工作	电源尚未加入
8051 RESET 启动	+5V 电源加入的瞬间
…	+5V 电源端逐渐增加
8051 RESET 完成	+5V 电源已稳定

CPU 填入所有重置值，其中 PC 值=0000H

8051 CPU 开始送出 0000H 地址读到第一个程序代码 74H，PC 值自动加 1 成为 0001H。8051 CPU 经过硬件的分析自动知道这是 MOV A，#指令。

8051 CPU 送出 0001H 的地址读回第二个程序代码 12H，PC 值自动加 1 成为 0002H，这时 ACC 会改变成 12H。

8051 CPU 送出 0002H 的地址读到第三个程序代码 24H，PC 值自动加 1 成为 0003H，它很聪明地知道这是 ADD A，#的指令，所以还要再读下一个字节后，才能组合成一个指令。

8051 CPU 送出 0003H 的地址读回第四个程序代码 34H，PC 值自动加 1 成为 0004H，这时 ACC 会完成加法的操作变成 46H。

8051 CPU 送出 0004H 的地址读回第五个程序代码 F5H，PC 值自动加 1 成为 0005H，这时 8051 知道要将数据送出去，但是要读到下一字节才见分晓。

8051 CPU 送出 0005H 的地址读回第六个程序代码 90H，PC 值自动加 1 成为 0006H，此时 CPU 知道 ACC 要送到 P1 端口上，所以最后会将 46H 的值送到 P1 上。

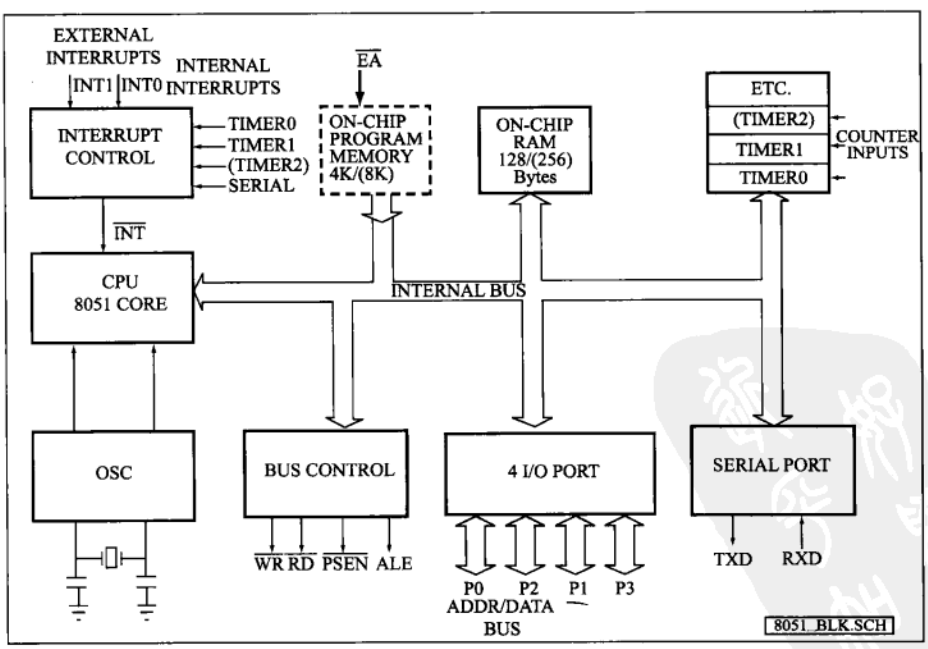
对应的 LED 亮了，到此只花 3μs 的时间。

接下来呢？CPU 继续送出下个 PC 值，分析每个指令然后执行。

8051 单片机执行的速度是非常快的，如果处理得当，8051 绝对是个好帮手。反之那就会发现处处碍手碍脚，其中绝大部分是跟你写的程序是有关系的，8051 单片机本身是不会出错的，会发生错误的 CPU 在出厂前都被拦下当废品了，错的正是你写的程序。

5.6 8051 内部的结构图

到目前为止，我们写的程序中只用到 ACC 累加器和 P1 端口。图 5-12 是 AT89C2051 的内部结构图，你可以找一下 ACC 和 P1 端口在那里。P3 端口除了可以做一般的输入/输出外，还可以做许多的特殊应用，例如定时/计数或串行通信等等，这些都在稍后的章节里提到。



8051/8052 BLOCK DIAGRAM

图 5-12 标准的 8051 内部结构图

图 5-12 是标准 8051 的内部结构图，在各结构的说明上，我们尽量都沿用英文名称，必要时才中英文并列。

系统 RESET 后，8051 内部的 PC 值会变成 0000H，这个值接着放在地址总线，送到图 5-12 中上方的程序存储区（ON-CHIP PROGRAM MEMORY）上，并读到一个字节的数据。在 CPU 8051 CORE 上会进行指令的分析，处理后的结果可以直接送到几个重要的寄存器上，也可以送到输入/输出端口上。

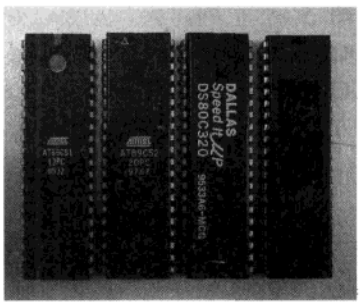


图 5-13　几款常见的 8051 单片机：AT89C51、AT89C52、DS80C320 和 Intel 8051

图 5-14 是 AT89C2051 的结构图，此图更为精确，它把重要的寄存器都列出来了，ACC 累加器是 AT89C2051 中最重要的寄存器，许多计算或逻辑判断都以 ACC 为中心，每次指令执行后会在 PSW 上留下影响的状态值，该值是程序是否执行的依据。

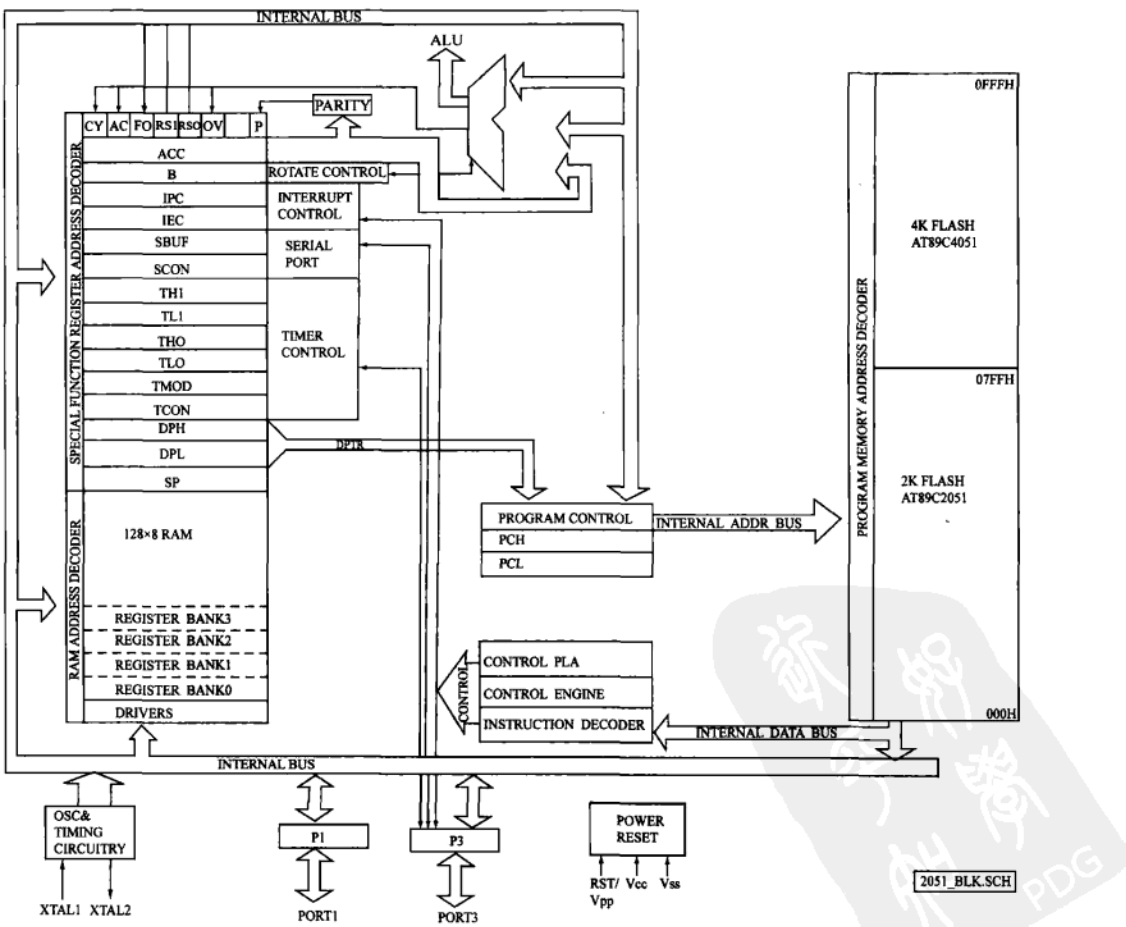


图 5-14　AT89C2051 的内部结构图

图 5-15 也列出来，AT89C2051 和 AT89C4051 有几乎相同的结构，唯一的差别是 AT89C2051 的程序空间为 2KB，而 AT89C4051 为 4KB。其他方面与传统的 8051 完全相同，AT89C2051 内部的程序内存为 FLASH，所以可以用电气的方式将数据烧入或清除。这种烧录的方式是最适合个人 DIY 用的。

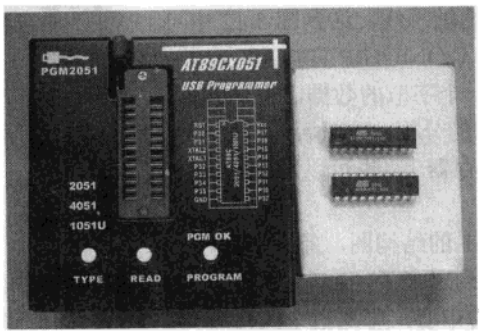


图 5-15　PGM2051 烧录器、AT89C2051 和 AT89C4051 的外观图

5.7　ASM51 的基本用法

到目前为止，我们已经学会如何写一个小小的简单程序，经过编译和 HEX2BIN 后，烧录到 AT89C2051 中，而且用 AT89C2051 驱动 LED 也都成功了。现在我们把 ASM51 编译器的一些重点列在下面：

$MOD51

程序的第一行一定要加这行。

ORG　0000H

指定程序由地址 0000H 开始存放程序代码。

END

程序结束时要加上此行。

另外，ASM51 对英文大小写是一视同仁的，所以 END 和 end 都是一样的，在编写 8051 汇编程序时，可以按您自己的习惯选择全部大写或全部小写。

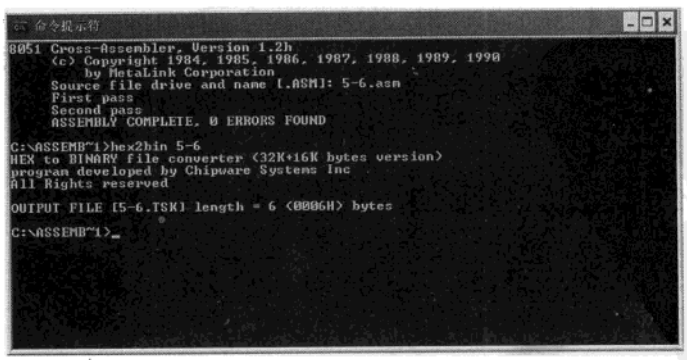


图 5-16　ASM 文件全部用小写编译也是对的

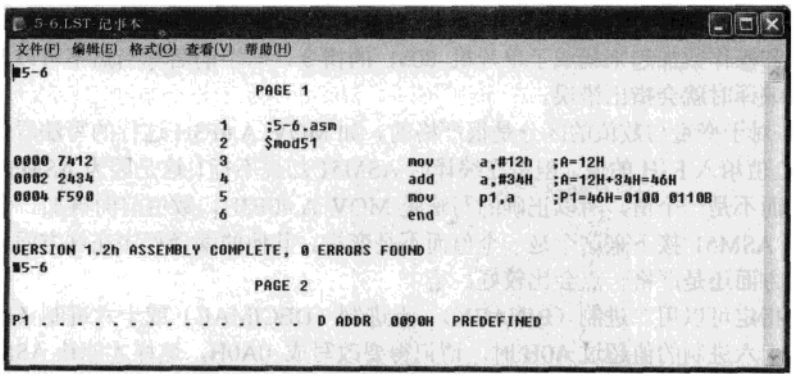


图 5-17　ASM 文件全部用小写编译后产生的 LST 文件，其中的 HEX 文件的数据是对的

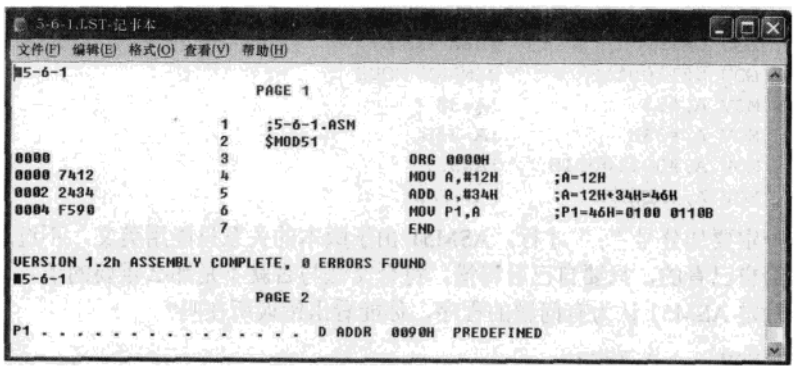


图 5-18　ASM 文件中加上 ORG 0000H 编译后产生的 LST 文件

5.8　ASM51 的用法

每一行 8051 汇编程序可以分成四大部分，分别是：

[名字]　[助记符]　[操作数]　[注释]

START:　MOV　A,#12H　；ACC=12H

名字是由一连串的英文字母或数字组成，最多可以有 32 个字母，但不能包含“#”等符号，但是下划线“_”是可以的，另外名字的第一个字母不可以是数字。

ASM51 又把名字分成两种：SYMBOL（变量）和 LABEL（标签），两者的中文名称几乎一样，但在 ASM51 上还是有差别的。

SYMBOL（变量）

是某个常数的定义或声明，通常 SYMBOL 会放在程序的最前面，且 SYMBOL 最后不需要加上冒号。

LABEL（标签）

标号是一个地址值，经过 ASM51 编译后就会被确定下来，每个 LABEL 后一定要加上冒

号“:”，否则在编译时会指出错误，但是 SYMBOL 之后是不要加冒号的。

助记符和操作数加起来就成了单片机 8051 的指令，名字的名称当然不可以跟助记符相同，否则在编译时就会指出错误。

ASM51 对于变量与数值的区分是很严格的。如 MOV A,#F3H 这行的写法看似无误，意思是把 ACC 值填入 F3H 的值。但是在编译时 ASM51 却说不行！这是因为 ASM51 认为 F3H 是一个变量而不是一个值，所以正确的写法是 MOV A,#0F3H，数值前面再加一个数字 0 就好了，告诉 ASM51 接下来这个是一个值而不是变量。其他的编译程序在这方面可能比较宽松，不过这方面还是严格一点会比较好。

数字的指定可以用二进制（BINARY）、十进制（DECIMAL）或十六进制（HEX）的方式表示，当十六进制的值超过 A0H 时，请记得要改写成 0A0H，这样才能让 ASM51 接受。以下几种数值的写法对 ASM51 来说都是正确的：

```
DEC33   EQU 33              ;DECIMAL MODE
HEX33   EQU 33H             ;HEX MODE
HEXFA   EQU OFAH            ;HEX MODE
BIN33   EQU 00110011B       BINARY MODE
        MOV A,#33           'A=33
        MOV A,#33H          ;A=33H
        MOV A,#00110011B    ;A=33H
        MOV A,#HEXFA        ; A=FAH
```

注释前一定要加分号“;”才行，ASM51 由于版本的关系只能用英文，不过这些程序注释通常是写给自己看的，只要自己看得懂，符合文法与否就不是那么重要的了。

以下几行是 ASM51 认为有问题的程序，你能看出错误所在吗?

```
1TEST:  MOV     A,#2FH
        ADD     A,#ABH
        ADC     A,#F6H
TEST_PROGRAM_STARTHERE:
        MOV     P1,#330
;       END
```

5.9 ASM51 常用伪指令的用法

汇编语言的规模可以非常小，就像前一章所示范的 FIRST.ASM 程序，只有短短数行而已。但是，汇编语言也可以写到超过 10000 行以上的大型程序，这时就要借助 ASM51 的伪指令（Directive）了。

所谓的伪指令说穿了就是不是真正的 8051 指令，它是发展 ASM51 ASSEMBLER 的厂商另外扩展出来的，以协助程序员更方便编写程序，如果我们查看 ASM51 编译出的 LST 文件时，可以明显地看出伪指令是无法产生对应的 8051 指令的。写程序不就是把 8051 的指令一行一行填上去吗，难到还有特别快的方法？请看以下例子：

如果有一个程序内出现 MOV A，#36H 的次数有 30 多次。

```
STEP1:      MOV     A,#36H
...
```

```
STEP2:      MOV     A,#36H
...
...
STEP30:     MOV     A,#36H
...
            END
```

如果有一天突然发现 36H 这个值是不对的，应该改为 58H 才对，如果是你不懂应用伪指令的话，可能就要花点时间直接找到这些位置，然后逐一做修正？但是，ASM51 有个伪指令叫 EQU，使用这种写法才会更方便。

```
PARA1   EQU         36H    ;REPLACE ALL PARA1 WITH 36H
STEP1:  MOV         A,#PARA1
STEP2   MOV         A,#PARA1
...
...
STEP30: MOV         A,#PARA1
        END
```

当程序需要修改时，只要修改第一行为 PARA EQU 58H 就行了。这样不但快速而且不容易出错。以下就是几个常用的伪指令说明，除了学会 8051 的指令外，认识常用的伪指令，一定会使你写的程序更有灵活性。

ORG 伪指令

指定 ASM51 从哪个地址开始存放程序代码，在一个程序中 ORG 可以出现许多次。

END 伪指令

定义 END 之后就没有任何指令了。

DB 伪指令

这是 Define Byte 的缩写，ASM51 遇到 DB 指令时，会把 DB 后的数据以 8 位（BYTE）的方式存到程序存储区内。

DW 伪指令

这是 Define Word 的缩写，ASM51 遇到 DW 指令时，会把 DW 后的数据以 16 位（WORD）的方式存到程序存储区内。

5-8-1 ASM 伪指令的程序示范

【例 5-3】 5-8-1.ASM

```
;DEMO   THE USE OF 'DB' AND 'DW'
RESET   EQU     0000H
RANGE   EQU     2680H
;
$MOD51
        ORG     RESET               ;RESET FROM 0000H
        MOV     A,#12H              ;A=12H
        ADD     A,#00110100B        ;A=A+34H
        MOV     P1,A                ;P1=A
;
```

```
TABLE:
        DB      '0'
        DB      '1'
        DB      '2'
        DB      '3456789'
SCALE:
        DW      1234
        DW      0FFDBH
        DW      RANGE/2

        END
```

DB 和 DW 是很常见的伪指令，主要用在查表和字符串定义，这些写法和用法将会在下一章的 MOVC 指令中说明。DB 与 DW 所指定的数据是要存在程序存储区中。

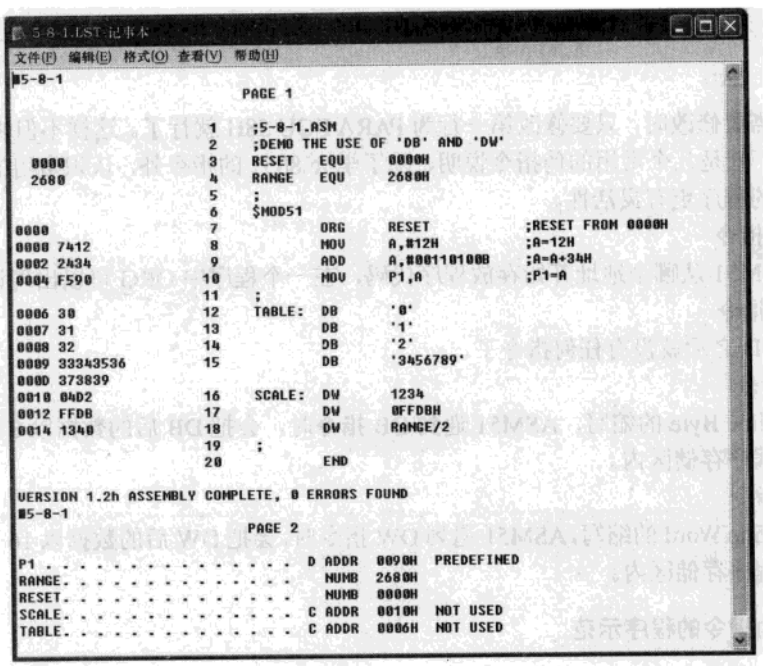


图 5-19　5-8-1.ASM 编译后的 LIST 文件，请注意 DB 和 DW 的用法，“1”是指 1 的 ASCII 码为 31H，而“3456789”则是一字符串。DW 可以指定一个 16 位值，RANGE 在程序最前面被声明为 2680H，而 RANGE/2 刚好是 1340H。ASM51 可以在编译阶段进行简单的算术操作

5.10　软件除错的写法

ASM51 也提供条件式（Conditional）的伪指令。这方面的伪指令包括 IF、ELSE 和 ENDIF 三个。假设在程序开发阶段，我们一直希望能把系统状态显示在 P1 端口上。当程序完成后，我们要把上述的功能取消时，只要把 DEBUG 的值重新设成 0，然后再执行一次 ASM51 就行了。

【例 5-4】5-9-1.ASM

```
RESET   EQU     0000H                   ;RESET
DEBUG   EQU     1                       ;PROGRAM IN DEBUG MODE
$MOD51
        ORG     RESET                   ;RESET FROM 0000H
        MOV     A,#12H                  ;A=12H
        ADD     A,#00110100B            ;A=A+34H
IF(DEBUG)
        MOV     P1,A                    ;P1=A IN DEBUG MODE
ELSE
        MOV     SBUF,A                  ;SBUF=A IN NOT DEBUG MODE
ENDIF
END
```

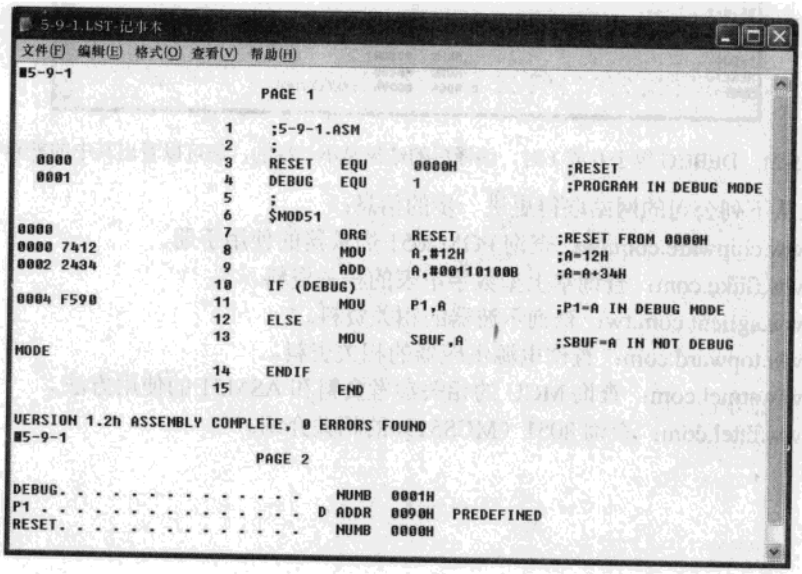
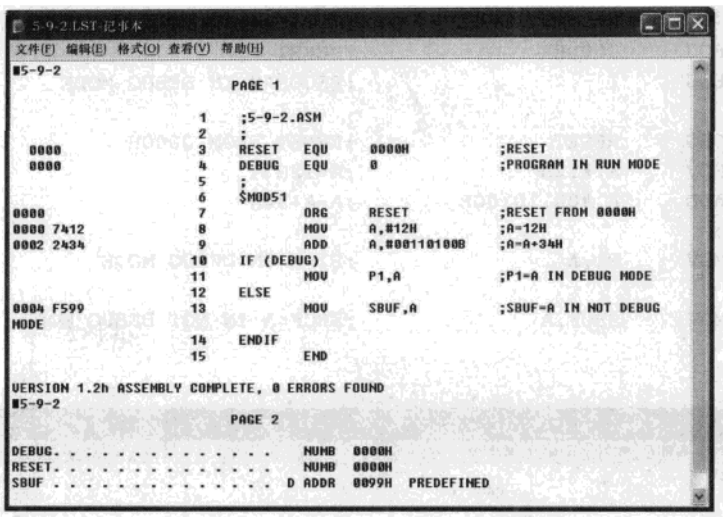


图 5-20　5-9-1.ASM 编译后的文件

【例 5-5】5-9-2.ASM

```
;
RESET   EQU     0000H                   ;RESET
DEBUG   EQU     0                       ;PROGRAM IN RUN MODE
;
$MOD51
        ORG     RESET                   ;RESET FROM 0000H
        MOV     A,#12H                  ;A=12H
        ADD     A,#00110100B            ;A=A+34H
IF(DEBUG)
        MOV     P1,A                    ;P1=A IN DEBUG MODE
ELSE
        MOV     SBUF,A                  ;SBUF=A IN NOT DEBUG MODE
ENDIF
        END
```



5-9-2.LST-记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
■5-9-2
                        PAGE 1

                    1   ;5-9-2.ASM
                    2   ;
  0000              3   RESET   EQU     0000H           ;RESET
  0000              4   DEBUG   EQU     0               ;PROGRAM IN RUN MODE
                    5   ;
                    6   $MOD51
0000                7           ORG     RESET           ;RESET FROM 0000H
0000 7412           8           MOV     A,#12H          ;A=12H
0002 2434           9           ADD     A,#00110100B    ;A=A+34H
                   10   IF (DEBUG)
                   11           MOV     P1,A            ;P1=A IN DEBUG MODE
                   12   ELSE
0004 F599          13           MOV     SBUF,A          ;SBUF=A IN NOT DEBUG
MODE
                   14   ENDIF
                   15           END

VERSION 1.2h ASSEMBLY COMPLETE, 0 ERRORS FOUND
■5-9-2
                        PAGE 2

DEBUG. . . . . .    NUMB   0000H
RESET. . . . . .    NUMB   0000H
SBUF . . . . . .  D ADDR   0099H  PREDEFINED
```

图 5-21　DEBUG 等于 0 或 1 时，编译后的结果是不一样的，您可以看出其中的差异吗

您可以从下列公司的网站取得更进一步的信息：

（1）www.chipware.com.tw：查询 PGM2051 烧录器的使用手册。

（2）www.fluke.com：查询掌上型数字电表的相关资料。

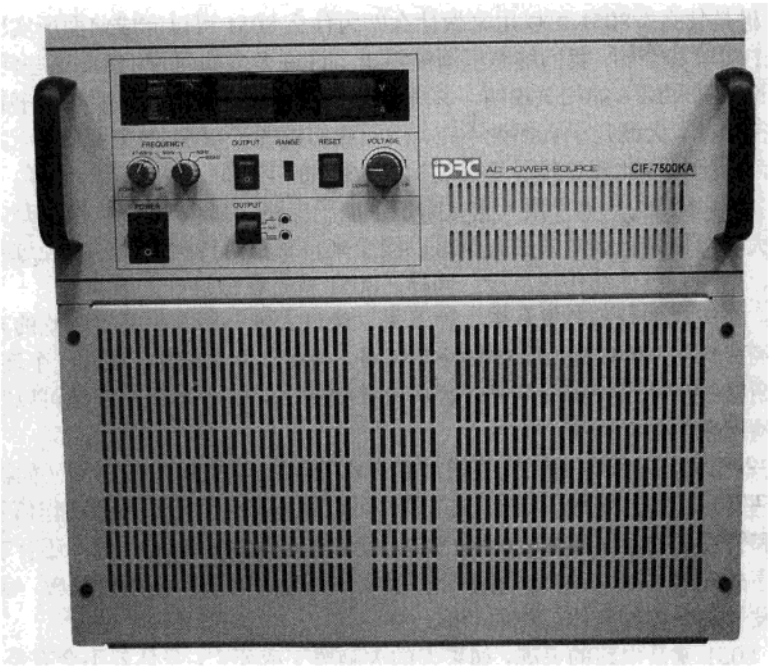
（3）www.agilent.com.tw：查询示波器的相关资料。

（4）www.topward.com：查询电源供应器的相关资料。

（5）www.atmel.com：查询 MCU 的相关参考资料和 ASM51 的使用方法。

（6）www.intel.com：查询 8051（MCS51）的相关介绍。

6 8051的系统结构



8051的系统结构是学习8位单片机硬件组成很好的一个范例。绝大多数单片机的基本工作原理，跟8051是很相似的！本章用了大量的范例和图表，详细剖析8051的硬件工作原理，相信您在仔细读完本章的内容之后，一定能大大提高您设计8051程序的能力，让程序的工作更流畅。事不宜迟，我们现在就开始吧！

第6章　8051的系统结构

6.1　基 本 结 构

8051单片机是什么？8051可以用来做什么？为什么8051可以完成前面所实验的操作？经过前面几章的实验和介绍，我们脑海里渐渐浮出了许许多多的问号，然而这些问题的背后，有着庞大的理论数据和深入的相关说明，并非三言两语可以讲完。如果要在本书把这些原理和过程完整交待清楚，大概几千页都说不完，因此我们挑选了8051的主要概念，并用举例的方式加以说明，希望可以比理论推导更贴切地将8051的工作原理做全新的诠释。

谈到8051的基本结构，如果用人来比喻整个电子电路，那么8051大概就是人的大脑。先想一想人的大脑可以做什么？人的大脑可以接受来自感官所传递的信号，经过判断，进而产生面部表情、发声、肢体动作的反应。同样，8051就是在做这样的事情。

举例来说：今天我们看到苹果从树上掉下来，经过大脑分析苹果是可以吃的东西，掉下来代表果实已经成熟了，所以命令身体去捡起苹果，并擦干净拿来吃。这样一个连贯的动作，必须先通过眼睛接收到苹果掉下的信号，经头脑寻找过去记忆中苹果是成熟可以吃的信息，再下达用手去捡起来吃的动作。

8051的基本结构，就跟人的头脑很像：它有掌管输入/输出信号的引脚（如输入输出端口），就像大脑里和肢体感官衔接的神经元一样，可以用来接受信号，传递动作信号；也有掌管数据的存储和管理的内存（如RAM、ROM），就像大脑里的存储区一样；还有掌管运算与逻辑判断的程序处理器（如CPU），用来分析接收信号是否有效，该如何反应，就像头脑里掌信号分析和发号施令控制身体动作的功能一样。

换句话说，8051就是电路的灵魂，如果人的大脑罢工或死亡，身体就不会有反应和动作；同理，8051如果不工作或损毁，整个电路就不会有动作产生。归纳前面所提到的重点，8051的基本结构可以分为以下三个大类：CPU、内存与输入/输出三大部分：

（1）中央处理器（CPU）

根据程序事先的设计，进行数据运算及运算结果的传递和储存。

（2）内存（Memory）

8051单片机又把存储器分成两部分：

①数据存储器（Data Memory），这部分是可以随时修改的。运算时用来存放临时数据的存储器，就像算术时用手指头或纸笔来记录进位一样。

②程序存储器（Program Memory），存储程序代码的存储器，这部分是无法做任何更改的，8051一切操作都是按照程序规划来执行的。

（3）输入/输出端口（P1、P3）

信号接收与传递的窗口，可外接输入/输出与其他电子元件。若是标准的8051则还多了P0和P2端口。

到此为止，我们对8051的基本结构有了初步的概念，接下来的章节要一一来深入探讨。

图 6-1 是 AT89C2051 的结构图，这是一张非常重要的图，最能展现和说明 AT89C2051 的内部结构。

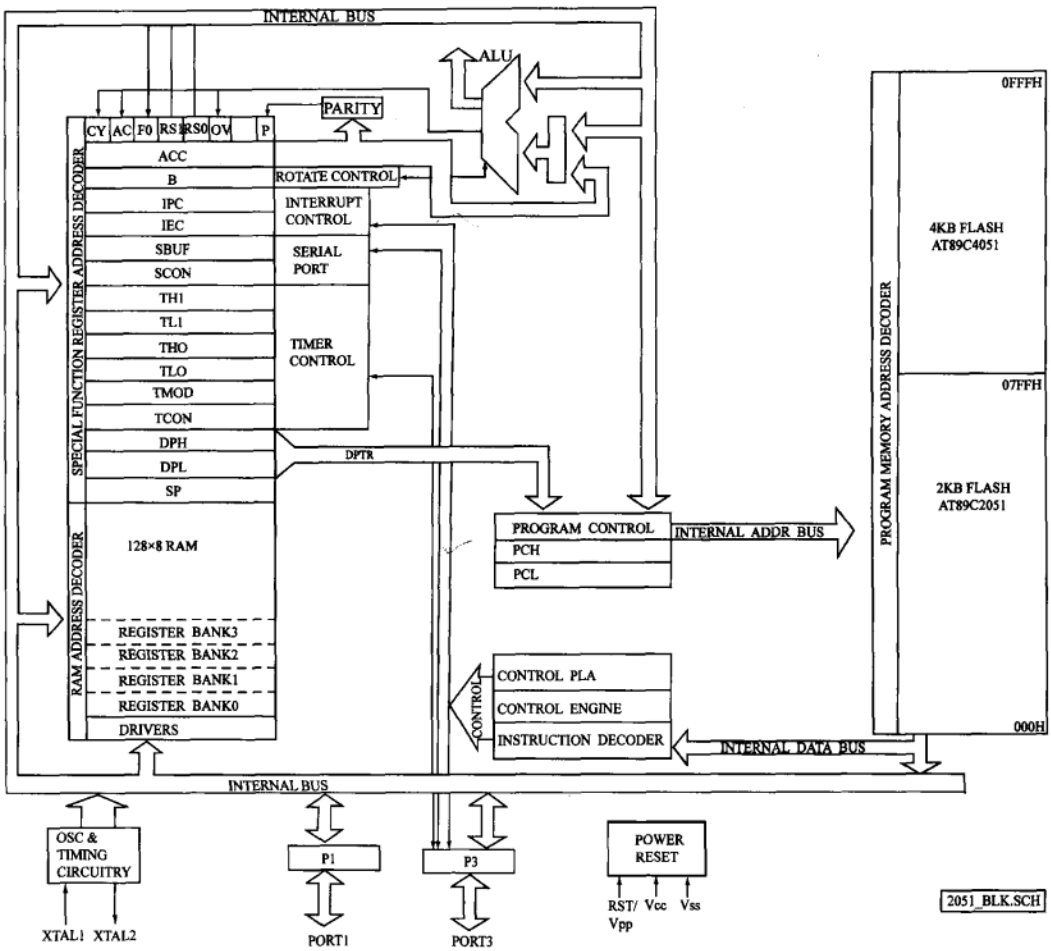


图 6-1　AT89C2051 结构图

请从左边看起，左边整排是 Data Memory，其实有 256 字节，但又分为两部分：一半给系统当存储区，另一半是可以自由读写的 RAM 区。左上方依序是重要的寄存器 PSW（状态寄存器）、ACC 累加器和 B 寄存器等等，SP 堆栈寄存器以下则是 RAM 区，其中包括 R0~R7 寄存器也归在这里。

图 6-1 中下方有一个方块标示：INSTRUCTION DECODER 指令译码部分，所有从程序区来的数据会在这里进行分析，并产生对应的控制信号到 ALU（算数与逻辑处理单元）和特殊功能寄存器（SPECIAL FUNCTION REGISTER）上。

图的左下方是石英晶振的处理方块，右下方则是电源输入和重置处理电路。P1 与 P3 端口在中下方。图的右方则是程序存储器，AT89C2051 只有 2KB 的程序空间，其中 PC 值可以由 0000H~ 07FFH，而 AT89C4051 就有多一倍的程序空间，PC 值可达 0FFFH。

6.2 CPU 如何工作

8051 单片机如何工作？首先要写一个 8051 的汇编程序，并且经过 ASM51 编译无误后，烧录到 AT89C2051 的 IC 当中，这是软件方面的规划。

硬件方面我们要准备石英晶振和+5V 电源，并且在 AT89C2051 的 RESET 脚上加一枚重置用的电容。然后在 P1 端口接上 LED，当所有线路都检查就绪后，通电后看 LED 的显示，就可以得知程序结果，这些都是我们可以 DIY 的部分。

事实上，我们操作 PGM2051 烧录器，是把程序烧录到 AT89C2051 内部 Program Memory 的 FLASH 上，请再往前翻一页看一下 AT89C2051 的结构图，FLASH 正是图右边的大部分。当我们在 PGM2051 烧录器下达 Erase 命令时，整块 FLASH 程序区会全部被清除成 FFH，而整个过程不到 10ms（即百分之一秒）就完成了。FLASH 被清除成 FFH 的 AT89C2051 直接通电会有什么操作吗？你认为的答案呢？

如果你的答案是 CPU 完全不工作，看起来好像是对的，因为所有的输入/输出都没有变化，通电试了几次的结果都真的没有任何变化。可是如果仔细推敲的话，好像有一些疑问存在：石英晶振在工作着，RESET 脚也在状态 0，8051 单片机这时不晓得在做什么？其实这时候 8051 还是在工作的，但是因为 FLASH 上的数据都是 FFH，这个数据若比较标准 8051 的指令刚好是 MOV R7,A，也就是把 A 的值填入 R7 寄存器内。

如果整个存储区内都是 FFH 时，则 8051 从头到尾都是在做 MOV R7,A 这个相同的指令，所以 CPU 并未停止它存取指令的操作，只是它一直重复在做 MOV R7,A 这个操作，直到电源被关掉为止。

我们特别写个程序来试试看，先来看看下面的例子。

【例 6-1】 6-2-1.ASM

```
$MOD51
        ORG     0000H
        CPL     A
        MOV     P1,A        ;P1=ACC
        END
```

这个程序的意思是要把累加器 A 的值取反（CPL），然后送到 P1 端口输出，你要用前面所提到的 ASM51 编译器，将程序编译为 HEX 文件，再转换为 BIN 文件烧录到 AT89C201 中，接着把实验过的面包板电路再搬出来用，通过 LED 把状态显示出来。

当我们用示波器观察 P1 端口上的变化时，却看到一个方波信号，频率约在 225Hz 左右，这是因为 CPU 并不知道要停止，所以做完 MOV P1，A 指令后，PC 值还是一直增加，但是每次都是做 MOV R7,A 的操作，直到 PC 值等于 AT89C2051 的上限值 07FFH，下一个 PC 值又返回到 0000H，8051 单片机又从头做一次 A 取反操作。所以 8051 开机后程序只要做一次的写法就变成下面的样子：

【例 6-2】 6-2-2.ASM

```
$MOD51
        ORG     0000H
        CPL     A
        MOV     P1,A        ;P1=ACC
```

```
LOOP:
        SJMP    LOOP        ;ENDLESS LOOP
        END
```

不过，要规划程序工作，还要知道输入/输出端口如何控制，内存如何使用及外面的电子元件怎么接，不单单只是知道程序的指令怎么用就行。

6.3 P1 的工作模式

介绍完 CPU 的工作重点后，接着来看看 8051 要如何进行输入/输出的操作，首先要介绍的就是 P1 端口。

在 6-2-1.ASM 的范例中，实现了从 AT89C2051 的 P1 端口输出，可是在实现的过程，我们发现了一个特别的现象：为什么 LED 点亮时的亮度会不一样？现在我们把 P1 端口的输出改成 FFH，让每一只引脚的输出电压电平都是 HIGH，再用电表一一测量输出的电压。

【例 6-3】 6-3-1.ASM

```
$MOD51
        MOV     P1,#0FFH    ;把 P1 端口的输出通通填成 HIGH
        END
```

将这个程序编译好烧录到 AT89C2051 中，就可以让 AT89C2051 的所有引脚都输出 HIGH，经过电表测量所得到的数据，我们发现 P1.0 和 P1.1 的电压，很明显跟其他的引脚不同，这是因为 AT89C2051 的 P1.0 和 P1.1 还兼做比较器的功能，多少会影响输出的电压。比较好的方法是在 LED 与正电源间串上一个 1kΩ 的电阻，将流过 LED 的电流限制在 5V/1kΩ=5mA 左右，这种安排就可以得到比较均匀的亮度了。

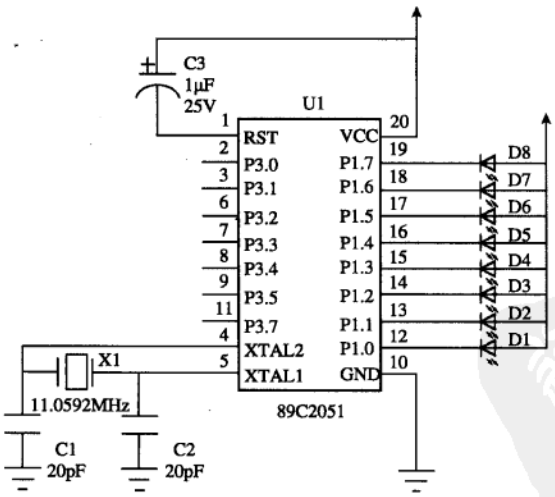


图 6-2 原先的电路图

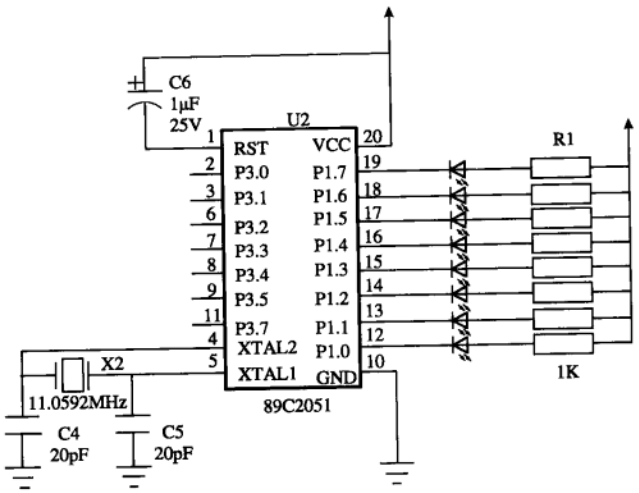


图 6-3　加上 1kΩ 电阻的路线图

如果做输入的话，只要利用下面这段程序就可以读取了。读取到的数据，会存储在 ACC 累加器中。

【例 6-4】 6-3-2.ASM

```
$MOD51
            ORG     0000H
            MOV     P1,#0FFH        ;P1 in INPUT mode
            MOV     A,P1            ;A=P1
            END
```

问题来了，什么是 ACC 累加器？别急别急，我们在 6.5 中会详细的介绍。

6.4　P3 的工作模式

P3 端口的输入/输出控制方式，跟 P1 端口是完全一样的，在程序的写法上，只要把 P1 改成 P3 就可以了。

【例 6-5】 6-4-1.ASM

```
;P3=OUTPUT MODE
$MOD51
            MOV     P3,#0AAH        ;P3=10101010B
            END
```

【例 6-6】 6-4-2.ASM

```
;P3=INPUT MODE
$MOD51
            MOV     P3,#0FFH        ;PUT P3 in INPUT MODE
            MOV     A,P3            ;READ P3
            END
```

不过 P3 端口除了可以用来作一般的输入/输出功能外，还有一些特别的用途。下面是各个引脚的列表。

表 6-1　AT89C2051 的 P3 端口各引脚功能一览，其中 P3.6 为 P1.0 和 P1.1 的比较输出

P3.0	RxD（串行输出端口）
P3.1	TxD（串行输出端口）
P3.2	INT0（外部中断 0）
P3.3	INT1（外部中断 1）
P3.4	T0（外部定时计数输入点）
P3.5	T1（外部定时计数输入点）
P3.6	当电压电平 P1.0>P1.1，P3.6=1；当电压电平 P1.0<P1.1，P3.6=0

AT89C2051 的 P3 端口除了一般的输入/输出控制外，它还能做串行通信、定时/计数和外部中断的功能，在稍后的章节再做介绍。

AT89C2051 是无法进行程序空间扩充的，对于一般的控制应用，2KB 的程序空间应该已经够用了。如果程序超过 2KB 但仍少于 4KB 的话，也可以用 AT89C4051 来替代。

传统 40 引脚的 8051 仍有程序空间不够的问题，所以有部分引脚是供做外部程序或数据存储器存取功能用，可以用来扩充外部存储器。因为早期制作存储器的技术还不够纯熟，内置存储器会增加很多 CPU 的成本，如今绝大多数的 8051 都已经内置足够的存储器空间，且外挂存储器还有速度与兼容性的问题，这个功能在较新的 CPU 规划时，就渐渐被省略了。

6.5　重要的寄存器认识（一）ACC 累加器

介绍完 8051 的输入/输出端口，我们来看看 8051 最常用也最重要的一些特殊功能寄存器，首先要介绍的，就是前面所看到的 ACC 累加器。

8051 所有跟算术或逻辑运算有关的功能，都要通过 ACC 累加器。不过 ACC 累加器只有 8 位，所以运算时被限制在 0～255（00H～FFH）。

【例 6-7】 6-5-1.ASM

```
$MOD51
                MOV     A,#01H          ;A=01H
                ADD     A,#02H          ;A=A+02H
                MOV     P1,A            ;P1=A
LOOP:           SJMP    LOOP
                END
```

通过 P1 端口，我们可以看到运算的结果显示为 0000 0011B（03H）。如果运算值超过 0～255 的范围，那么 ACC 累加器中的数据会变成什么呢？

【例 6-8】 6-5-2.ASM

```
$MOD51
```

```
        MOV         A,#100      ;A=100
        ADD         A,#200      ;A=A+200
        MOV         P1,A        ;P1=A
LOOP:   SJMP        LOOP
        END
```

通过 P1 端口，我们可以看到运算的结果显示为 0100 0100B（44H）。

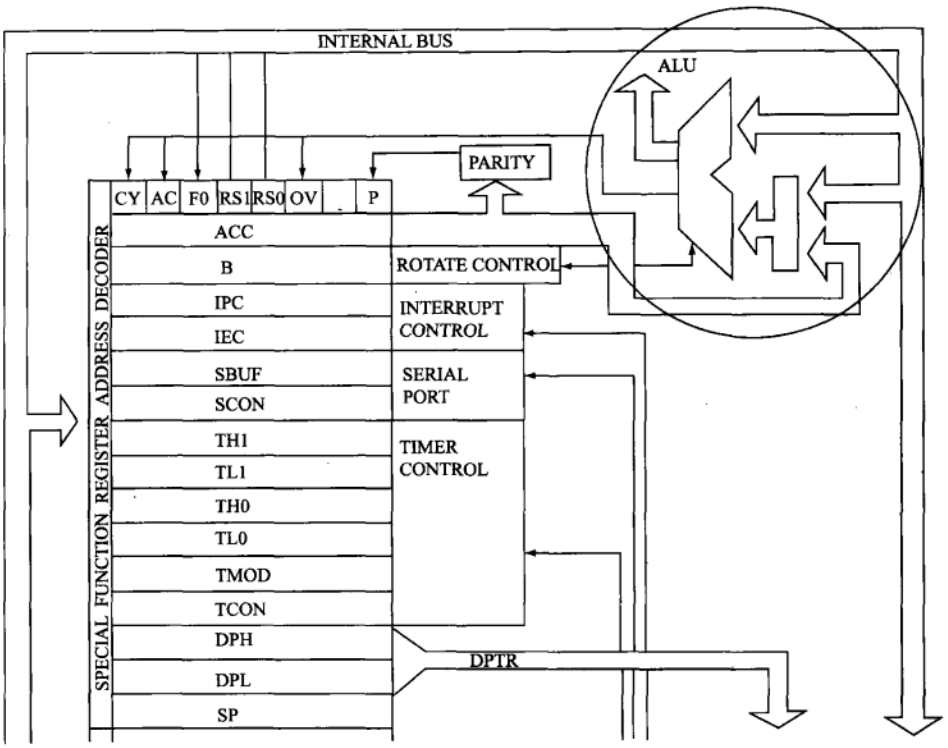


图 6-4　AT89C2051 的 ACC 累加器是最重要的寄存器，所有的逻辑和算术经 ALU 处理后，才会再转存到 ACC 累加器或其他寄存器上

为什么结果是 44H？因为 256 以上的数值，ACC 累加器无法存放，但是进位的结果会保存在 PSW 状态寄存器上。所以就会把 256 以上的部分去掉，留下 0～255 可以表现的数值（300−256=44H）。

除了算术运算外，ACC 累加器也是某些特定功能所指定寄存器（如串行通信、索引功能、外部存储器的存取），这些在稍后的内容会陆续介绍。

问题又来了：假如我要运算的数值超过 0～255 的范围，那我应该怎么办？

6.6　重要的寄存器认识（二）PSW 寄存器

其实，ACC 累加器并非真的把 256 以上的数值给舍弃了，而是通过另外一个寄存器把运算时进位的信息存储起来，而这个寄存器就是我们接下来要介绍的 PSW 寄存器。

表 6-2 PSW 各个位一览表

（DOH）

CY	AC	F0	RS1	RS0	OV	—	P
D7	D6	D5	D4	D3	D2	D1	D0

PSW 的正式名称为“Program Status Word”（程序状态字），它是用来显示 CPU 中几个关键操作的状态，像前面所提到的进位问题，就是用 PSW 里的 CY 标志位来查看算术运算是否进位，如果有进位，CY 标志位的值就会等于 1，如果没有进位，CY 值会等于 0。

【例 6-9】6-6-1.ASM

```
$MOD51
                MOV     A,#01H          ;把累加器的值设为 1
                ADD     A,#02H          ;把累加器的值加上 2
                JC      ON              ;检查 CY 的值
;
OFF:            MOV     P1,#00H         ;如果 CY=0,P1 输出为 LOW
                SJMP    $
;
ON;             MOV     P1,#0FFH        ;如果 CY=1,P1 输出为 HIGH
                SJMP    $
                END
```

通过电表去测量 P1 端口的引脚，我们发现所有的引脚都是 LOW 电平，所以 ACC 累加器在运算过程中是没有进位的。

【例 6-10】6-6-2.ASM

```
$MOD51
                MOV     A,#100          ;把累加器的值设为 100
                ADD     A,#200          ;把累加器的值加上 200
                JC      ON              ;检查 CY 的值
;
OFF:            MOV     P1,#00H         ;如果 CY=0,P1 输出为 LOW
                SJMP    $
;
ON;             MOV     P1,#0FFH        ;如果 CY=1,P1 输出为 HIGH
                JMP     $
                END
```

通过电表去测量 P1 端口的引脚，我们发现所有的引脚电平都是 HIGH 电平，所以 ACC 累加器在运算过程有进位。

PSW 寄存器中的其他位，在列表中相对应的章节里会有更详细的介绍。

6.7 重要的寄存器认识（三）DPTR 寄存器

8051 有一个很特别的寄存器叫 DPTR，为什么特别？因为它是 8051 中唯一的一组 16 位寄存器，它的主要功能有两个：第一是程序内部的数据索引功能，第二是存取外部存储器的数据。

如果要进行索引功能（查表）时，DPTR 所代表的是内部程序存储器的地址，同样必须配合 ACC 累加器才能存取。以下是相关的程序范例：

【例 6-11】6-7-1.ASM

```
$MOD51
            MOV     DPTR,#TABLE     ;把 DPTR 填入查表的起始位置
            MOV     A,#02H          ;查询表格的第二项数据
            MOVC    A,@A+DPTR       ;把查表数据存储到 ACC 累加器
            MOV     P1,A            ;把查表数据用 P1 端口显现出来
            SJMP    $
TABLE:
            DB      11H             ;表格内容
            DB      22H
            DB      33H
            DB      44H
            DB      55H
            END
```

猜猜看，查到的表格数据应该是多少？把这个程序编译好下载到 CPU 中，就可以得到正确答案了。P1 端口所呈现的数值应该为 33H，11H 是第 0 项数据，由于 ACC 的值是 02H，而 DPTR 指向 TABLE 表，A+DPTR 的位置刚刚好指到 TABLE+2，即存放 33H 的地址。

当 DPTR 用来存取外部存储器时，DPTR 所代表的是外部数据存储器的地址，必须配合 ACC 累加器才能存取。我们看看下面的范例，不过此例无法应用在 AT89C2051 上，因为 AT89C2051 本身是无法扩充外部存储器的。

【例 6-12】6-7-2.ASM

```
$MOD51
            MOV     DPTR,#1234H     ;在 DPTR 填入外部存储器的地址
            MOVX    A,@DPTR         ;把 1234H 地址上的数据存到 ACC 累加器
            END
```

【例 6-13】6-7-3.ASM

```
$MOD51
            MOV     DPTR,#1234H     ;在 DPTR 填入外部存储器的地址
            MOVX    @DPTR,A         ;把 ACC 累加器的数据写到 1234H 地址上
            END
```

6-7-2.ASM 是读取的用法，6-7-3.ASM 是存储的方法。

到此为止，我们知道 DPTR 是 8051 用来表达存储器地址的寄存器，问题是：什么叫存储器地址？为什么会有外部和内部存储器的区别？程序存储器跟数据存储器又有何不同？这一连串的问题，只要认识 8051 的存储器结构后，一切就迎刃而解了。

6.8 重要的寄存器认识（四）SP 和 B 寄存器

SP 寄存器我们又称为堆栈寄存器，这个寄存器存放一个指针值。SP 寄存器通常在程序开始就先定义一次，以后就不理它了。SP 是 STACK POINTER 的缩写，每当我们的程序做 CALL 调用时，CALL 指令就会把此时的 PC 值共 16 位放到 SP 所指定的 DATA

MEMORY 中。当程序执行到 RET 返回的指令时，又会把上次暂存的 PC 值取回，继续原有程序的执行。

在 DATA MEMORY 中，SP 值以上的区域不应该再放置任何值，亦即此区域是供程序暂存 PC 值用的，若再存放其他值时，程序执行当中有可能无意间会被覆盖掉。以 AT89C2051 为例，系统一开始工作后，会把 SP 值填为 60H，这表示 60H～7FH 共 32 字节是给系统用的，你不可以把其他变量放在此区域中。

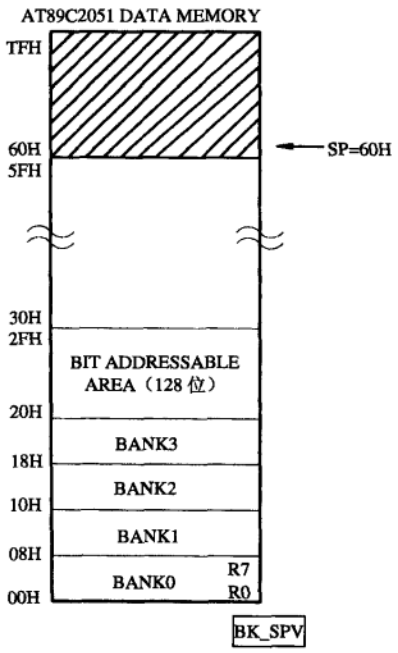


图 6-5　SP 设定示意图，当 8051 程序的堆栈值设定为 60H 时，代表 60H～7FH 斜体区域是不可以放置任何变量的。所以真正可以放变量的区域为 30H～5FH，共有 48 个位置

可以把 SP 值设在 78H 吗？若程序很简单时是可以的，78H～7FH 只有 8 字节的空间，程序中若经过多次调用后，SP 值可能会到达 7FH，这是 AT89C2051 DATA MEMORY 的上限，再做一次调用后可能系统就死机了。所以 SP 值的设定是有点学问的，假使你写的程序看起来不可能出错，但是还是经常死机的话，就要怀疑 SP 值是否设得太大了。如果程序有数个中断服务程序在同时执行时，更要注意 SP 值了。

B 寄存器也是重要的寄存器之一，主要支持 8051 单片机做乘与除的运算。8051 的乘法指令为 MUL AB，相乘后 16 位的结果放在 B 与 A 寄存器上。8051 的除法指令为 DIV AB，A 为相除后的商，B 为相除后的余数值。当然，B 寄存器也可以做一般寄存器使用。

6.9　AT89C2051 的存储器配置

在介绍真正 8051 指令前，我们来了解一下 AT89C0051 的存储器配置情况。AT89C2051

共有 2KB 的程序存储（Program Memory）空间，这段区域是纯粹放置程序的。另外有 256 字节的数据存储（Data Memory）空间，用来存放随时可修改的数据。

图 6-6 是存储器的配置图，Data Memory 上半段称为特殊功能寄存器（Special Fuction Register），简称 SFR。SFR 内有 128 字节的空间，但只有其中 9 个地址可用。在 Data Memory 的下半部中，最前面一段为 R0～R7 寄存器占用，共占有四个 BANK，每个 BANK 占 8 字节，由程序指定存取哪个 BANK 内的值，BANK 上面密集区（20H～2FH）是可以单一位寻址的，请参考图 6-8。所谓单一位寻址是指这里的每一个单一位都可以用程序将其设成 1 或 0。

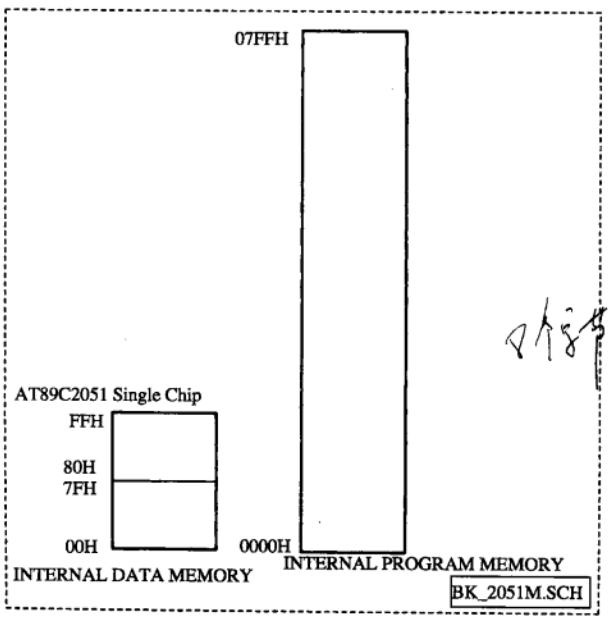


图 6-6　AT89C2051 的存储器配置图，DATA MEMORY 的上半部分是 SFR

学习 8051 单片机的工程师我们一般都称为固件工程师（Firmware Engineer），代表要同时横跨软件与硬件两大领域。固件工程师在硬件方面当然要知道 AT89C2051 的 P1 和 P3 端口是如何使用的，同时也要详细分析每个输入/输出端口的电气特性与驱动能力。另一方面也要有能力写一些测试程序来验证外部的硬件电路是否正确，所以还是要理解 AT89C2051 内部整个存储器的配置状况。

在软件方面除了要知道寄存器的正确位置外，还要知道程序长度的最大值。以 AT89C2051 为例，程序的最大值为 07FFH，即十进制的 2048 字节，当我们在使用 PGM2051 进行烧录时，通常会注意一下 TSK 文件的程序长度，若显示的长度值快超过 2000 字节时就要改用程序容量较大的 AT89C4051 来进行烧录了。

AT89C2051 提供了 5 个程序的断点，当程序被中断时，PC 值会先被填上一个特定地址值，然后由此开始中断服务程序，这些特定地址我们称为中断进入点（Entry Point），这些数据都会在 AT89C2051 的程序区配置图上标示，代表图 6-7 和图 6-8 都是 AT89C2051 相当重要的图。

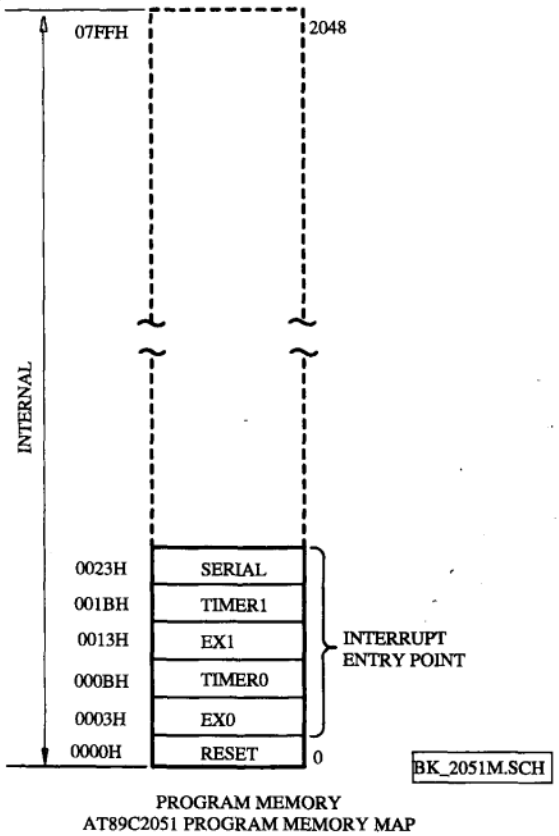


图 6-7　AT89C2051 PROGRAM MEMORY RESET 与五个中断的进入点

在图 6-8 里有两块区域是可单一位寻址的，分别是 DATA MEMORY 的 20H～2FH 与 SFR 上的所有寄存器。以 20H 为例，20H 上共有 8 位，而 20H～2FH 共有 16 字节，所以总共有 128 位可以分别寻址。在图 6-8 的 SFR 上，上面每个寄存器内的任何一个位也都可以用程序将其设成 1 或 0。

在 ASM51 程序中，我们可以直接写 SETB 20H.0，这是把（20H.0）这个位设成 1，而 CLR ACC.7 就是将 ACC 的位 7 清除为 0。

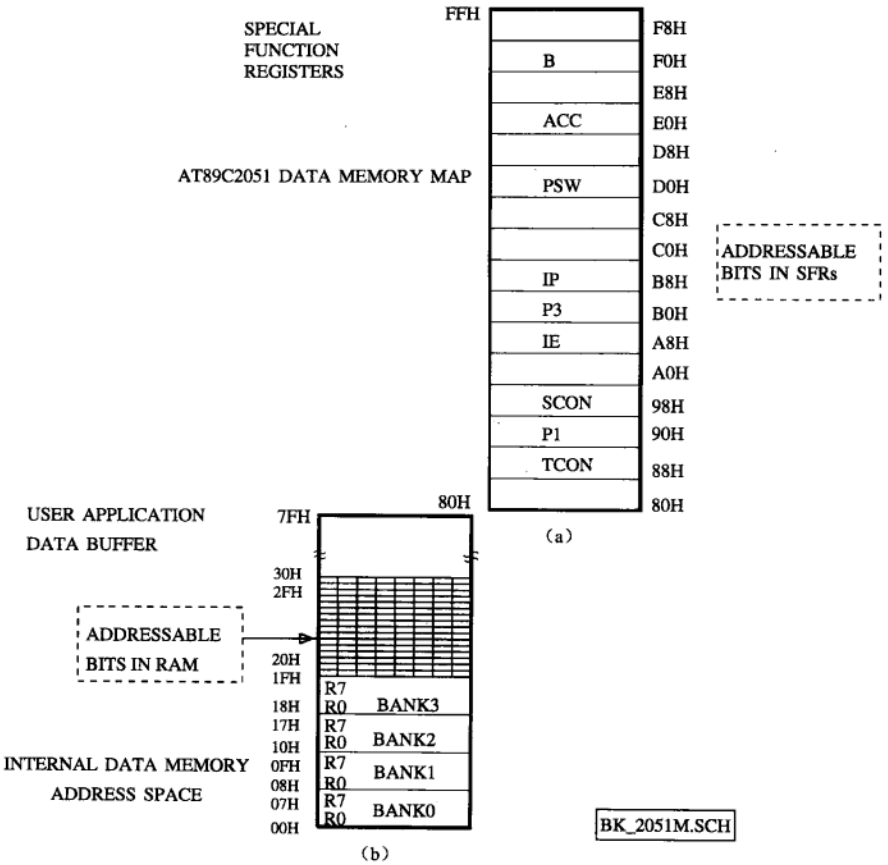


图 6-8　AT89C2051 的 DATA MEMORY，SFR 与 20H～2FH 这段空间是可以做单一位寻址的

您可以从下列公司的网站取得更进一步的信息：

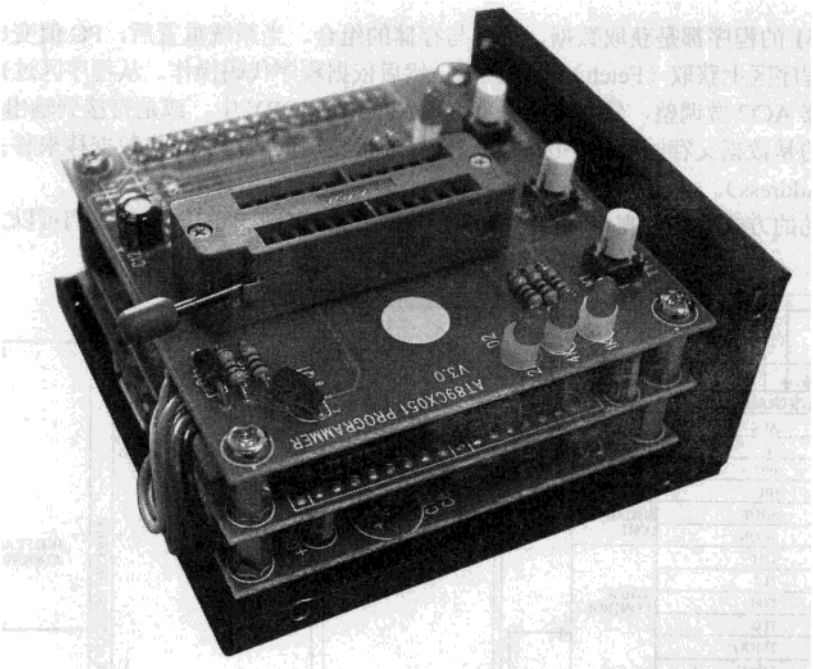
（1）www.chipware.com.tw: 查询单片机相关应用的参考资料。

（2）www.flag.com.tw: 查询“8051 单片机彻底研究”系列丛书的相关介绍。

（3）www.atmel.com: 制造 8051 的厂商，有很多 MCU 的相关参考资料。

（4）www.intel.com: 查询 8051（MCS51）的相关介绍。

7 8051的寻址方式



单片机8051共有五种寻址方式，也许有人会问为什么要学习寻址？如果我们把8051的工作分成三个部分：取得数据（Input）、处理数据（Process）和存储结果（Save），那么8051的程序说穿了就是这三种过程的组合。而其中取得数据与存储结果的操作，就直接和寻址有关。

第 7 章　8051 的寻址方式

7.1　寻址的种类

所有 8051 的程序都是获取数据、处理与存储的组合。当系统重置后，PC 值变成 0000H 后，开始从程序区上获取（Fetch）程序代码，然后依据程序代码操作。从程序区过来的指令会引导累加器 ACC 做调整，然后将结果存在 DATA MEMORY 中，或是直接到输出端口上。唯一不可能的是数据又存回到程序区中，因为程序区是只读的，这些将数据移来移去的过程就是寻址（Address）。

数据移动的方式有几种呢？请参考图 7-1，这是 AT89C2051 的框图，我们可以大略归纳出以下两种：

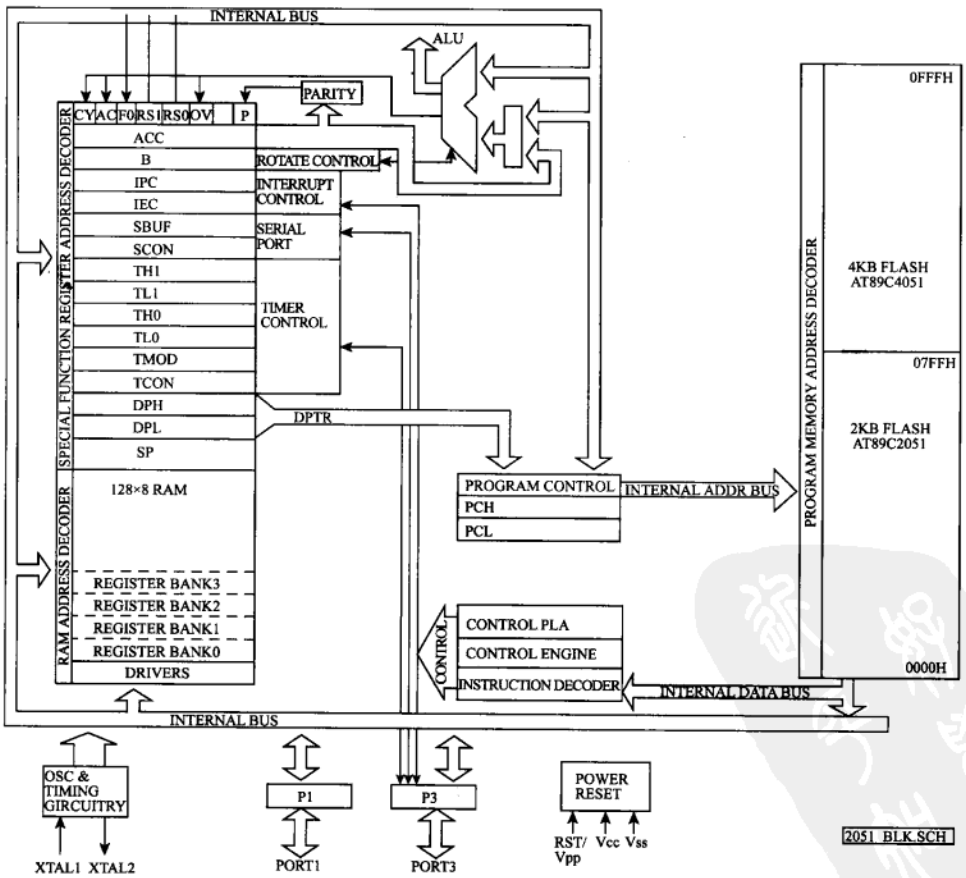


图 7-1　AT89C2051 的框图，在这章里应该注意的是数据的流向

（1）寄存器值直接得自程序区。
（2）寄存器值取自其他寄存器。
单片机又把上述两种情况细分成五种寻址方式：
（1）立即寻址方式，直接设定寄存器的值，如 MOV A,#58H
（2）直接寻址方式，直接指定地址后取值，如 MOV A,30H
（3）间接寻址方式，间接的方式取得 Data Memory 的值，如 MOV A,@R0
（4）寄存器寻址方式，寄存器间值的转换，如 MOV B,R0
（5）变址寻址方式，利用 PC 或 DPTR 当变址取值，如 MOVC A,@A+DPTR

7.2 立即寻址方式

立即寻址方式，我们在以前的示范程序中就常使用了。立即寻址可以将某一值直接传到寄存器或 Data Memory 上，立即寻址方式一定有一个井字号“#”作前辍。

```
MOV     A,#20H      ; 将累加器内容填成 20H
MOV     R6,#085H    ; 将 R6 寄存器内容填成 B5H
MOV     P1,#0FEH    ; 将 P1 端口内容填成 FEH
MOV     34H,#15H    ; 将 34H 地址内容填成 15H
```

上面这几种写法，都是立即寻址方式的写法，立即寻址的数据前面都有一个“#”，我们能很明显地看到被送进去的内容，而且当中几种用法，我们确实曾经使用过。现在如果用立即寻址方式来完成一个程序应该会顺利很多。

ASM51 编译程序在立即寻址上，对于设定值有许多写法可参考，这些写法都是很好的范例：

```
HIGH    EQU     50
LOW     EQU     2
        MOV     A,#30H          ;十六进制的表示
        MOV     A,#48           ;十进制表示
        MOV     A,#00110000B.   ;二进制的表示
        MOV     A, #'O'         ;ASCII 0 的表示
        MOV     A,#(HIGH-LOW)   ;编译时就相减
```

立即寻址方式其实不只 MOV 指令可以用，其他如加法 ADD 指令或逻辑运算指令都可以使用。

```
LIMITH  EQU     20
LIMITL  EQU     10
        ADD     A,#28H          ;A=A+28H
        ADDC    A,#''           ;A=A+20H+CY
        ORL     A,#33H          ;A=A OR 33H
        ANL     A,#FEH          ;A=A AND FEH
        ANL     A,#11111110B    ;A=A AND FEH
        XRL     A,#30H          ;A=A XOR 30H
        ADD     A,#(UMITH-LIMITL)
```

立即寻址方式和我们接下来要提的直接寻址方式很相似，两者写法只差一个“#”号，但其意义是完全不同的，初学者经常会搞混，请一定要注意。

```
MOV     A,#30H          ;A 值等于 30H
MOV     A,30H           ;A=(30H)，把 DATA MEMORY 30H 的内容取出并存入 ACC 中
```

7.3 直接寻址方式

直接寻址方式主要用在 8051 内部 DATA MEMORY 间的数据交换，当然也包括 SFR 间的数据交换，如果将寄存器的值送到输出端口上，也是属于此种寻址方式。

```
BUF1 EQU        20H
     MOV        A,30H        ;将 30H 地址内的值送到累加器
     MOV        P1,A         ;将累加器的值送到 P1 端口
     MOV        B,BUF1       ;B=(20H)
     MOV        A,SBUF       ;A=(SBUF)
     ADD        A,30H        ;A=A+(30H)，取 30H 的内容
     ADDC       A,BUF1       ;A=A+(20H)+CY，取 BUF1 的内容
     ORL        A,32H        ;A=A OR(32H)，取 32H 的内容做 OR 处理
     XRL        A,50H        ;A=A XOR(50H)
错误范例：
     ADD        28H,BUF1     ;(28H)=(28H)+(20H)
                             ;8051 没有这种指令
```

在 8051 中，只要是算术和逻辑运算，一定要在累加器上处理，运算后的结果也一定放在累加器上。所以写法要改成：

```
正确的写法：
     MOV        A,28H        ;A: (28H)
     ADD        A,BUF1       ;A=A+(BUF1)
     MOV        28H,A        ;(28H)=A
```

7.4 间接寻址方式

间接寻址是先通过寄存器取得指定地址，然后再取得该地址的值，在 8051 中只有 R0 和 R1 可以做间接寻址。间接寻址的指令中一定有“@”符号，且间接寻址只能在 DATA MEMORY 内使用。请先看看以下的例子：

```
     MOV        R1,#8H
     MOV        A,@R1            ;A=(28H)
     ADD        A,@R1            ;A=A+(28H)
     ADDC       A,@R1            ;A=A+(28H)+CY
     ORL        A,@R1            ;A=A OR(28H)
     ANL        A,@R1            ;A=A AND(28H)
     ADD        A,@R7            ;错误，R7 无法间接寻址
```

请比较下面两种写法：

```
BUFFER  DATA    20H
        MOV     R0,#BUFFER       ;R0=BUFFER 采用间接寻址法
        MOV     A,@R0            ;A=(R0)，取 R0 所指地址的值
        MOV     P1,A             ;P1=A

        MOV     A,BUFFER         ;A=(BUFFER)，直接寻址法
        MOV     P1,A             ;P1=A
```

两种写法虽然不同，但最后的结果都一样。不过间接寻址还是有其使用的场合，当要取得一连串数据进行运算时，间接寻址方式就会显得比较合适了。

7.5 寄存器寻址方式

寄存器寻址就是指寄存器之间的数据移动，累加器 ACC、Rn 寄存器(n=0～7)、P1 端口 P3 端口间的数据移动都可以用寄存器寻址方式，8051 经常用此方式来暂时保存计算值。选用寄存器寻址方式的另一个好处是可以节省程序代码所占有的空间。

【例 7-1】7-4-1.ASM

```
$MOD51
    MOV         A,R0        ;A=R0
    ADD         A,R7        ;A=A+R7
    MOV         P1,A        ;P1=A
    MOV         B,A         ;B=A
    MOV         SP,A        ;SP=A
    MOV         PSW,A       ;PSW=A
```

上列各行指令所产生的代码，前两行只有 1 字节，如果改用下面的写法则全部都要 2 字节。

```
    MOV         A,00H       ;A=(00H)
    ADD         A,07H       ;A=A+(07H)
    MOV         90H,A       ;(90H)=A
    MOV         0F0H,A      ;(F0H),A
    MOV         81H,A       ;(81H)=A
    MOV         0D0H,A      ;(D0H)=A
    END
```

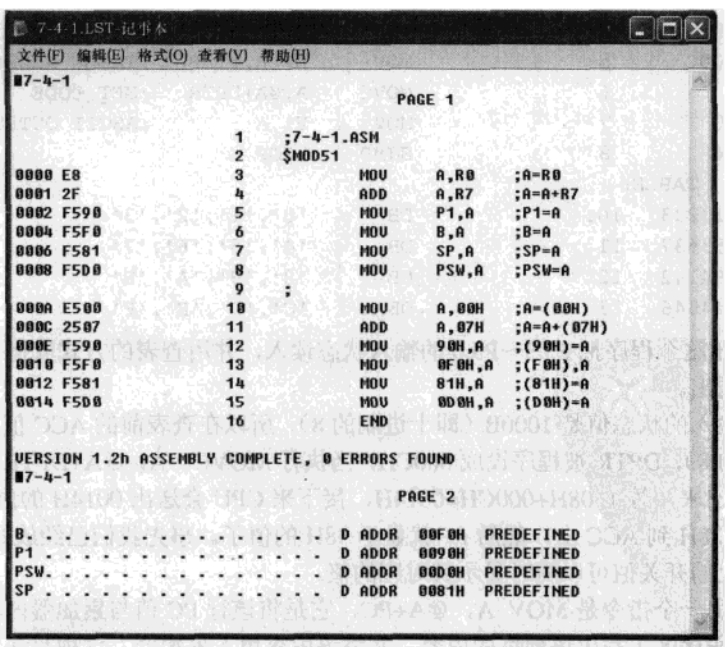


图 7-2 寄存器寻址与直接寻址的比较，很明显寄存器寻址会占较少的程序空间

7.6 变址寻址方式

变址寻址方式比较特殊，它通过 DPTR 或 PC 值去取得程序区内的数据，说简单一点就是查表取值。查表取到的值此时是放在 ACC 累加器上，我们先看一下例子：

【例 7-2】7-5-1.ASM

```
$MOD51
LOOP:   MOV         A,P3            ;RAED FROM P3
        ANL         A,#0FH          ;GET bit3~bit0 ONLY
        MOV         DPTR,#TABLE     ;TABLE ADDRESS
        MOVC        A,@A+DPTR       ;GET CODE
        MOV         P1,A            ;ASCII OUTPUT
        SJMP        LOOP
TABLE:
        DB          '0','1','2','3'
        DB          '4','5','6','7'
        DB          '8','9','A','B'
        DB          'C','D','E','F'
        END
```

以下是用 ASM51 转出来的 7-5-1.LST 文件：

```
                    1       ;7-5-1.ASM
                    2       $MOD51
0000 E5B0           3       LOOP:   MOV     A,P3          ;RAED FROM P3
0002 540F           4               ANL     A,#0FH        ;GET bit3~bit0 ONLY
0004 90000C         5               MOV     DPTR,#TABLE   ;TABLE ADDRESS
0007 93             6               MOVC    A,@A+DPTR     ;GET CODE
0008 F590           7               MOV     P1,A          ;ASCII OUTPUT
000A 80F4           8               SJMP    LOOP
000C    9   TABLE:
000C 30313233      10               DB      '0','1','2','3'
0010 34353637      11               DB      '4','5','6','7'
0014 38394142      12               DB      '8','9','A','B'
0018 43444546      13               DB      'C','D','E','F'
```

7-5-1.ASM 这个程序把 P3.3～P3.0 的输入状态读入，并用查表的方式取得 ASCII 值，然后从 P1 端口送出。

假设 P3 输入的状态值是 1000B（即十进制的 8），所以在查表前的 ACC 值是 00001000B（也是十进制的 8），DPTR 被程序设成 000CH，当执行 MOVC　A，@A+DPTR 时，A+DPTR 的值先被计算出来，等于 08H+000CH=0014H，接下来 CPU 会送出 0014H 的值到程序区中，并读回一个值 38H 到 ACC 上，然后 P1 就显示 38H 的值了。事先我们已经放好 16 个 ASCII 值，所以 P3 上的开关值可以随时显示其对应的值。

变址寻址另一个指令是 MOV A，@A+PC，它是将该行 PC 值与累加器内容相加后的值当做地址，到程序区上找出该地址的内容，并将该内容填入累加器，这种写法不多见，不过当初设计 8051 的 Intel 应该是为了某种需要，才把这个指令加入的。

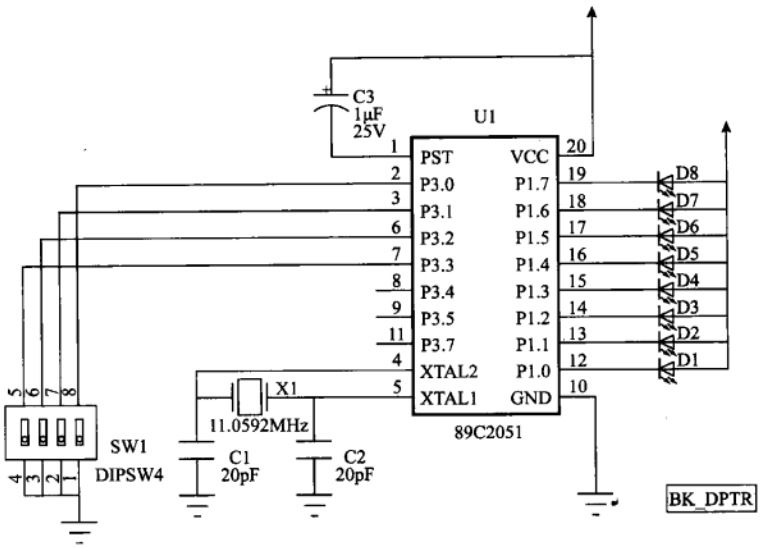


图 7-3　DPTR 查表的示范电路图

7.7　寻址方式实例

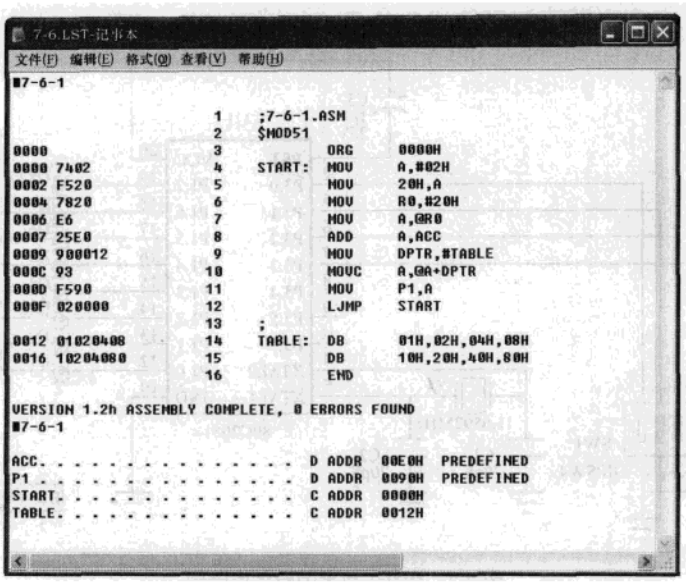
以下是一个结合所有寻址方式的程序范例，我们要通过工作框图，实际了解寻址操作在 CPU 中是怎么进行的。先来看看程序的内容：

【例 7-3】 7-6-1.ASM

```
$MOD51
            ORG         0000H
START:      MOV         A,#02H          ;立即寻址
            MOV         20H,A           ;直接寻址
            MOV         R0,#20H         ;立即寻址
            MOV         A,@R0           ;间接寻址
            ADD         A,ACC
            MOV         DRTR,#TABLE     ;变址寻址
            MOVC        A,@A+DPTR
            MOV         P1,A            ;直接寻址
            LJMP        START
;
TABLE:      DB          01H,02H,04H,08H
            DB          10H,20H,40H,80H
            END
```

程序码的内容是一些寻址操作的综合，经过 ASM51 编译后，我们可以看到 .LST 文件中把这些程序码转换成机器码及其相对应的程序地址，接下来我们就一步一步来看看程序是如何操作的。

在 7-4 的 7-6-1.LST 文件中，我们可以看到各种寻址所需要的程序空间大小，及对应的程序存储器地址。



```
7-6.LST-记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
■7-6-1

                        1    ;7-6-1.ASM
                        2    $MOD51
0000                    3            ORG     0000H
0000 7402               4    START:  MOV     A,#02H
0002 F520               5            MOV     20H,A
0004 7820               6            MOV     R0,#20H
0006 E6                 7            MOV     A,@R0
0007 25E0               8            ADD     A,ACC
0009 900012             9            MOV     DPTR,#TABLE
000C 93                10            MOVC    A,@A+DPTR
000D F590              11            MOV     P1,A
000F 020000            12            LJMP    START
                       13    ;
0012 01020408          14    TABLE:  DB      01H,02H,04H,08H
0016 10204080          15            DB      10H,20H,40H,80H
                       16            END

VERSION 1.2h ASSEMBLY COMPLETE, 0 ERRORS FOUND
■7-6-1

ACC. . . . . D ADDR  00E0H  PREDEFINED
P1 . . . . . D ADDR  0090H  PREDEFINED
START. . . . . C ADDR  0000H
TABLE. . . . . C ADDR  0012H
```

图 7-4　7-6-1.LST 的内容

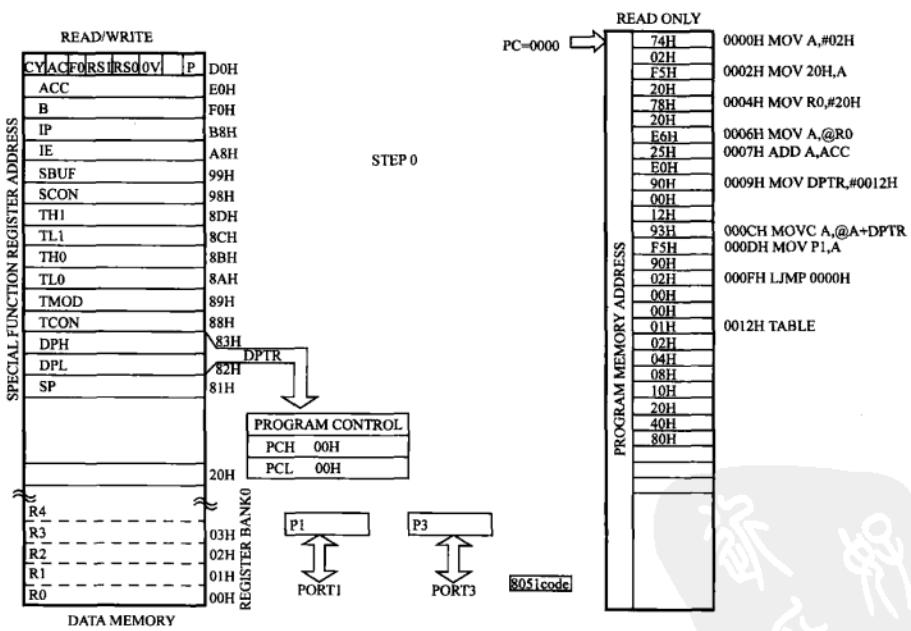


图 7-5　系统重置后的起始状态

图 7-5 中，当系统 RESET 之后，PC 值会从 0000H 开始，此时所有寄存器中的内容，是系统的默认值。再看到右边 PROGRAM MEMORY ADDRESS 部分，这是程序转换成机器码之后摆放在程序存储区的对应位置，程序第一个所要执行的，就是 MOV　A，#02H 这个立即寻址的操作，对应的机器码为 74H 02H，一共占了两个字节的程序空间。

图 7-6 中，因为程序第一个所要执行的操作是 MOV　A，#02H，所以经 CPU 判断为两个字节的指令，因此 PC 值在读入指令后，会自动加 2 到 0002H 的地址上，再按指令把 02H 的值填入 ACC 累加器中，此时 ACC 累加器的内容会变成 02H，而这个操作，就是一个标准的立即寻址操作。

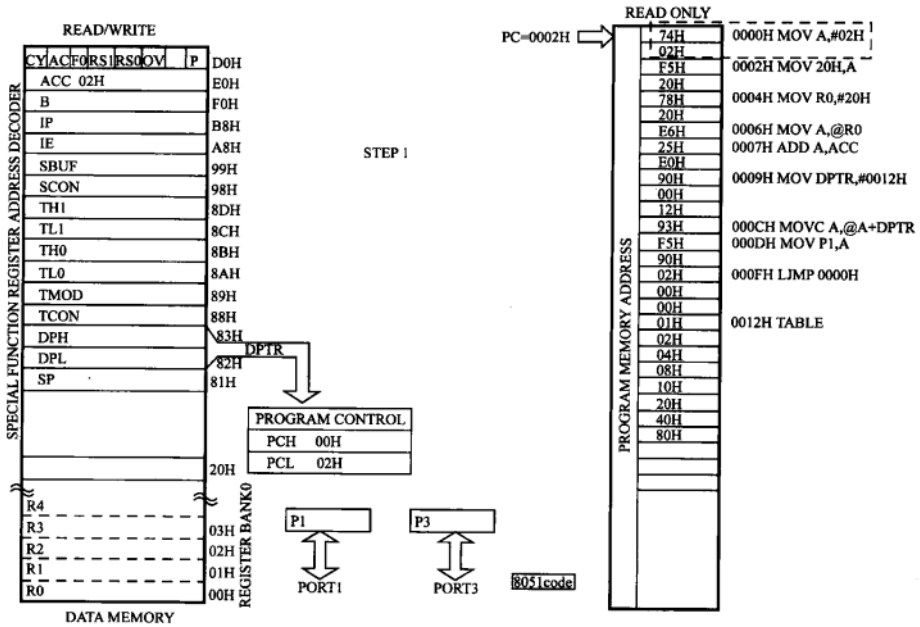


图 7-6　系统开始执行第一个程序操作

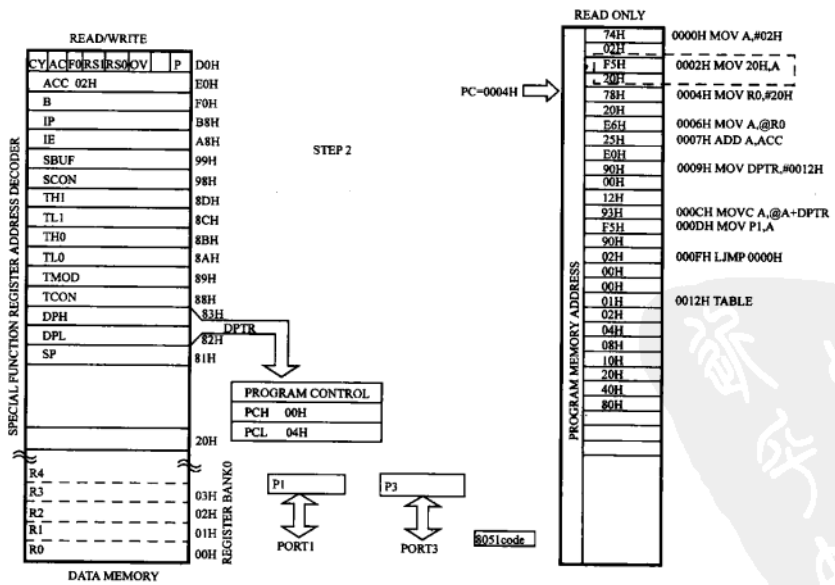


图 7-7　直接寻址方式工作框图

图 7-7 中，完成了第一个指令后，系统继续读入下一个指令，经判断 MOV 20H，A 为两个字节的指令，所以 PC 值又自动加 2，来到了 0004H 的地址上，此时的操作是将 ACC 累加器里的数值，复制到 20H 这个 DATA MEMORY 上，因此我们在框图的左边，发现 ACC 累加器与 20H 这两个方格里都出现了 02H 的数值，这个操作就是直接寻址的标准方式。与立即寻址最大的不同，除了程序上多了一个“#”外，在操作上则是把数据从来源端（本例为 ACC 累加器）复制到目的端（20H），并不是直接在 ACC 累加器上填入 20H 的数值，使用时千万不要混淆了。

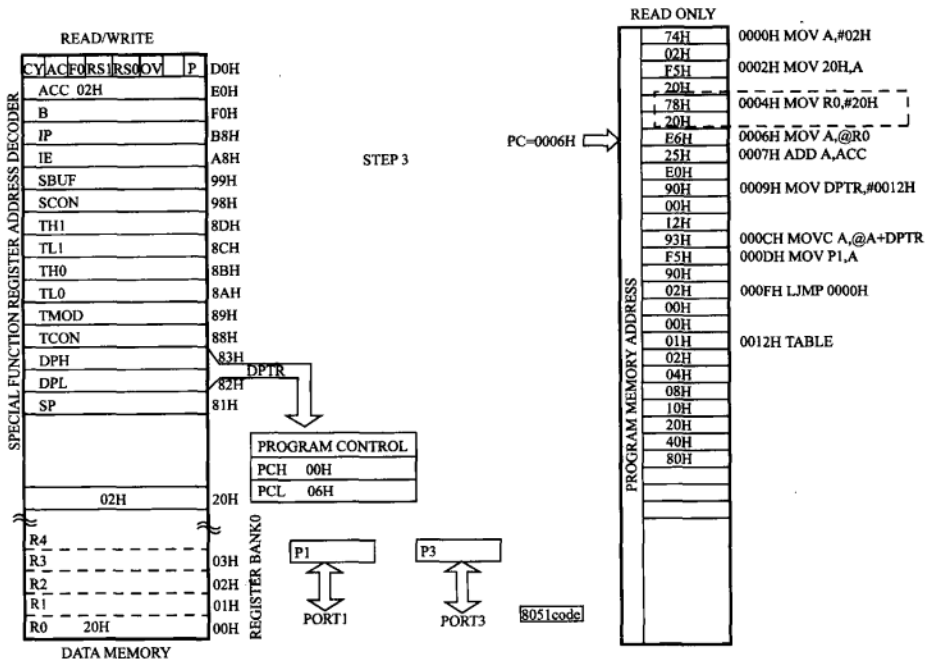


图 7-8　间接寻址的前置工作

图 7-8 中，由于 8051 中只有 R0 和 R1 两个寄存器可以进行间接寻址，因此当我们需要使用间接寻址前，必须指定一个间接的地址给 R0 或 R1，接下来才能进行间接寻址，在本例中我们填入 R0 内的间接地址是 20H，在左边最下方的 R0 方格里，我们可以看到 R0 里的内容为 20H，而这个操作是立即寻址方式。

图 7-9 中，若使用间接寻址，在程序中一定会有“@”符号。MOV　A，@R0 这个操作，会先在 R0 中获取（20H）的值，把它当作是复制数值的来源端，然后把（20H）里面的数值读出来后，再复制到 ACC 累加器中，所以我们可以看到 ACC 累加器里面的值还是 02H，并不是 20H，这是因为（20H）在这里是来源端地址，并不是我们想要复制的数值。若将这一行程序改写为直接寻址方式，就是 MOV A,20H。

图 7-10 中，使用变址寻址，要通过 ACC 累加器来指定查表的地址，因此 ADD　A，ACC 这一行程序，只是想要改变 ACC 累加器的值，作为查表时要选择哪一数据的依据。经过这个操作后，ACC 累加器的内容变成了 04H，也就是我们等一下查表时要找出表格里的第四笔数据。

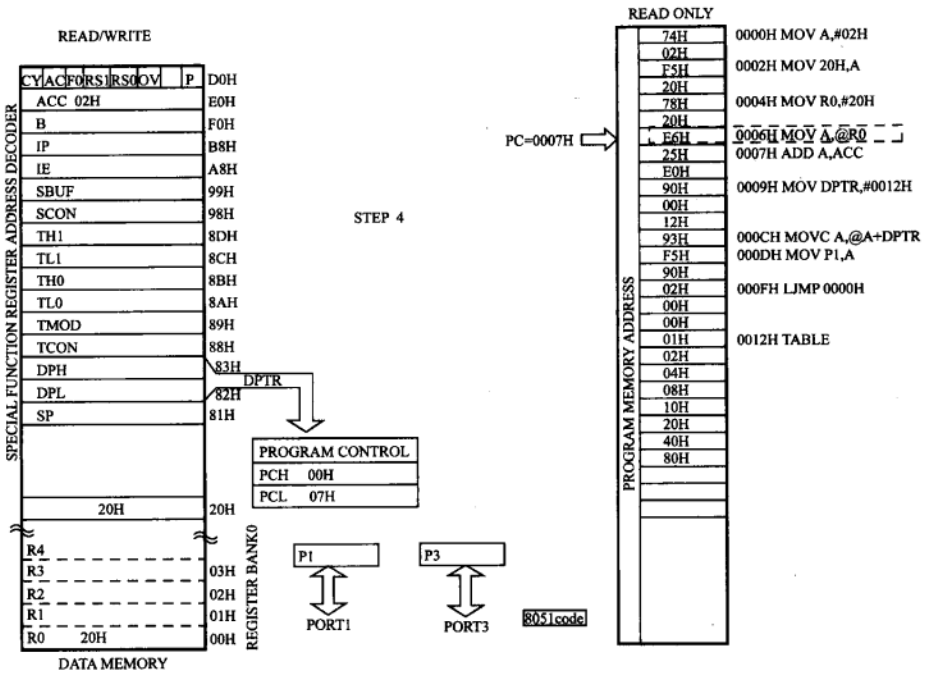


图 7-9　间接寻址的工作框图

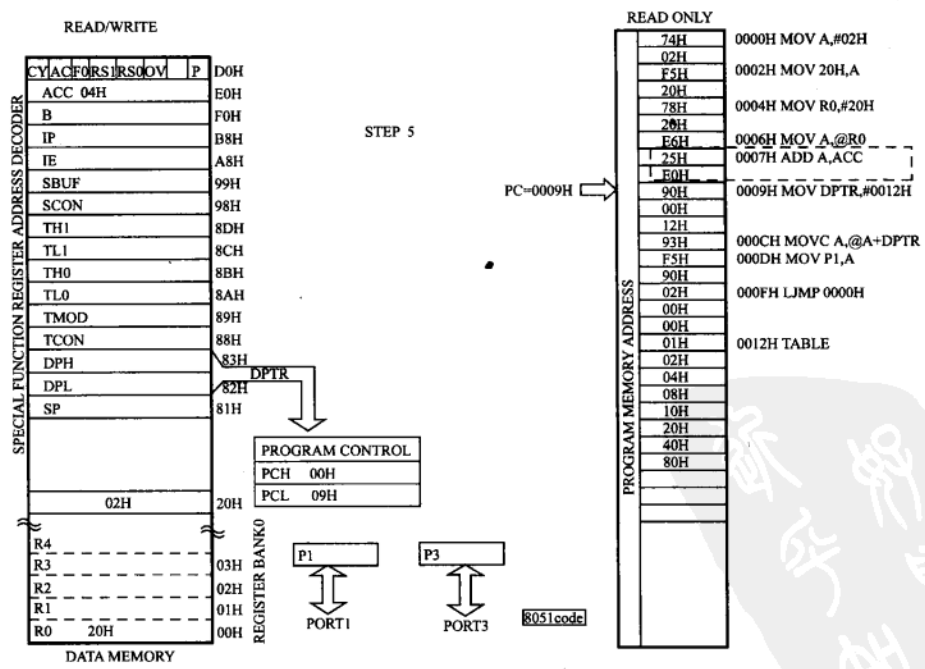


图 7-10　变址寻址的前置工作 1

图 7-11 中，变址寻址除了指定要找第几笔数据外，还要指定表格起点，MOV DPTR, #TABLE 就是告诉 CPU 表格的起点在哪里。DPTR 是一个 16 位的寄存器，分别由 DPH 与 DPL 两个 8 位寄存器所组成，当我们使用了 MOV DPTR, #TABLE 的指令，ASM51 会自动将 TABLE 所对应的 16 位地址计算出来，并依高低字节将数值填到 DPTR 的 DPH 和 DPL 中。

图 7-12 中，当变址寻址开始操作时，CPU 会先到 DPTR 把查表的起点复制到 PROGRAM CONTROL（PC）的 PCH 和 PCL 寄存器，这两个寄存器是提供给系统用的，没办法用程序直接去控制或存取。接着 CPU 会把 ACC 累加器里的数值拿来跟 DPTR 的值相加，作为读取数据的起始地址，经运算所得到的结果是 0016H，所以 CPU 就会到程序区的 0016H 位置上，把数值抓取到 ACC 累加器。以本例来说，(0016H)上的数值为 10H，因此 CPU 会把 10H 的值复制到 ACC 累加器上。

图 7-13 中，如果我们要把得到的结果输出到硬件电路上来观察，只要通过寻址方式就可以轻松做到，以本例说明：我们利用直接寻址方式将 ACC 累加器的数值 10H 复制到 P1 的寄存器上，此刻 CPU 的硬件电路会自动将 10H 由 P1 端口输出。

图 7-14 中，由于程序的规划，是将所有的程序操作执行完成后，再回到 0000H 重新执行。这个操作我们称为软件重置，它是强迫系统回到程序最开始的起点，用来避开某些程序中不稳定的无穷循环。不过，软件重置要留意的重点是：所有寄存器的内容并不一定是系统的默认值。因此，当程序中有需要做软件重置操作时，别忘了将寄存器一并填成系统的默认值了。

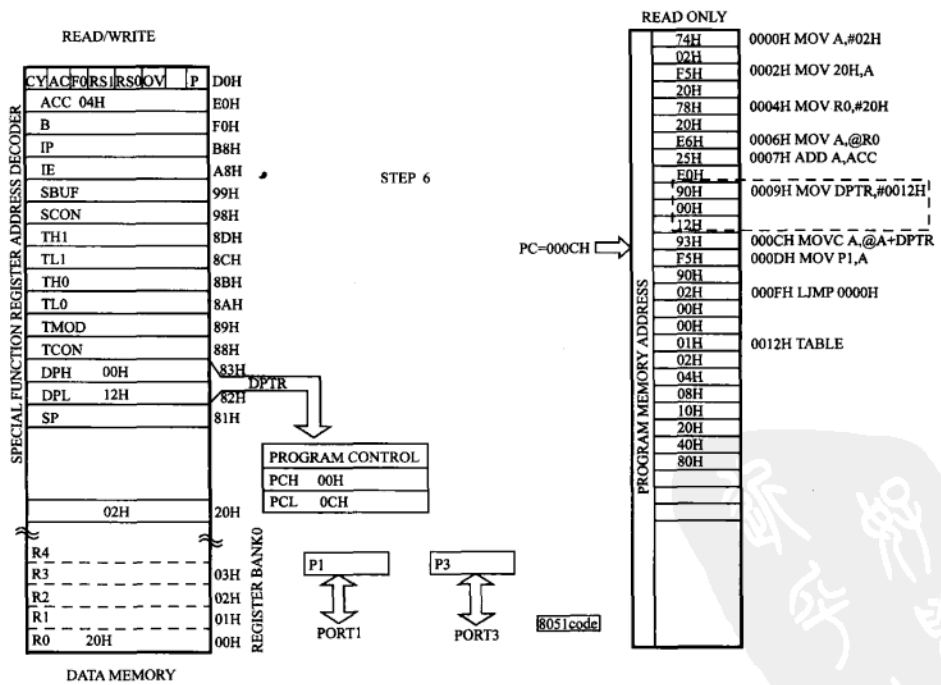


图 7-11　变址寻址的前置工作 2

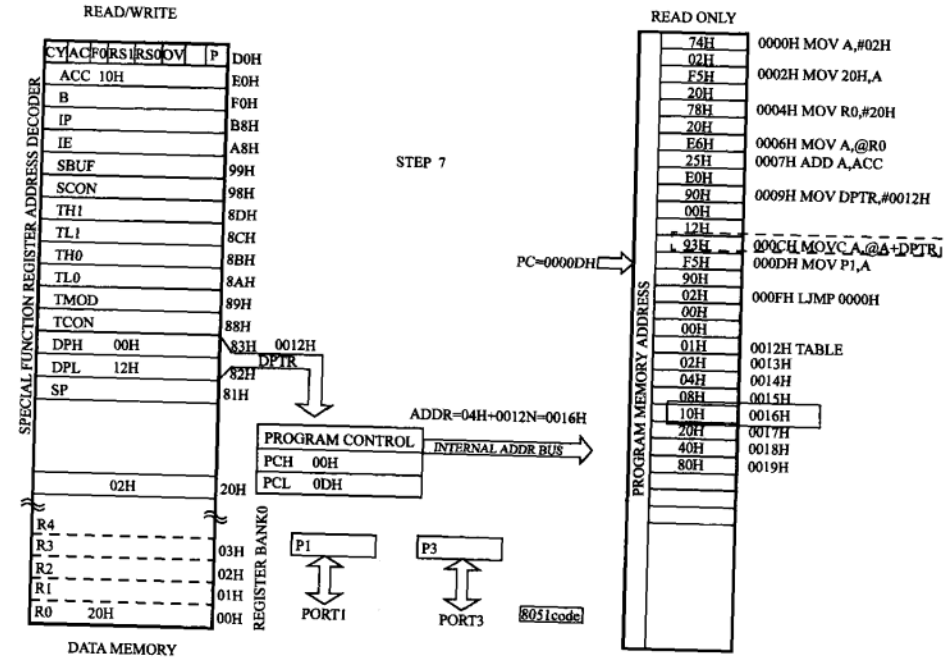


图 7-12　变址寻址的工作框图

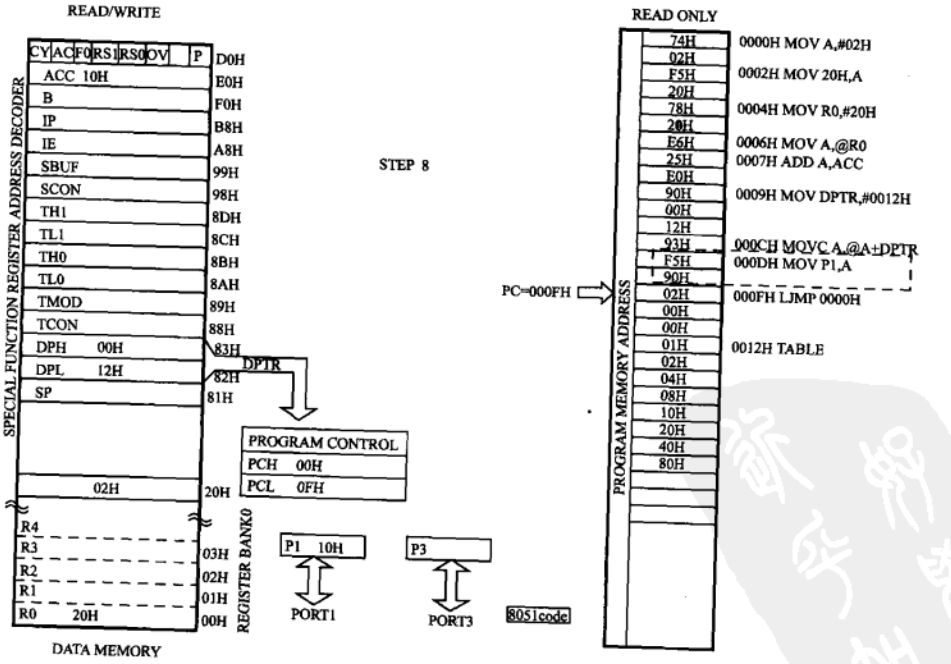


图 7-13　用寻址方式可以进行输入/输出操作

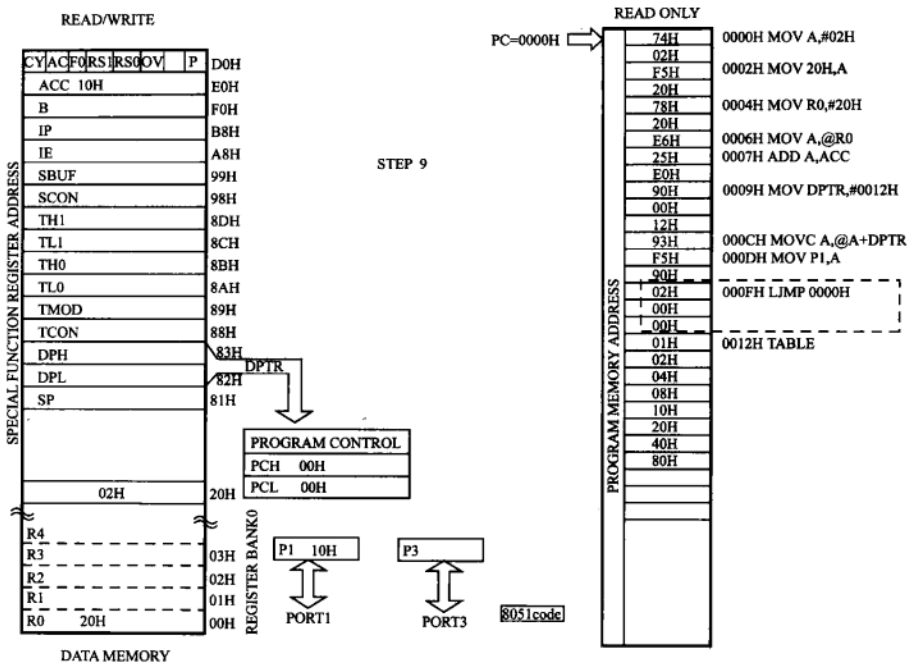


图 7-14　利用程序做软件重置操作

您可以由下列公司的网站取得更进一步的信息：

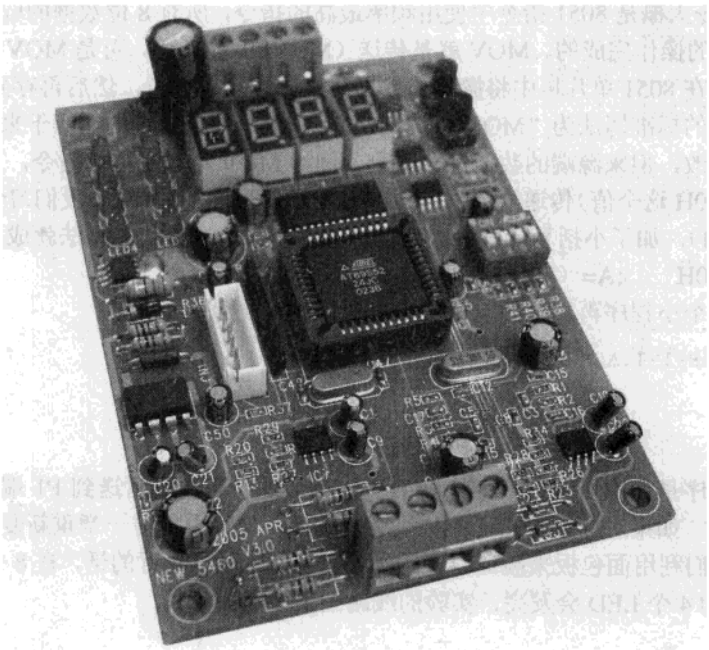
（1）www.chipware.com.tw：查询单片机相关应用的参考资料。

（2）www.flag.com.tw：查询“8051 单片机彻底研究”系列丛书的相关介绍。

（3）www.atmel.com：制造 8051 的厂商，有很多 MCU 的相关参考资料。

（4）www.intel.tom：查询 8051（MCS51）的相关介绍。

8 8051的指令系统



了解了8051的系统结构寻址方式后，接下来将碰到学习8051最艰难的课题：8051的指令系统。为了使DIY的您更容易上手，我们特将本章改成Learning and Doing边做边学的学习方式。先做一个小实验，再把实验所得到的结果，反推过来学习8051指令系统。实验的电路很简单，您自己使用面包板接线的话，不用5min就可以完成了，接着只要准备一台+5V的电源供应器和一台数字电表就可以开始了。

第 8 章　8051 的指令系统

8.1　传送指令 MOV

MOV 指令大概是 8051 指令中使用频率最高的指令，所有 8 位数据的取得与存储都是通过 MOV 指令的操作完成的。MOV 就是传送（MOVE）的意思，可是 MOV 不是真的把东西传过去，而是在 8051 单片机中将搬移的数据从来源端复制一份，然后再存到目的端上。

传送指令的标准写法为“MOV 目的端，来源端”，即目的端的值等于来源端，目的端的值一定会被更改，但来源端的数据不会改变。例如 MOV A,30H 这条指令，是把 30H 内部的 8 位数据（不 30H 这个值）传递给 ACC 累加器，在程序后面的注释上，我们习惯写成 A=(30H) 或 A<—（30H），加了小括号代表读取该位址的内容。所以正式的写法就成为：

MOV A,30H　　;A=（30H）

现在用一个小程序再加上面包板实验来认识 MOV 指令：

【例 8-1】 8-1-1.ASM

```
$MOD51                          ;声明芯片使用 51 规格
    MOV       P1, #0FH          ;将 0FH 值送到 P1 端口
    END                         ;程序结束
```

这段小程序是使用 MOV 指令将 0FH（#data）这个数复制后送到 P1 端口，这是一个典型的立即寻址。如果对寻址方式还不是很清楚的话，建议您把前一章重新复习一遍。程序烧录完成后，我们利用面包板来验证程序的操作。若程序顺利执行的话，在 8 个 LED 中，代表 P1.7 到 P1.4 的 4 个 LED 会发亮，实验的线路图请参考图 8-1。

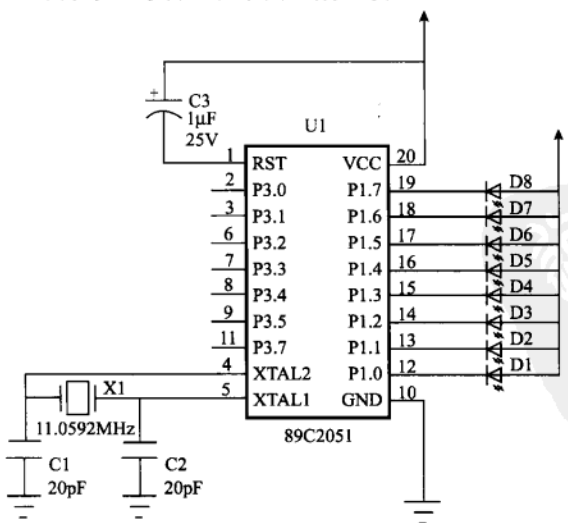


图 8-1　学习 8051 指令所用到的线路图，完成本节的实验后，只要更新程序就可以学习其他章节的指令

LED 的驱动是采用负逻辑的接法，这是让收到 0 的 LED 亮，收到 1 的 LED 不亮。而程序中送到 P1 端口的数值为 0FH，也就是 00001111B，所以应该有四个 LED 会亮，另外四个 LED 不会亮。

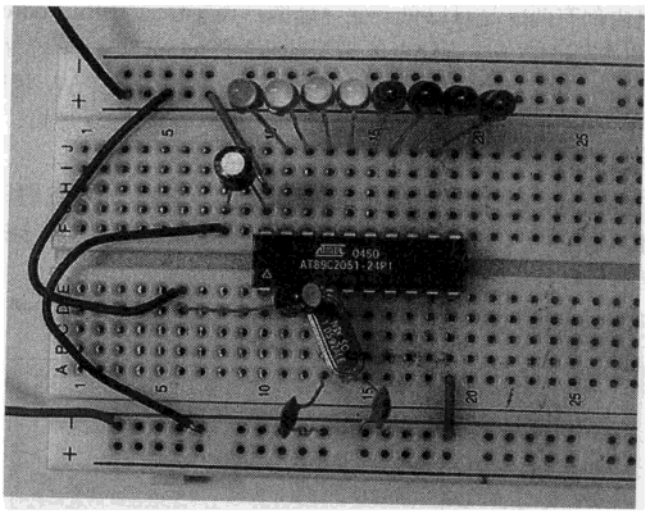


图 8-2　按照线路图先把烧录好的芯片放上，再把周边元件配置好，最后再接上+5V 电源，并观察 LED 显示的状态

同样的操作我们也可以用下面的程序来表示：

【例 8-2】 8-1-2.ASM

```
$MOD51
    MOV         A, #0FH       ;将 0FH 值送到累加器，立即寻址
    MOV         P1, A         ;将累加器值送到 P1 端口，寄存器寻址
    END
```

这次我们先使用 MOV 指令将 0FH 值指定给累加器 ACC 后，一样用 MOV 指令将 A 的值送到 P1 端口，即 P1=A。我们将程序烧入 AT89C2051 单片机后放在面包板上，LED 一样会亮四颗，而且会和前一个程序的结果相同。我们再看下面另一种写法：

【例 8-3】 8-1-3.ASM

```
$MOD51
    MOV         20H, #0FH     ;(20H)=0FH            立即寻址
    MOV         90H, 20H      ;P1= (90H) = (20H)    直接寻址
    END
```

【例 8-4】 8-1-4.ASM

```
$MOD51
    MOV         R0,#20H       ;R0=20H               立即寻址
    MOV         @R0, #0FH     ;(20H)=0FH            间接寻址
    MOV         90H,@R0       ;P1=(90H)=(20)H       间接寻址
    END
```

8051 内部只有一个 16 位的指针称为 DPTR，我们也是用 MOV 指令来定义其值，写法为 MOV DPTR,#TABLE，这也是立即寻址的一种，一次指定了 16 位的值。

8.2 SETB 和 CLR 置定和清除指令

在 8051 单片机内部有两区是可以单一位寻址的，英文称之为 Bit Addressable，在该区域里的每一个位都可以分别设成 1 或清除为 0，请看图 8-3 的 AT89C2051 可位寻址图，除了 20H～2FH 共有 128 位可以单一位寻址外，还有寄存器 B、ACC 累加器、PSW 程序状态寄存器、IP 中断优先寄存器、IE 中断允许寄存器、SCON 串行控制寄存器、P3 端口、P1 端口与 TCON 定时控制寄存器等等也可以进行位寻址。

前一节我们仅用 MOV 一个指令控制 P1 端口上的八个 LED，如果我们只打算让一个 LED 亮或不亮，除了 MOV 指令外，SETB 和 CLR 两个位指令也是可以的。SETB 是将目前的状态置成 1，而 CLR 则是将目前的状态清为 0。

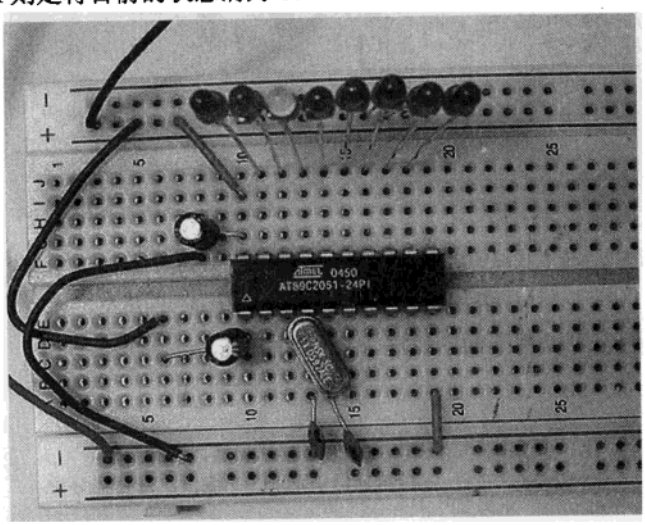


图 8-3　利用上一次的线路图，对 LED 采用负逻辑的接法，程序中只让 P1.5 引脚上的 LED 亮

下面的程序例，只让线路板的 P1.5 引脚的 LED 亮。

【例 8-5】 8-2-1.ASM

```
$MOD51
    ORG     0000H
    SETB    P1.7
    SETB    P1.6
    CLR     P1.5          ;将 P1.5 引脚的状态清为 0
    SETB    P1.4
    SETB    P1.3
    SETB    P1.2
    SETB    P1.1
    SETB    P1.0
    END
```

果然在面包板上的 8 个 LED 灯，只亮了 P1.5 引脚。这个程序总共做了 8 次指令操作，才把 P1 端口上的 8 个 LED 搞定。当 AT89C2051 重置时会自动将 P1 端口设成都是 1，即

11111111B，所以上面的程序可以修改精简成下一个程序。

【例 8-6】8-2-2.ASM

```
$MOD51
    ORG     0000H
    CLR     P1.5            ;将 P1.5 引脚的状态清为 0
    END
```

如果程序要更好的话，需要考虑到重置的操作若不确定时，所以要在程序最前面加上 P1 端口的起始值设定。

【例 8-7】8-2-3.ASM

```
$MOD51
    MOV     P1,#0FFH        ;P1 端口=FFH
    CLR     P1.5            ;把 P1.5 引脚状态清为 0
    END
```

8.3 加减 1 指令 INC 和 DEC

加 1 或减 1 的指令使用机会也比较多，在 8051 的指令集中也包括了 INC（加 1）与 DEC（减 1）的指令，以下是几种使用 INC 和 DEC 的场合。

```
INC A                   ;A=A+1
INC R0                  ;R0 值加 1
INC 20H                 ;(20H)=(20H)+1，内容值加 1
INC @R1                 ;R1 所指的地址内容值加 1
DEC A                   ;A=A-1
DEC R0                  ;R0=R0-1
DEC 20H                 ;(20H)=(20H)-1
DEC @R1                 ;R1 所指的地址内容值减 1
```

使用 INC 和 DEC 指令唯一要注意的是这两个指令并不影响 PSW 的状态值，所以在 INC 和 DEC 之后，是不可以做程序的条件判断的。

【例 8-8】8-3-1.ASM

```
$MOD51
    MOV     A,#0FH          ;累加器=0FH
    INC     A               ;A=A+1
    MOV     P1,@R0          ;P1=A=00010000B
    END
```

将这段程序码烧入后，LED 亮了七个，只有代表 P1.4 的 LED 不亮，这跟我们在注释上推导的结果相同。

接下来看另一个程序范例：

【例 8-9】8-3-2.ASM

```
$MOD51
    MOV     20H,#0FH        ;(20H)=0FH
    INC     20H             ;(20H)=(20H)+1
    MOV     P1,20H          ;P1=(20H)=00010000B
    END
```

以下程序是示范间接寻址的写法：

【例 8-10】 8-3-3.ASM

```
$MOD51
    MOV     20H,#0FH        ;(20H)=0FH
    MOV     R0,#20H         ;R0=20H
    INC     @R0             ;(20H)=(20H)+1
    MOV     P1,20H          ;P1=(20H)=00010000B.
    END
```

上面这两个例子的结果是一样的，只是程序写法上分别用了直接寻址和间接寻址的方法。在我们编写程序时，可以按应用环境的需求选用较适合的写法。

8.4 加法指令 ADD 和 ADDC

加法指令是 8051 单片机指令集中最常使用的算术指令。加法指令包括 ADD 和 ADDC。两者唯一的不同是后者把进位标志位 CY 也加上去。提到加法运算一定要提到 PSW 上的 CY 标志位，当两个字节数据做相加的操作时，一定会影响 CY 的状态值。若相加后超过 FFH 时，CY 值会被设成 1，反之 CY 会变为 0。8051 单片机指定加法运算一定要在 ACC 累加器上进行，结果也放在 ACC 上，就是这个缘故，处理相加的寄存器才叫做累加器（ACCUMULATOR）。加法指令会影响 CY 标志位、OV 标志位与 AC 标志位值。

PSW 各位的定义：

表 8-1 PSW 各位的定义

（D0H）

CY	AC	F0	RS1	RS0	OV	—	P
D7	D6	D5	D4	D3	D2	D1	D0

CY（PSW.7）：运算进位标志位

AC（PSW.6）：辅助运算标志位

OV（PSW.2）：运算溢位标志位

【例 8-11】 8-4-1.ASM

```
$MOD51
    ORG     0000H
    MOV     PSW,#000H       ;PSW=00H
    MOV     A,#0F2H         ;A=F2H
    ADD     A,#69H          ;A=A+69H=5BH CY=1
    MOV     A,PSW           ;A=PSW
    CPL     A               ;A=!A
    MOV     P1,A            ;P1=!PSW
END
```

上面的程序是把 F2H+69H=15BH，由于 ACC 只能存放 8 位的数据，所以 ACC 值成为 5BH，由于加法后有进位，CY 值被设成 1。我们打算把 PSW 的状态显示在 P1 端口上，但是人的习惯是 LED 亮代表状态 1，因此在程序当中先把 PSW 读到 ACC 上，执行反相后再送到 P1 端口上。执行后的结果是表示 CY（PSW.7）的 LED 亮了。

如果要把连续两笔 16 位数据加起来时，这时会进行两次加法，第一次并没有进位情形，

可以用 ADD 指令来做，相加后 CY 值一定会改变。而第二次相加时，就一定要用 ADDC 指令了，否则无法把进位值加进来。

【例 8-12】 8-4-2.ASM

```
;2 BYTE DATA ADDITION
$MOD51
    ORG         0000H
    MOV         R0,#20H
    MOV         R1,#30H
    MOV         A,@R0               ;A=(20H)
    ADD         A,@R1               ;A=(20H)+(30H)
    MOV         @R1,A               ;(30H)=(20H)+(30H)
    INC         R0                  ;R0=21H
    INC         R1                  ;R1=31H
    MOV         A,@R0               ;A=(21H)
第二次相加一定定用 ADDC 指令
    ADDC        A,@R1               ;A=(21H)+(31H)+CY
    MOV         @R1,A               ;(31H)=(21H)+(31H)+CY
    END
```

上面的程序用了 R0 和 R1 做间接寻址，把两个 16 位的数据加起来，而在相加之前数据要事先放在（21H）（20H）与（31H）（30H）上，结果放在地址 31H 和 30H 上。

8.5 减法指令 SUBB

8051 单片机上只有一个减法指令 SUBB，而且减法运算一定在 ACC 累加器上进行。SUBB 执行时，会先减 CY 进位标志位后，再减去指定位置的值，结果会留在 ACC 累加器上。SUBB 指令会影响 CY 标志位、OV 标志位与 AC 标志位值。来看看 SUBB 的使用范例，如表 8-2 所示。

表 8-2 PSW 各位的定义

（D0H）

CY	AC	F0	RS1	RS0	0V	—	P
D7	D6	D5	D4	D3	D2	D1	D0

CY（PSW.7）：运算进位标志位

AC（PSW.6）：辅助运算标志位

OV（PSW.2）：运算溢位标志位

【例 8-13】 8-5-1.ASM

```
$MOD51
    ORG         0000H
    CLR         CY                  ;CY=00H
    MOV         A,#0F2H             ;A=F2H
    SUBB        A,#69H              ;A=A-69H=89H CY=0
    MOV         A,PSW               ;A=PSW
    CPL         A                   ;A=!A
    MOV         P1,A                ;P1=!PSW
    END
```

上面的程序是把 F2H-69H=89H，F2H 大于 69H，减完后还有 89H，由于减法后没有借位，CY 值被设成 0。如果要看剩余值的话，请将 P1 改为 ACC 值。

【例 8-14】 8-5-2.ASM

```
$MOD51
    ORG         0000H
    CLR         CY              ;CY=00H
    MOV         A,#0F2H         ;A=F2H
    SUBB        A,#69H          ;A=A-69H=89H CY=0
    MOV         P1,A            ;P1=!PSW
    END
```

【例 8-15】 8-5-3.ASM

```
;2 BYTE DATA SUBSTRACT
$MOD51
    ORG         0000H
    MOV         R0,#20H
    MOV         R1,#030H
    CLR         CY
    MOV         A,@R0           ;A=(20H)
    SUBB        A,@R1           ;A=(20H)-(30H)
    MOV         @R1,A           ;(30H)=(20H)-(30H)
    INC         R0              ;R0=21H
    INC         R1              ;R1=31H
    MOV         A,@R0           ;A=(21H)
    SUBB        A,@R1           ;A=(21H)-(31H)-CY
    MOV         @R1,A           ;(31H)=(21H)-(31H)-CY
    END
```

8051 的减法和我们常见的十进位减法相似，一定是从个位数先减起，若有借位的话，再先把十位数值减 1（CY 值），然后才进行十位数的相减，若十位数相减有借位时也会反映在 CY 上。

8.6 逻辑指令 ANL/ORL/XRL

在 8051 逻辑指令当中，AND 运算是很常见的，OR 运算的机会也不少，而真正使用异或 XOR 运算的比较少。我们还是从程序中来学习，经过逻辑运算的结果依旧是放在 P1 端口上。

现在我们打算让 B5H 和 26H 两个数值分别使用 ANL、ORL 和 XOR 指令后，看 8 个 LED 的变化各有什么不同的地方？在做逻辑指令时，最好把数值表示方式改成二进制表示，以方便我们观察。

【例 8-16】 8-6-1.ASM

```
$MOD51
    MOV         A,#10110101B    ;送 B5H 到累加器
    ANL         A,#00100110B    ;值 26H 做 AND 逻辑与运算=24H
    CPL         A               ;A 值取反
    MOV         P1,A
    END
```

结果应该是 00100100B，由于 LED 的接法为负逻辑，所以在输出前做一次反相后才送到 P1 端口上。

【例 8-17】 8-6-2.ASM

```
$MOD51
    MOV         A,#10110101B    ;A=B5H
```

```
        ORL         A,#00100110B    ;A=B5H OR 26H=B7H
        CPL         A
        MOV         P1,A
        END
```

结果是 10110111B，应该有 6 个 LED 亮，只有 P1.6 与 P1.3 这两个 LED 灯不亮。

【例 8-18】 8-6-3.ASM

```
$MOD51
        MOV         A,#10110101B    ;A=B5H
        XRL         A,#00100110B    ;A=B5H XOR 26H=93H
        CPL         A
        MOV         P1,A
        END
```

结果是 10010011B，应该有 4 个 LED 亮才对，XOR 用在数据比较和加密上，一般的控制程序反而很少见。

```
B5H=10110101B
26H=00100110B

    10110101        10110101        10110101
AND)00100110    OR)00100110    XOR)00100110
    00100100        10110111        10010011
    =24H            =B7H            =93H

AND 与运算的口诀：只要有一个 0 输出就是 0
OR 或运算的口诀：只要有一个 1 输出就是 1
XOR 异或运算的口诀：相同为 0 不同是 1
```

图 8-4　AND、OR 与 XOR 的比较表

8.7　CALL 调用指令

如果你编写超过 50 行的程序，程序分支（PROGRAM BRUNCH）的写法是不可避免的。因为程序越来越大之后，程序不可能如同以前的例子，从头到尾都没有做条件判断或调用的操作。

当你写的程序越来越复杂时，你一定会发现有些程序的重复性是很高的，有些环节可能使用多次，这时你就要学习用调用 CALL 来处理这些重复性高的部分程序。有人把被调用的程序称为子程序（SUBROUTINE）或例程（ROUTINE）。这里我们统一称为例程好了，例程最后面一定是 RET 指令。当我们调用 CALL 例程时，8051 单片机要做一个改变 PC 程序记数值的操作，它会把现在的 PC 值（16 位）暂时存在堆栈指定的地址上，然后把 PC 值换成该例程的进入点，开始执行例程直到遇到 RET 指令，这时 8051 又会从堆栈中取回原先的 PC 值，继续原来的程序。

调用指令又有 ACALL 和 LCALL 两种，ACALL 编译后只占 2 字节，LCALL 则会占 3 字节。两个的差别在调用例程的远近，本书实验都是以 AT89C2051 为主，程序若在 2KB 以内的话，可以尽量采用 ACALL 调用以节省程序空间。

假设 ACC 累加器内有一堆数据，要每隔一小段时间就加 1 之后送到 P1 端口，即通过 LED

把该值显示出来。如果程序利用 LCALL，写法会像下面这个样子。

【例 8-19】8-7-1.ASM

```
BUFFER      DATA        20H
$MOD51
            ORG         0000H
            MOV         SP,#50H          ;知道为什么要设定堆栈 STACK?
            MOV         A,#00H           ;累加器=00H
            MOV         BUFFER,A         ;(20H)=00H
LOOP:       MOV         P1,BUFFER        ;P1=(BUFFER)
            INC         BUFFER           ;将 BUFFER 内容加上 1
            LCALL       DELAY            ;延迟一小段时间
            SJMP        LOOP
;ROUTINE
;为时间延迟用
DELAY:      MOV         R0,#00H
DLY:        MOV         R1,#00H
            DJNZ        R1,$
            DJNZ        R0,DLY
            RET
            END
```

DELAY 例程纯粹是一个延迟程序，由于 8051 单片机执行的速度太快了，我们把 BUFFER 内的值加 1 送到 P1 端口的时间只要百万分之几秒，如果不把速度放慢的话，肉眼是无法看到 LED 的变化。所以在 LED 被点亮后就暂停一下，这段时间就是调用 DELAY 例程造成的。DELAY 例程只做 R0 和 R1 减 1 的指令，利用单纯的减 1 的指令消耗时间，最后减到 R0 和 R1 都成 0 后才碰到 RET 指令，重新返回到主程序上，继续做 LOOP 的操作。

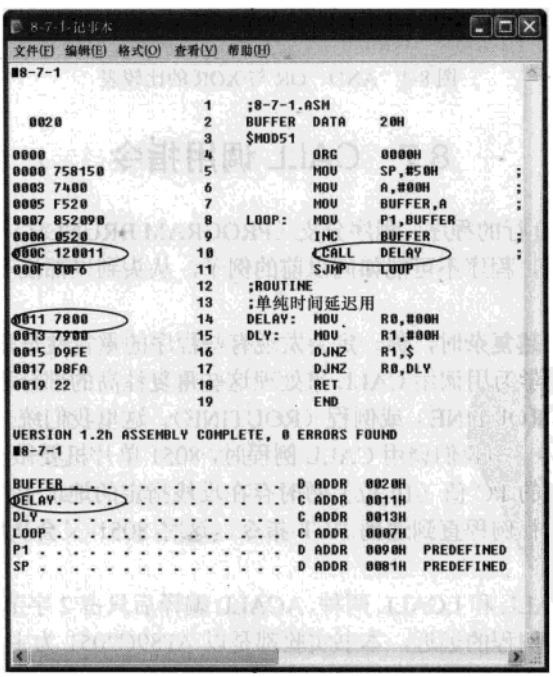
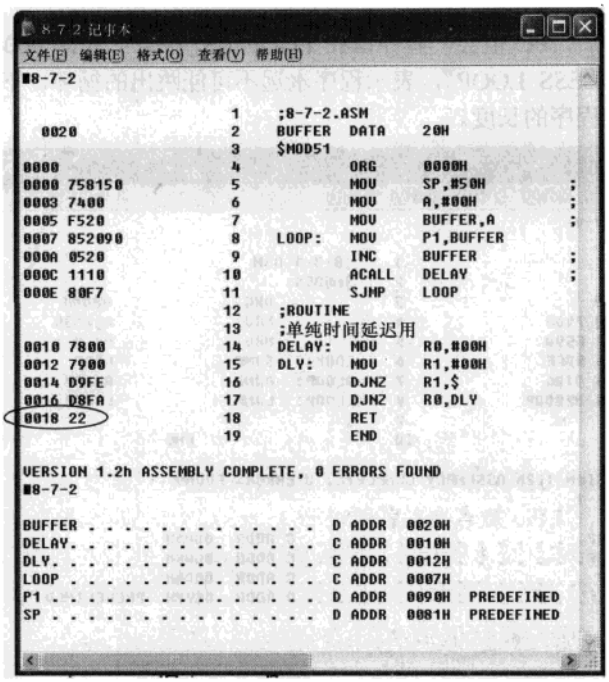


图 8-5　8-7-1.LST 文件

请注意 PC 值等于 000CH 的时候，有一个 LCALL DELAY 的调用操作，调用的地址正好是 0011H，而第 14 行是 DELAY 例程的开始点，往左边核对其 PC 值是 0011H，DELAY 例程存放在 0011H～0019H 共 9 字节，整个例程长度为 25 字节。



```
8-7-2 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
■8-7-2

                        1   ;8-7-2.ASM
  0020                  2   BUFFER  DATA    20H
                        3   $MOD51
0000                    4           ORG     0000H
0000 758150             5           MOV     SP,#50H          ;
0003 7400               6           MOV     A,#00H           ;
0005 F520               7           MOV     BUFFER,A         ;
0007 852090             8   LOOP:   MOV     P1,BUFFER
000A 0520               9           INC     BUFFER           ;
000C 1110              10           ACALL   DELAY            ;
000E 80F7              11           SJMP    LOOP
                       12   ;ROUTINE
                       13   ;单纯时间延迟用
0010 7800              14   DELAY:  MOV     R0,#00H
0012 7900              15   DLY:    MOV     R1,#00H
0014 D9FE              16           DJNZ    R1,$
0016 D8FA              17           DJNZ    R0,DLY
0018 22                18           RET
                       19           END

VERSION 1.2h ASSEMBLY COMPLETE, 0 ERRORS FOUND
■8-7-2

BUFFER . . . . . D ADDR  0020H
DELAY. . . . . C ADDR  0010H
DLY. . . . . C ADDR  0012H
LOOP . . . . . C ADDR  0007H
P1 . . . . . D ADDR  0090H  PREDEFINED
SP . . . . . D ADDR  0081H  PREDEFINED
```

图 8-6　8-7-2.LST 文件

请注意 PC 值等于 000CH 的时候，只把 LCALL 调用操作改为 ACALL，整个程序长度成为 24 字节。

8.8　跳转指令 JUMP

JUMP 指令也是改变 PROGRAM COUNTER 程序计数器的指令，强迫改变 PC 值，以便程序转向，与 CALL 不同的是 JUMP 跳转时，并不会将目前的 PC 值存入堆栈中。8051 单片机的无条件的 JUMP 指令又分为 SJMP、AJMP 和 LJMP，SJMP 跳转的距离只能在 128 字节以内，AJMP 可达 2KB 之多，LJMP 则能够跳到 64KB 的任一个地址上。

ASM51 编译程序在翻译你写的汇编语言程序时，就会顺便检查你写的 JUMP 指令是否恰当。在编写 AT89C2051 的程序时，碰到短程的跳转尽量用 SJMP，其他跳转就用 AJMP 指令。写其他大型的程序就要用 LJMP 做远程的跳转。

【例 8-20】 8-8-1.ASM

```
$MOD51
        ORG         0000H
        MOV         A,#63H
        MOV         P1,A
```

```
LOOP:   SJMP        LOOP        ;程序永远停留在这里
ALOOP:  AJMP        ALOOP
LLOOP:  LJMP        LLOOP
;
        END
```

上面的程序执行后，PC 值会一直停留在 LOOP 这个标志上，即 0004H。这种循环我们称为无穷循环“ENDLESS LOOP”，表示程序永远不可能跳出的循环。下面另外两个 JUMP 范例仅在示范其转出程序的长度。

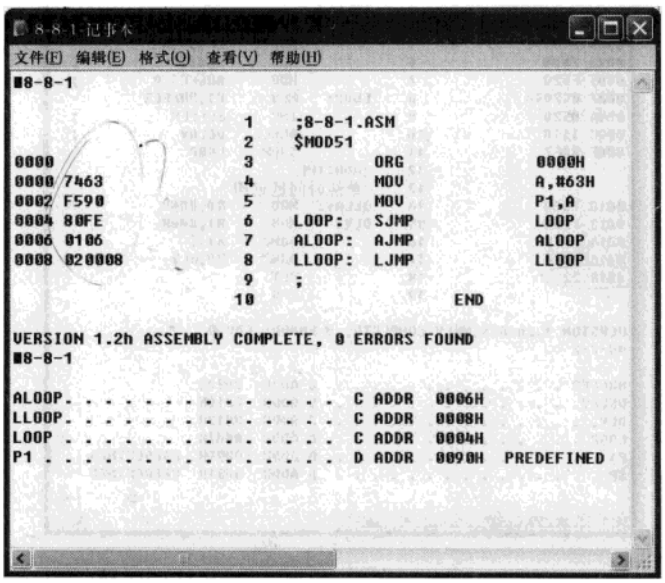


图 8-7　请注意三种 JUMP 写法所转出来的程序长度

8.9　DJNZ 条件跳转指令

在 8051 指令中有一个功能颇强的条件跳转指令 DJNZ，可以先把指定的内容值减 1，接着比较内容是否已减到 0 了，若内容为 0，程序就继续往下执行，若还不是 0 就往指定的地方执行，做了一连串的操作，但只要单行指令就行了。我们试着来熟悉使用这个指令的 DELAY 例程。

【例 8-21】 8-9-1.ASM

```
$MOD51
        ORG         0001H
        MOV         P1,#0FFH            ;将 FFH 送到 P1
;
LOOP:   SETB        P1.5                ;P1.5=1,LED 不亮
        ACALL       DELAY
        CLR         P1.5                ;P1.5=0,LED 亮
        ACALL       DELAY
        SJMP        LOOP
;
```

```
DELAY:  MOV     R2,#00H         ;R2=00H
DLY1:   MOV     R3,#00H         ;R3=00H
        DJNZ    R3,$            ;把 R3 内容减 1 直到 R3=0
        DJNZ    R2, DLY1        ;R2-1
        RET
;
        END
```

通电执行后发现代表 P1.5 的 LED 一直在闪烁，这表示程序在 LOOP 标志的循环中反复执行。

类似 DELAY 例程的写法会一直出现在本书的其他范例中，现在我们来了解一下 DELAY 例程的操作。这里面用到了两个 DJNZ 的指令，R2 的循环称外循环，属于 R3 的称内循环。

我们先观察内循环程序只有两行：

```
MOV    R3,#00H  ;R3=0
DJNZ    R3, $   ;R3=R3-1 直到 R3=0
```

内循环先把 R3 设成 0，接着 R3 减 1 直到 R3 等于 0 为止，总共做了 256 次的减 1 的操作。“$”字号对 ASM51 编译程序而言，是跳转到原 PC 值，所以第二行在实际执行时是 R3 一直减 0 之后才往下做。

```
        MOV     R2,#00H     ;R2=0
DLY1:   内循环              ;R3 做减 1 的操作 256 次
        DJNZ    R2,DLY1
        RET
```

外循环的主角换成 R2，写法跟内循环相似，不过 R2 减 1 后若不是 0 时，是跳到再做一次内循环，然后再做 R2 减 1 的操作。亦即 R2 每做一次减 1 的操作前就做 256 次 R3 减 1 的操作，所以整体来讲：DELAY 总共做了 65536 次减 1 的操作，次数一多后所累积的执行时间就多了。这种写法所产生的延迟时间约在 0.2～0.3s 左右，刚好够肉眼可以分辨，所以用些例程来做 LED 的闪烁控制一定够用的。

请务必要理解 DJNZ 的写法，如果不会用 DJNZ 而改用 ACC 累加器来处理时，程序会变得复杂且庞大，请看另一 DELAY 的写法：

```
DELAY:  MOV     COUNT1,#00H
DLY1:   MOV     COUNT2,#00H
DLY2:   DEC     COUNT2
        MOV     A,COUNT2
        JNZ     DLY2
        DEC     COUNT1
        MOV     A,COUNT1
        JNZ     DLY1
        RET
```

DJNZ 所使用的寄存器不仅限于 R0～R7，所有 DATA MEMORY 内的存储区都适用，所以前一个程序可以改写成以下的样子：

【例 8-22】 8-9-2.ASM

```
CNT1    DATA        2EH
CNT2    DATA        2FH
;
$MOD51
        ORG         0000H
```

```
        MOV         P1,#0FFH        ;将 FFH 送到 P1
;
LOOP:   SETB        P1.5            ;P1.5=1,LED 不亮
        ACALL       DELAY
        CLR         P1.5            ;P1.5=0,LED 亮
        ACALL       DELAY
        SJMP        LOOP
;
DELAY:  MOV         CNT1,#00H       ;(2EH)=00H
DLY1:   MOV         CNT2,#00H       ;(2FH)=00H
        DJNZ        CNT2,$          ;把 CNT2 内容减 1 直到 CNT2=0
        DJNZ        CNT1,DLY1       ;CNT1-1
        RET
;
        END
```

请看下面这个程序，判断一下 LED 总共会亮几次？

【例 8-23】 8-9-3.ASM

```
COUNT   DATA        30H
$MOD51
        ORG         0000H
        MOV         P1,#0FFH            ;将 FFH 送到 P1
        MOV         COUNT,#3FH
;
LOOP:   SETB        P1.5                ;P1.5=1,LED 不亮
        ACALL       DELAY
        CLR         P1.5                ;P1.5=0,LED 亮
        ACALL       DELAY
        DJNZ        COUNT,LOOP
        SJMP        $                   ;ENDLESS LOOP
;
DELAY:  MOV         R2,#00H             ;R2=00H
DLY1:   MOV         R3,#00H             ;R3=00H
        DJNZ        R3,$                ;把 R3 内容减 1 直到 R3=0
        DJNZ        R2,DLY1             ;R2-1
        RET
;
        END
```

8.10 JB 和 JNB 跳转指令

8051 单片机内部 DATA MEMORY 有两个区域是可以单一位寻址的，如果要取该区的某一位（bit）做条件判断时，就要用到 JB 和 JNB 这两个指令了。

JB 是判断该位若为 1 时要跳转到何处。

JNB 是判断该位若为 0 时要跳转到何处。

请先看一个程序范例：

【例 8-24】 8-10-1.ASM

```
$MOD51
        ORG     0000H
        MOV     P3,#0FFH        ;P3 INPUT MODE
```

```
        MOV     P1,#0FFH        ;LED 全部 OFF
CHECK:  JB      P3.0,ALL_ON     ;利用 JB 指令判断 P3.0 引脚
ONLY_ONE:                       ;P3.0=0
        CLR     P1.0            ;1 LED ON
        ACALL   DELAY           ;延迟程序执行速度
        SETB    P1.0            ;1 LED OFF
        ACALL   DELAY
        SJMP    CHECK           ;重新检查
;P3.0=1
ALL_ON: MOV     P1,#00H         ;ALL LEDS TURN ON
        ACALL   DELAY
        MOV     P1,#0FFH        ;ALLLEDS TURN OFF
        ACALL   DELAY
        SJMP    CHECK           ;重新检查
;
DELAY:          MOV     R0,#00H
DLY1:           MOV     R1,#00H
        DJNZ    R1,$
        DJNZ    R0,DLY1
        RET
        END
```

这个程序刚进入时，会把 P3 端口设成输入模式，而 P1 端口上的 LED 全部灭掉。接下来就是一个 JB P3.0,ALL_ON 的判断式，如果 P3.0=1 的话，程序会跳转到 ALL_ON 这段子程序中执行，八个 LED 会一直闪烁；反之，只有 P1.0 对应的 LED 会闪烁，其他 7 个 LED 都是不亮的。

下面这个程序检查 20H.4 的状态，若为 1，则 LED 全部点亮，反之，只有 1 个 LED 亮。

【例 8-25】 8-10-2.ASM

```
BUFFER      DATA        20H
TEST_BIT    BIT         20H.4
$MOD51
            ORG         0000H
            MOV         BUFFER,#00H
            MOV         P1,#0FFH        ;LED 全部 OFF
CHECK:      INC         BUFFER
            JB          TEST_BIT,ALL_ON ;TEST_BIT?
;
;TEST_BIT=0
ONLY_ONE:                               ;TEST_BIT=0
            CLR         P1.0            ;1 LED ON
            ACALL       DELAY           ;延迟
            SETB        P1.0            ;1 LDE OFF
            ACALL       DELAY
    SJMP    CHECK                       ;重新检查
;
;TEST_BIT=1
ALL_ON:     MOV         P1,#00H         ;ALL LEDS TURN ON
            ACALL       DELAY
            MOV         P1,#0FFH        ;ALL LEDS TURN OFF
            ACALL       DELAY
            SJMP        CHECK           ;重新检查
DELAY:      MOV         R0,#00H
DLY1:       MOV         R1,#00H
```

```
            DJNZ        R1,$
            DJNZ        R0,DLY1
            RET
            END
```

在写 8051 程序当中，条件判断式往往会让你昏头转向，以上个程序为例，程序中共有四个跳转的地方，如果一不小心的话，很可能就会把跳转的位置搞混。这一点连有经验的程序设计师也偶尔会中箭落马，我们建议用 CALL 的写法，程序的流程才会更清晰，也不容易出错，请再看一下另一种写法。

【例 8-26】 8-10-3.ASM

```
BUFFER      DATA        20H
TEST_BIT    BIT         20H.4
$MOD51
            ORG         0000H
            MOV         BUFFER,#00H
            MOV         P1,#0FFH        ;LED 全部 OFF
            MOV         SP,#60H
CHECK:      INC         BUFFER
            JNB         TEST_BIT,ONES   ;CALL 继续往下做
            SJMP        CHECK
ONES:       ACALL       ONLY_ONE        ;CALL 继续往下做
            SJMP        CHECK
ONLY_ONE:                               ;TEST_BIT=0
            CLR         P1.0            ;1LED ON
            ACALL       DELAY           ;延迟
            SETB        P1.0            ;1LED OFF
            ACALL       DELAY
            RET
;TEST_BIT=1
ALL_ON:     MOV         P1,#00H         ;ALL LEDS TURN ON
            ACALL       DELAY
            MOV         P1,#0FFH        ;ALL LEDS TURN OFF
            ACALL       DELAY
            RET
DELAY:      MOV         R0,#00H
DLY1:       MOV         R1,#00H
            DJNZ        R1,$
            DJNZ        R0,DLY1
            RET
            END
```

改用 CALL 的写法在 SJMP 方向上就非常清楚了，不论 JNB 的结果如何，最后都回到 CHECK 上。

8.11 CJNE 与 JC 的搭配应用

在 8051 中有两个很重要的条件判断指令：第一个是前面已经提到的 DJNZ，另一个就是现在所要提到的 CJNE。DJNZ 的应用，通常是用做循环指令；而 CJNE 的应用，则是用在判断数值是否相等，如果搭配 JC，就可以写出判别数值大小的功能。

CJNE（Compare and Jump if Not Equal）的原义是比较两个数值，如果不相等就跳转；JC（Jump if Carry is set）这个指令的功能，则是观察 CY 标志的状态，JC 是当 CY=1 就跳转。为

什么这两个指令相互搭配就可以比较数值的大小呢？我们用实际的例子来说明。

【例 8-27】 8-11-1.ASM

```
$MOD51
BUFFER  DATA        030H
NUMBER  EQU         10
CHECK   BIT         P1.0
;
        MOV         BUFFER,#10
        MOV         A,BUFFER
        CJNE        A,#NUMBER,NEQ       ;(BUFFER)-NUMBER
        SJMP        EQUAL
;(BUFFER)<>NUMBER
NEQ:    SETB        CHECK
        SJMP        $
;(BUFFER)=NUMBER
EQUAL   CLR         CHECK
        SJMP        $
;
        END
```

8-11-1.ASM 的用途，是检查 BUFFER 内的数值是否等于 10，若相等则 P1.0 输出 0，反之则输出 1。在这个程序里，因为 BUFFER 的值是 10，等于 10，所以 P1.0 输出的值为 0。

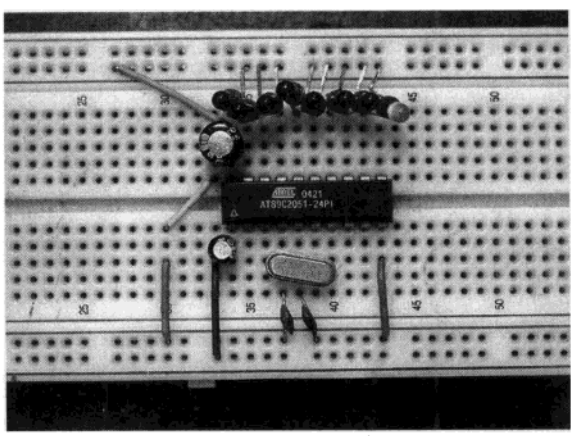


图 8-8　利用图 8-1 的线路图，我们实际验证 8-11-1.ASM 程序的执行结果，发现 P1.0 的输出为 0，所以 P1.0 上的 LED 灯亮了

【例 8-28】 8-11-2.ASM

```
$MOD51
BUFFER      DATA        030H
NUMBER      EQU         10
CHECK       BIT         P1.0
;
            MOV         BUFFER,#11
            MOV         A,BUFFER
            CJNE        A,#NUMBER,NEQ       ;(BUFFER)-NUMBER
NEQ:        JC          ST
    SJMP    GT_EQ
;(BUFFER)> =NUMBER
```

```
GT_EQ:      SETB        CHECK
    SJMP    $
;(BUFFER)<NUMBER
ST:         CLR         CHECK
    SJMP    $
;
    END
```

在 8-11-2.ASM 这个程序中，CJNE A,#NUMBER,NEQ 的操作也是检查 BUFFER 内的值是否等于 10，不过程序里又加上了 JC 指令，结果会变如何呢？

当 CJNE 的操作在执行时，是用 ACC 累加器中的数值减(ACC)-NUMBER，当 ACC 中的数值不够 NUMBER 减的时候，会产生借位信号，此时的 CY 标志位就会变成 1。

回过头再看看 8-11-2.ASM，ACC 的值是 11，NUMBER 是 10，因为 11−10=1 并不会产生借位操作，所以 CY=0，再由 JC ST 与 SJMP　GT_EQ 操作，我们就可以确定 P1.0 的输入为 1 。而 8-11-2.ASM 就是典型用来判断运算式中“大于等于”和“小于”的写法。

【例 8-29】 8-11-3.ASM

```
$MOD51
BUFFER      DATA        030H
NUMBER      EQU         10
CHECK       BIT         P1.0
;
            MOV         BUFFER,#11
            MOV         A,BUFFER
            CJNE        A,#NUMBER,NEQ           ;(BUFFER)-NUMBER
            SJMP        ST_EQ                   ;(BUFFER)=NUMBER
NEQ:        JC          ST_EQ
            SJMP        GTR
;(BUFFER)>NUMBER
GTR:        SETB        CHECK
            SJMP        $
;(BUFFER)<=NUMBER
ST_EQ:      CLR         CHECK
            SJMP        $
;
            END
```

问题来了，如果我们所要做的操作是“大于”和“小于等于”呢？很简单，看看 8-11-3.ASM 的范例，在 CJNE A,#NUMBER,NEQ 这行指令下面，再加上 SJMP　ST_EQ 的操作，这样一来，当等于的操作产生时，会进到 ST_EQ 的子程序中执行，而不会进到 GT，如此就可以做到“大于”和“小于等于”的操作了。

您可从下列公司的网站取得更进一步的信息：

（1）www.chipware.com.tw：查询单片机相关应用的参考资料。

（2）www.flag.com.tw：查询“8051 单片机彻底研究”系列丛书的相关介绍。

（3）www.fluke.com：查询掌上型数字电表的相关资料。

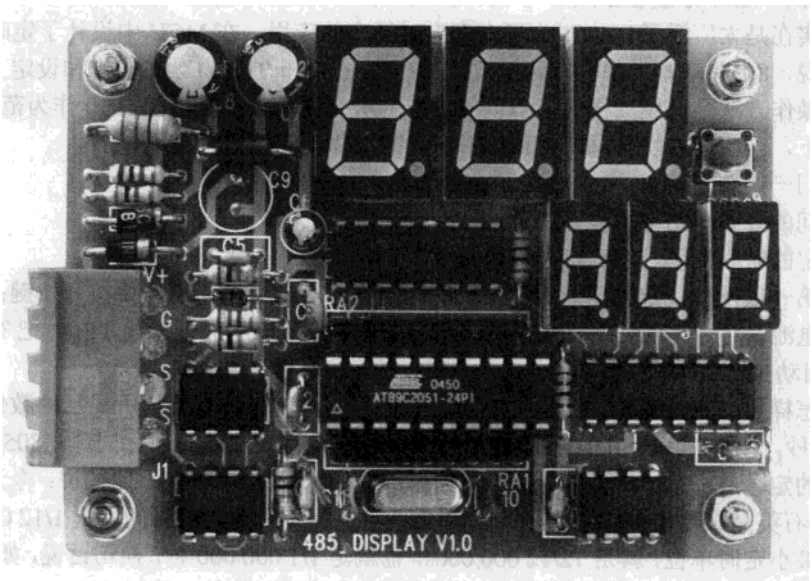
（4）www.tektronix.com：查询示波器的相关资料。

（5）www.topward.com：查询电源供应器的相关资料。

（6）www.atmel.com：制造 8051 的厂商，有很多 MCU 的相关参考资料。

（7）www.intel.com：查询 8051（MCS51）的相关介绍。

9 8051 Timer 的认识



接下来的三章是 8051 的常见用法，也是 8051 最引人入胜的地方。我们强烈建议初学者一定要实际操作，并将内容多看几遍，如果只是草草看过，是很难体会操作的原理和窍门的。本章首先要介绍的定时功能，是 8051 应用最多也最实用的功能，我们将通过程序范例和示波器观察，认识 8051 定时功能的工作原理及应用技巧。

第 9 章　8051 Timer 的认识

9.1　Timer0 的操作

日常生活中，定时功能无处不在：手表、马蹄表、闹钟、红绿灯切换、微波炉的时间设定……，定时的应用实在是太广泛了！8051 为了众多定时场合的应用，在 MCU 中设计了定时器。

一般来说，8051 基本上都会有两组定时器：Timer0 和 Timer1。在操作和设定上，这两个定时器的操作方法是一模一样的，所以接下来的所有说明，都会以 Timer0 作为范例。

回想一下，当我们在操作马蹄表或闹钟时，想要做的事情有哪些？

（1）时间一定会走。

（2）时间的开始与结束可以详细记录。

（3）预定的时间到了，有个信息可以提醒我们。

要完成第一项操作，定时器一定要做启动的操作。对马蹄表来说，要按开始键；对闹钟来说，要放电池。对 8051 的 Timer0 来说，有一个负责定时启动的标志位 TR0，把 TR0 设定为 1，就是启动 Timer0 的意思，设成 0 就是停止的意思。

只知道怎样启动定时器这样还是不够的。我们都知道，马蹄表上面所显示的数值，最小单位是 0.01 秒，不停把每个 0.01 秒累加，结果显示在马蹄表上的时间。同理，8051 的定时器也有所谓的定时最小单位，其时间是石英晶振每个振荡周期的 12 倍。比如说：今天我们使用 12MHz 的石英晶振当作 MCU 的振荡器，那么石英晶振每个振荡的时间就是 1/12 000 000s，而定时器的最小定时单位，就是 12/12 000 000s，也就是 1/1 000 000 秒。换句话说，如果 MCU 使用 12MHz 的石英晶振，当定时器每数一次，就是经过百万分之一秒（1μs）。

Timer0 的时间值会存储在一组专门用来记录时间变化的寄存器里面：TH0、TL0。同样以 12MHz 的石英晶振为例，可以记录的最大值是 65 535μs（就是 TH0=FFH、TL0=FFH，FFFFH=65 535），超过 65 535μs 就会自动归零。有了这组寄存器，我们就可以完成前面所提到的第二项操作，只要开始计时之前先把这组寄存器归零，等计时结束时再来读取这组寄存器，就可以知道经过多少时间了。

当这组寄存器超过 FFFFH 后会自动归零时，会产生一个溢位（OVERFLOW）信号，把定时溢位标志 TF0 设成 1，所以只要通过这个功能，我们就可以完成前面提到的第三个操作，也就是“定时”功能。

接下来，我们直接通过实际的程序范例来介绍 Timer0 的用法。

【例 9-1】 9-1-1.ASM

```
$MOD51
            ORG         000H
START       MOV         TMOD,#01H           ;SET TIMER0 TO MODE1
            ACALL       T0_RELOAD           ;CALL RELOADFUNCTION
;
LOOP        JNB         TF0,LOOP            ;WAIT FOR OVERFLOW
            ACALL       T0_RELOAD           ;RELOAD TIMER0 SETTINGS
```

```
            CPL         P1.0                ;CPLP1.0
            AJMP        LOOP                ;WAIT FOR OVERFLOW AGAIN
;
;SET TIMER0 FOR 10ms(XTAL=12MHz)
T0_RELOAD:
            CLR         TR0                 ;STOP TIMER0
            MOV         TH0,#0D8H           ;(65536-10000)/256
            MOV         TL0,#0F0H           ;(65536-10000)%256
            CLR         TF0                 ;CLEAR TIMER0 OVERFLOW FLAG
            SETB        TR0                 ;START TIMER0
            RET
            END
```

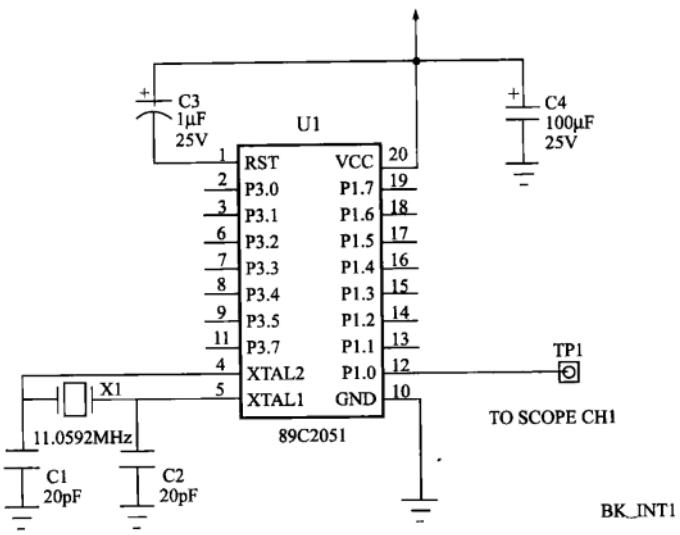


图 9-1　示波器观察到 P1.0 的电平变化

这个范例是利用 Timer0 产生每隔 10ms 就变化一次 P1.0 的高低电平，通过示波器观察，我们可以清楚地看到 P1.0 产生脉波的情况，请注意此时所使用的石英晶振是 12MHz 的规格。

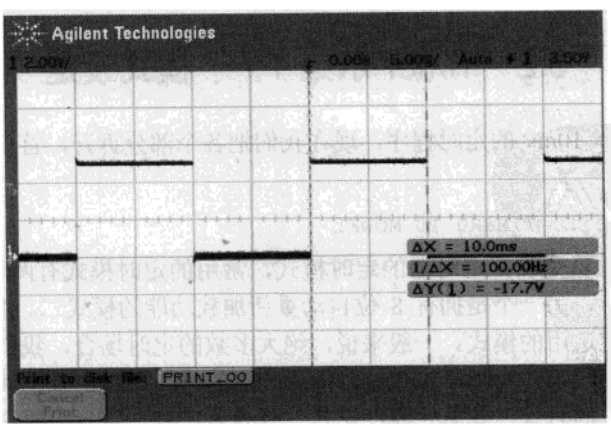


图 9-2　示波器观察到 P1.0 的电平变化，变化的时间间隔是非常准确的 10ms

如果要改成 5ms 就变化一次呢？只要改变两行程序就可以了。

```
MOV TH0,#0ECH ; (65536-5000) / 256
MOV TL0,#78H; (65536-5000) %256
```

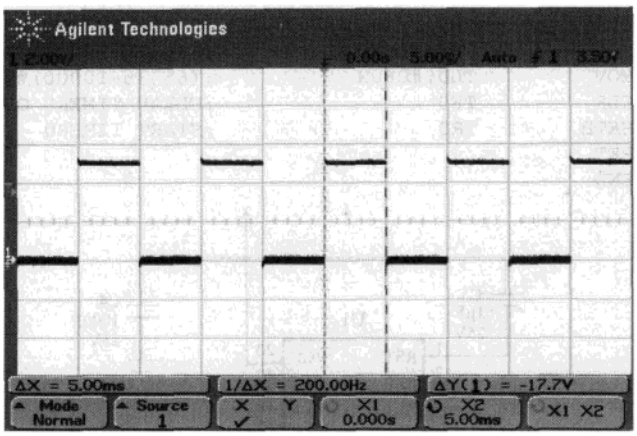


图 9-3　更改程序后，P1.0 变成 5ms 改变一次。我们得到一个很重要的信息：选用的石英晶振如果振荡频率很稳定，则定时器所读取的时间值一定也很准确

这个程序一共分为三个部分：

（1）是 START，主要是设定定时器的使用模式。

（2）是 LOOP，是等待定时器的倒数时间，时间一到就改变 P1.0 的状态。

（3）是 T0_RELOAD 这一整段的程序，主要操作是设定定时器的时间间隔及启动定时器的定时功能。

一口气介绍完定时器的所有主要功能，现在的你可能会觉得“写的很清楚，看的很模糊”什么是定时器的模式？TH0 和 TL0 的值是怎么来的？如果用的石英晶振不是 12MHz 的话，时间间隔怎么计算？超过 65 536μs 的定时功能如何处理？别急别急，我们会慢慢为你抽丝剥茧，让你轻松学会定时器的用法。

9.2　Timer 示范 1——模式设定

刚刚示范了一个 10ms 的定时程序，现在我们把各个部分拆开，首先要介绍的是定时模式设定。

```
MOV TMOD,#01H;SET TIMER0 TO MODE1
```

上面这行程序，就是设定定时器的定时模式，常用的定时模式有两种：一个是拥有 16 位定时能力的模式一，另一个是拥有 8 位自动重新加载功能的模式二。我们整理了表 9-1，让你可以轻松地设定选用的模式。一般来说，绝大多数的定时场合，我们都会以模式一来计时，因为定时的时间较长，以程序加载定时值所产生的时间误差是微乎其微。当定时的时间很短，需要斤斤计较的时候，就需要通过模式二来处理，特别当我们要进行通信的时候，一定要把 Timer1 设定成模式二才能执行。

表 9-1　Timer 模式的设定

Timer1	Timer0	程序写法
模式一	模式一	MOV TMOD, #11H
模式二	模式一	MOV TMOD, #21H
模式一	模式二	MOV TMOD, #12H
模式二	模式二	MOV TMOD, #22H

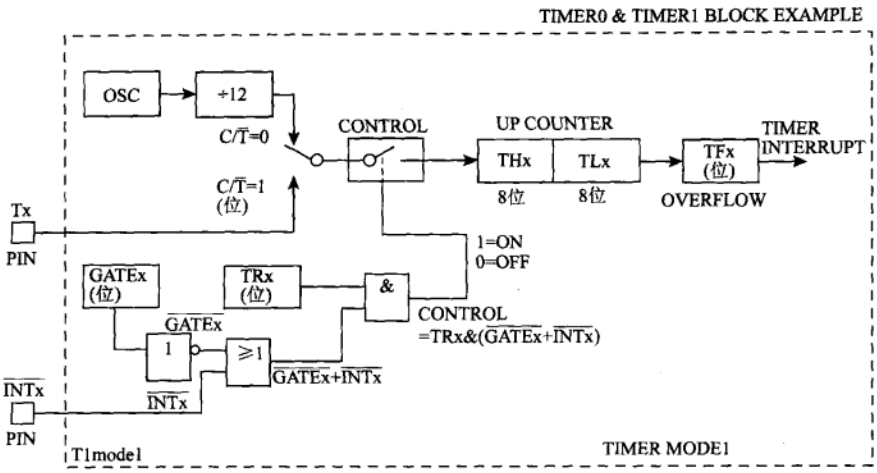


图 9-4　模式一的工作框图

图 9-4 中，定时模式一是 16 位的定时功能，左上方的 OSC 是石英晶振的振荡频率，每振荡 12 次，定时器就会数 1 次。若要启动定时功能，必须将 TRx 设为 1。当计数值超过 65 535，TFx 会被设成 1，此溢位标志位可产生中断信号。

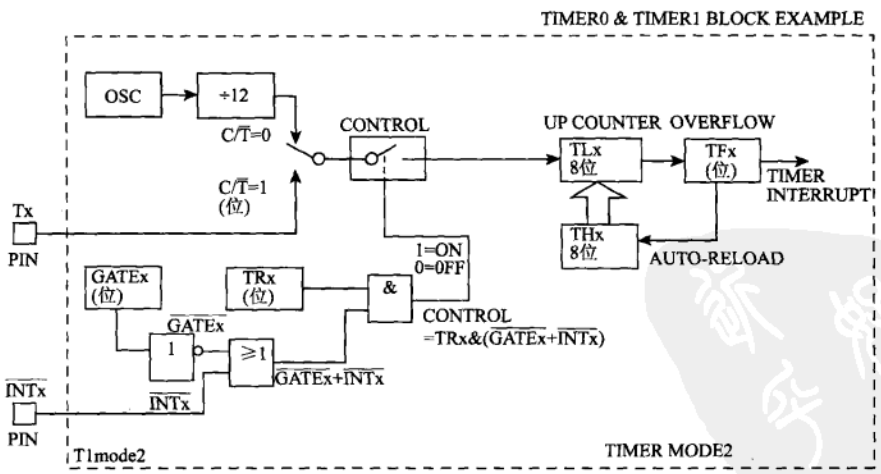


图 9-5　模式二的工作框图

图 9-5 中，工作的原理大致上与模式一相同，最大的差别在于模式二具有自动载入的功能，当溢位产生时，会自动将 THx 的值再填入 TLx 里，并自动将 TFx 清为 0。

9.3 Timer 示范 2——时间间隔的计算与设定

定时器的时间间隔计算跟石英晶振的振荡频率紧密相关，只要掌握“最小定时单位”的观念，就可以很快地将定时器的设定值算出来。

“最小定时单位”就是石英晶振 12 次所花费的时间。8051 常用的石英晶振的振荡频率是 12MHz 与 11.0592MHz，换算成最小定时单位，使用 12MHz 石英晶振时是前面所提到的 1μs，使用 11.0592MHz 石英晶振是：12 × (1/11.0592) =1.085 μs。

所以，当我们要为 Timer0 设定 10ms（10 000μs）的定时功能时，使用 12MHz 石英晶振，定时器必须数 10 000/1=10,000 次才算是 10ms；同理，使用 11.0592MHz 石英晶振时，定时器必须数 10 000/1.085=9,217 次才算是 10ms。

聪明的你是否已经联想到前面所提到“模式一”和“模式二”的使用场合？拿上面所举的例子来看，要使用 10ms 的定时功能，定时器至少要数 9 217 次以上，可是，模式二只计时 256 次（2 的 8 次方）就重新算起，所以我们使用模式一来计时是比较恰当的选择。

以 MCU 使用 12MHz 石英晶振为例，要完成 10ms 的定时功能，我们必须将 TH0 和 TL0 设定为下面的值。

```
MOV TH0,#0D8H; (65536-10000)/256
MOV TL0,#0F0H; (65536-10000)%256
```

TH0 跟 TL0 的值是怎么来的？在回答这个问题之前，先回想一下刚开始介绍定时器时，我们所提到的溢位功能。

当这组寄存器（TH0、TL0）超过 FFFFH 再变成零时，会产生一个计时溢位（OVERFLOW）信号，也就是溢位标志 TF0 设成 1，所以只要通过这个功能，我们就可以完成前面提到的第三个操作，也就是“定时”功能。

请记住 8051 的定时器只可以递增（UP COUNTER），利用计时溢位的操作，可以提醒我们定时的时间已经到了，而要达成溢位，就是一定要数到定时器“归零”，以 16 位的模式一来说，“归零”就是要数到 65 536 次（2 的 16 次幂），从 65 536 倒算 10 000 次当做起点，每数 10 000 次就会产生溢位，这个起点的值，就是 65 536−10 000=55 536。

还记得计算器有个“进制转换”的功能吗？此时只要把 55 536 在十进制状态下输入，再点击十六进制，我们会得到 D8F0 的值，这个值就是定时器 Timer0 当中 TH0 和 TL0 的值了。

如果换成了 11.0592MHz 的石英晶振，每数 10 000 次就会产生溢位，这个起点的值 TH0 跟 TL0 的值又应该是多少呢？答案是：

TH0=DBH，TL0=FFH，你算对了吗？

9.4 Timer 示范 3——定时功能的使用

经过前面几节内容的讲解，我们已经清楚地知道定时功能的工作原理，现在写一个定时功能的程序吧。

【例 9-2】 9-4-1.ASM

```
$MOD51
        ORG         0000H
START:  MOV         TMOD,#11H       ;TIMER0=MODE1                   (1)
        MOV         TH0,#0D8H       ;(65536-10000)/256              (2)
        MOV         TL0,#0F0H       ;(65536-10000)%256
        SETB        TR0             ;START TIMER0                   (3)
;
LOOP:   JNB         TF0, LOOP       ;WAIT FOR OVERFLOW              (4)
        CLR         TF0             ;CLEAR TF0                      (5)
        CPL         P1.0            ;CPL P1.0                       (6)
        SJMP        LOOP            ; WAIT FOR OVERFLOW AGAIN       (7)
        END
```

上面这一段程序，是希望每 10ms 定时将 P1.0 电平反相的程序，选用的石英晶振是 12MHz。

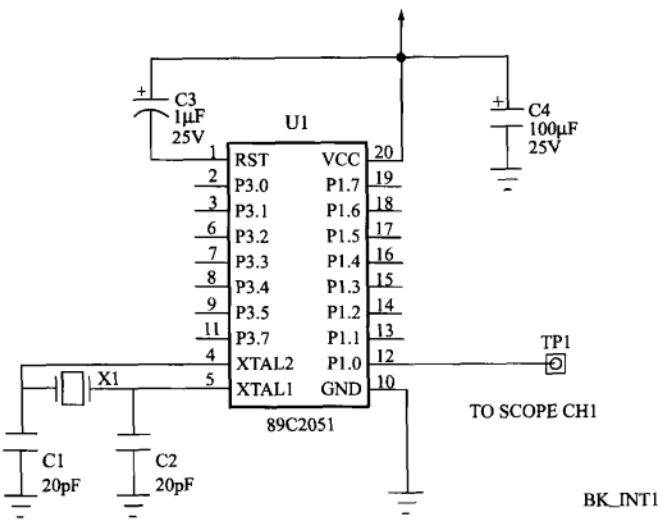


图 9-6　测试定时功能用的实验线路，接下来的实验会以这个线路做为范例

（1）将 Timer0 设定为定时模式一。

（2）将 TH0 和 TL0 填为 10ms 的设定值。

（3）让 Timer0 开始计时。

（4）等待 10ms 的溢位信号。

（5）将溢位信号清除。

（6）将 P1.0 电平反相。

（7）等待下一次 10ms 溢位。

将程序编译好烧录到 AT89C2051 中，然后我们用示波器观看一下 P1.0 所送出的信号是否正确。

奇怪？为什么 P1.0 反相的时间间隔是 65ms？怎么跟想像的完全不一样？回想一下我们遗漏了什么？

问题出在模式一的定时功能。它拥有 16 位的定时能力，可是却没有自动重新载入的功能。

因此当它数到 65 536 的时候，它会从 0 开始数起，所以当溢位发生时，它已经又数了 65 536 次了。那么要怎么修改才能做到 10ms 的定时功能呢？简单来说，就是我们通过程序去更新这个定时值，改写好的程序如下：

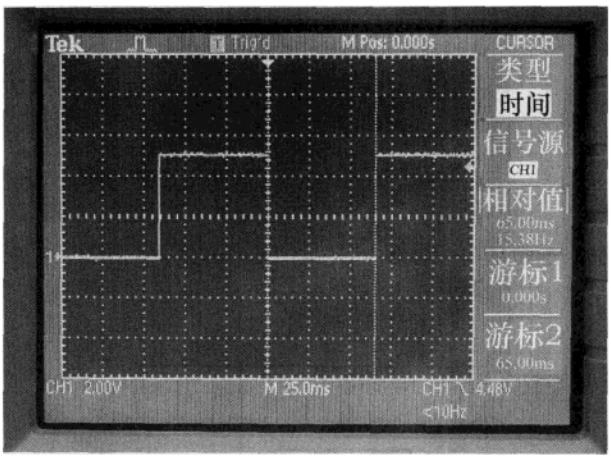


图 9-7　用示波器确认程序是否正确

【例 9-3】 9-4-2.ASM

```
$MOD51
        ORG     0000H
START:  MOV     TMOD, #11H          ;SET TIMER0 TO MODE1
        MOV     TH0, #0D8H          ;(65536-10000)/256
        MOV     TL0, #0F0H          ;(65536-10000)%256
        SETB    TR0                 ;START TIMER0
;
LOOP:   JNB     TF0, LOOP           ;WAIT FOR OVERFLOW
        ACALL   T0_RELOAD           ;RELOAD TIMER0 SETTINGS
        CPL     P1.0                ;CPL P1.0
        SJMP    LOOP                ;WAIT FOR OVERFLOW AGAIN
;
T0_RELOAD:
        CLR     TR0                 ;STOP TIMER0
        MOV     TH0,#0D8H           ;(65536-10000)/256
        MOV     TL0, #0F0H          ;(65536-10000)%256
        CLR     TF0                 ;CLEAR TIMER0 OVERFLOW FLAG
        SETB    TF0                 ;START TIMER0
        RET
;
        END
```

我们加入了一个通过程序去重新载入的功能（T0_RELOAD），当溢位发生时，我们会先把 10ms 的定时值填入 TH0 和 TL0 这组寄存器中，并清除溢位标志 TF0，如此一来，10ms 定时的功能就完成了。

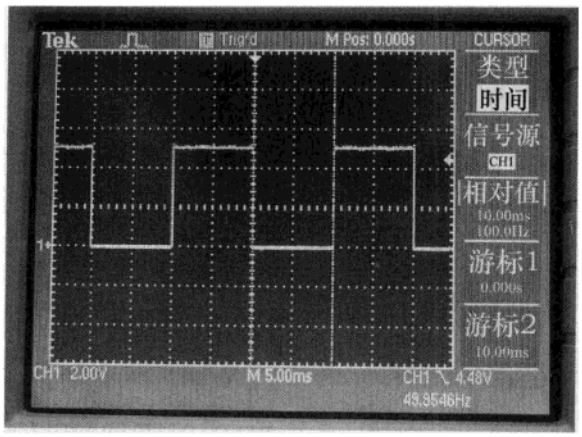


图 9-8　程序修改后 P1.0 的信号波形

9.5　Timer 示范 4——自动重新载入

调用子程序载入定时值是模式一要使用定时功能的变通方式，如果换成模式二，就完全没有这样的问题。依照模式一的设定方式，如果我们要使用一个 100μs 的定时器，写法应该是这样：

【例 9-4】 9-5-1.ASM

```
$MOD51
        ORG     0000H
START:  MOV     TMOD,#11H       ;SET TIMER0 TO  MODE1
        MOV     TH0,#0FFH       ;(65536-100)/256
        MOV     TL0,#9CH        ;(65536-100)%256
        SETB    TR0             ;START TIMER0
;
LOOP:   JNB     TF0,LOOP        ;WAIT FOR OVERFLOW
        ACALL   T0_RELOAD       ;RELOAD TIMER0 SETTINGS
        CPL     P1.0            ;CPL P1.0
        SJMP    LOOP            ;WAIT FOR OVERFLOW AGAIN
;
T0_RELOAD:
        CLR     TR0    ;STOP TIMER0
        MOV     TH0,#0FFH       ;(65536-100)/256
        MOV     TL0,#9CH        ;(65536-100)%256
        CLR     TF0             ;CLEAR TIMER0 OVERFLOW FLAG
        SETB    TF0             ;START TIMER0
        END
```

从示波器上观察 P1.0 的信号，我们发现：P1.0 反相的时间间隔竟然是 110μs，这多出来的 10μs 是怎么来的？答案是：这是调用 T0_RELOAD 这段子程序的执行时间。

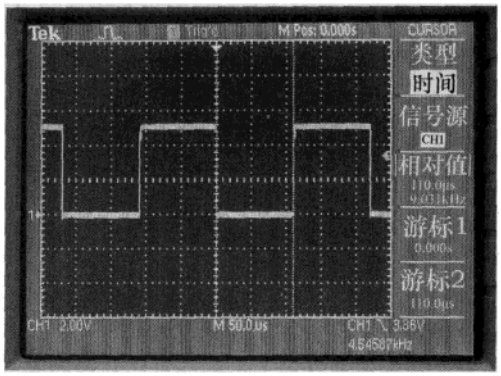


图 9-9　多了 10µs 的时间，是 T0_RELOAD 这段程序的执行时间

当我们使用 10ms 的定时功能时，10µs 对 10ms 来说，仅仅占了千分之一的比例，因此这么小的误差对整个时间间隔来说，影响是非常小的；可是当我们使用 100µs 的定时功能时，10µs 就占了 10%的比例，这样的影响就不能忽视了。因此，拥有自动重新载入功能的模式二，就显得非常重要。我们来看看模式二是怎么做到的？

【例 9-5】 9-5-2.ASM

```
$MOD51
        ORG     0000H
START:  MOV     TMOD,#12H       ;TIMER0=MODE2
        MOV     TH0,#9CH        ;(256-100)
        SETB    TR0             ;START TIMER0
;
LOOP:   JNB     TF0, LOOP       ;WAIT FOR OVERFLOW
        CLR     TF0             ;RELOAD TIMER0 SETTINGS
        CPL     P1.0            ;CPL P1.0
        SJMP    LOOP            ;WAIT FOR OVERFLOW AGAIN
        END
```

再从示波器看看 P1.0 的信号，我们发现：这真是太神奇了，屏幕上正不偏不移地显示着 100µs 的时间差。

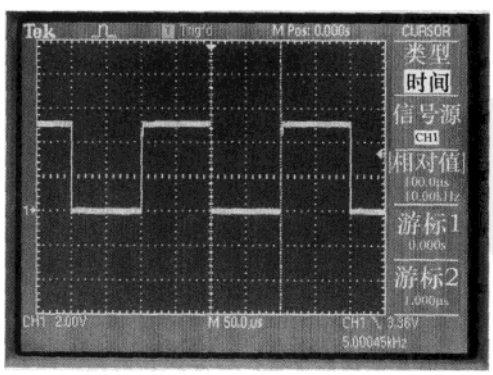


图 9-10　改用计时模式二的自动重新载入功能，定时功能变得更准确了

怎么知道 T0_RELOAD 这段程序需要 10μs 的时间来执行？如果我们推算一下程序指令的执行时间，我们就不难理解为什么会有这段误差产生了。

8051 单片机一般的指令只要一个机器周期的时间，跳转或调用则要 2～3 个机器周期。而一个机器周期相当于石英晶振振荡 12 次的时间。因此，以 12MHz 的石英晶振为例：如果指令的执行时间需要一个机器周期时，那么它的执行时间就是 $12\times(1/12\,000\,000)=10^{-6}s=1\mu s$。

8051 各指令执行的的时间可以在书最后附录的 8051 指令集中找到，标示为 2C 的指令，需要 2 个机器周期才能完成指令操作；标示为 4C 的指令，则需要 4 个机器周期才能完成操作；如果没有标示的，执行时间就是 1C，也就是 1 个机器周期可以完成指令操作。

举例来说：在指令集上 02H 的位置，有一个 LJMP addrl6[3B,2C]的指令，这是表示 LJMP addrl6 这个指令会占用 3 个字节的程序空间，执行时需要 2 个机器周期的时间来完成操作。

从定时器 Timer0 产生溢位到我们完成重新填入的值，这段程序所执行的时间经过一一查表后得到如下的结果：

```
JNB     TF0,LOOP        ;2μs(2 Cycles)
ACALL   T0_RELOAD       ;2μs(2 Cycles)
CLR     TR0             ;1μs(1 Cycles)
MOV     TH0,#0FFH       ;2μs(2 Cycles)
MOV     TL0,#9CH        ;2μs(2 Cycles)
CLR     TF0             ;1μs(1 Cycles)
```

最后的结论是当定时器重新再启动定时之前，已经花费了 10μs 的时间来进行判断与设定。

9.6 Timer 示范 5——超过 65ms 的定时功能

介绍了一堆定时功能的用法，看起来好像都不怎么实用，因为日常生活中的最小定时单位是秒，10ms 是百分之一秒，100μs 是万分之一秒，使用这样的定时器，连最基本的读秒功能都做不到。

问题总是有变通的方法可以解决的，只要我们愿意花点时间去理解定时的原理。既然能够做出 10ms 的定时器，就可以做出 1s 甚至 100s 的定时器。

还记得介绍过的“最小定时单位”吗？我们知道定时器的功能，是通过“最小定时单位”的累加来完成，那么我们也可以将 10ms 的定时器当作最小定时单位，再另外做一个定时器，你想到了吗？

MCU 里还有很多空着没用的寄存器，拿几个出来当作类似 TH0 和 TL0 这组寄存器的功能，就可以让定时的时间扩充很多很多了。来看看下面这个程序吧：

【例 9-6】 9-6-1.ASM

```
$MOD51
        ORG     0000H
START:  MOV     R0,#100             ;TIMER  BUFFER                  (1)
        MOV     TMOD,#11H           ;SET TIMER0 TO  MODE1
        MOV     TH0,#0D8H           ;(65536-10000)/256
        MOV     TL0, #0F0H          ;(65536-10000)%256
        SETB    TR0                 ;START TIMER0
;
```

```
LOOP:   JNB     TF0, LOOP           ;WAIT FOR OVERFLOW
        ACALL   T0_RELOAD           ;RELOAD TIMER0 SETTINGS
        DJNZ    R0, LOOP            ;100 COUNTS FOR 10ms            (2)
        MOV     R0, #100            ;RELOAD 100 COUNTS              (3)
        CPL     P1.0                ;CPL P1.0 FOR 1s
        SJMP    LOOP                ;WAIT FOR OVERFLOW AGAIN
T0_RELOAD:
        CLR     TR0                 ;STOP TIMER0
        MOV     TH0,#0D8H           ;(65536-10000)/256
        MOV     TL0, #0F0H          ;(65536-10000)%256
        CLR     TF0                 ;CLEAR TIMER0 OVERFLOW FLAG
        SETB    TF0                 ;START TIMER0
        RET
;
        END
```

把原先的 10ms 定时器程序加入了（1）（2）（3）这三行程序，我们再来看看 P1.0 所送出的信号。

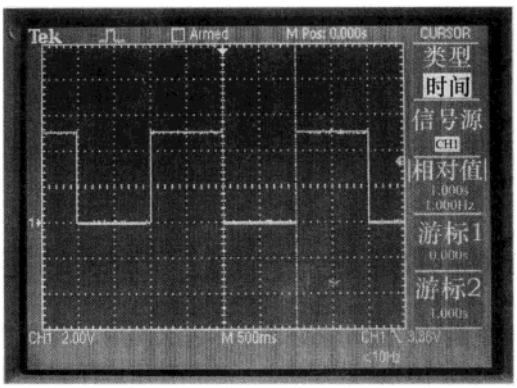


图 9-11　P1.0 信号是一秒才改变一次电平，时间间隔远远超过了原先定时器所能定时的范围

以为不可能达成的定时功能，经过了小小的修改后，出现了曙光。加入的这三行程序，实现了 1s 定时功能，到底是怎么做到的呢？

说穿了，就是应用“最小定时单位”的观念完成的，我们来看看程序的工作原理：

（1）利用 R0 做为定时的寄存器，功能有点类似 TH0 和 TL0 的功能，先在 R0 里填入 100。

（2）当溢位产生时，代表时间已经过了 10ms，此时程序会帮 R0 减 1，如果减 1 之后 R0 的值不是 0，则继续等待溢位产生，而不是执行 P1.0 的电平反相。

（3）经过了 100 次等待，R0 的值终于被减为 0 了，此时要做定时值载入的操作，把 R0 再填为 100，就像 T0_RELOAD 的功能一样，填完之后再执行 P1.0 的反相。

有了这样的概念后，我们就可以轻松完成所有应用场合的定时功能了。

9.7　设定定时器的标准流程

经过了一连串的介绍后，我们整理出一个定时器的使用流程，让你在最短的时间内，设定好需要的定时功能，如图 9-12 所示。

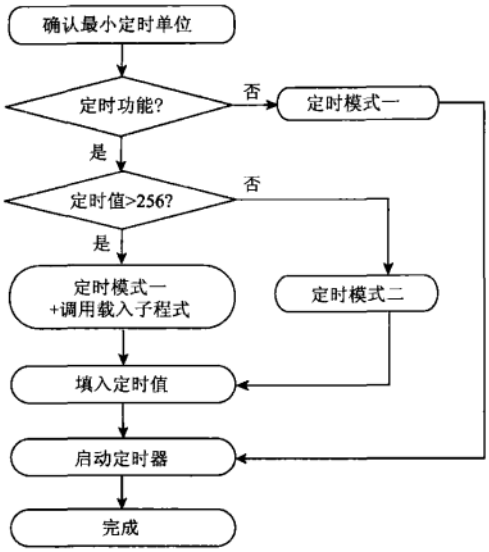


图 9-12　定时器设定流程图

（1）确定 MCU 所使用的石英晶振规格，计算出“最小定时单位”的值。

（2）套入所要应用的定时场合，如果不需要定时功能，则选用模式一。

（3）如果需要定时功能，计时值在 256 以内，可以选用模式二；如果不是，仍选用模式一，但是要加入重新载入的子程序。

（4）选择适当的模式后，计算所需的定时值，填入 THx、TLx 中（x=0，1）。

（5）启动定时器。

9.8　简易方波信号产生器

介绍完定时功能的用法，当然要找个实用的例子来牛刀小试一下。接下来，我们要做一个简易的方波信号产生器，先介绍一下什么叫方波。方波，顾名思义就是方方正正的波形，一个完整的方波，指的是信号的电平上升下降一个循环，也就是信号从这一次电平上升到下一次电平上升，这样算一个方波。假如我们要做一个 1kHz 的方波信号产生器，我们该怎么做呢？

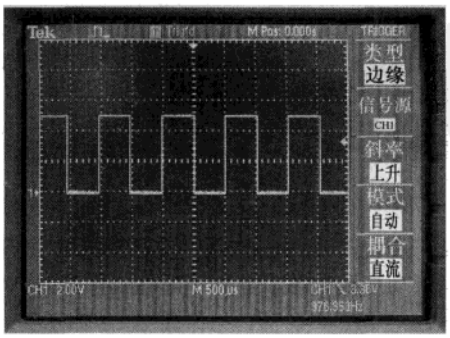


图 9-13　1kHz 方波的波形

1kHz 的方波，就是一个方波的周期为 1/1 000s，也就是 1ms。而我们知道一个完整方波需要变化电平两次，每变化一次信号电平需要 0.5ms，也就是 500μs。我们以 Timer0 为例，选用 11.0592MHz 的石英晶振，优先判断定时器该选用何种模式比较恰当。

需要的“最小定时单位”数=500/1.085=461（取整数）因为 461 已经超过模式二所能定时的范围，所以我们选用模式一。

65536-461=65075=FE33H（TH0=FEH，TL0=33H)求出 TH0 和 TL0 的值之后，我们就可以套用前面所介绍的程序。

【例 9-7】9-8-1.ASM

```
$MOD51
        ORG     0000H
START:  MOV     TMOD,#11H       ;SET TIMER0 TO  MODE1
        MOV     TH0,#0FEH       ;(65536-461)/256
        MOV     TL0,#33H        ;(65536-461)%256
        SETB    TR0             ;START TIMER0
;
LOOP:   JNB     TF0,LOOP        ;WAIT FOR OVERFLOW
        ACALL   T0_RELOAD       ;RELOAD TIMER0 SETTINGS
        CPL     P1.0            ;CPL P1.0
        SJMP    LOOP            ;WAIT FOR OVERFLOW AGAIN
;
T0_RELOAD:
        CLR     TR0             ;STOP TIMER0
        MOV     TH0,#0FEH       ;(65536-461)/256
        MOV     TL0,#33H        ;(65536-461)%256
        CLR     TF0             ;CLEAR TIMER0 OVERFLOW FLAG
        SETB    TR0             ;START TIMER0
        RET
        END
```

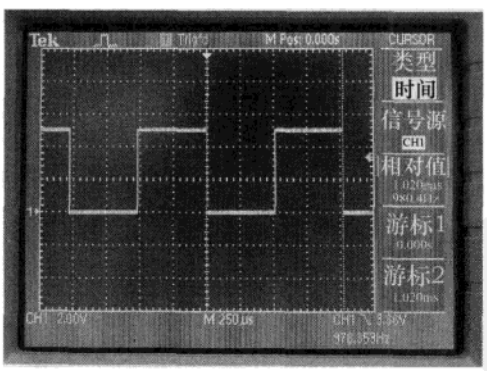


图 9-14　方波频率的观察，从 P1.0 输出方波，频率是 980Hz

通过示波器的观察，我们发现产生的方波频率是 980Hz，跟想像中的 1kHz 有点差距，原因在我们提到自动重新载入功能的时候，就知道是受 T0_RELOAD 这段程序的执行时间所影响，那我们该如何修正呢？

还记得前面使用 12MHz 的石英晶振时，调用载入程序会多了 10μs 的时间吗？（参考 8.7

的程序范例）换句话说，MCU 多算了 10/1=10 个最小定时单位的时间，所以，我们要把多算的扣除，此时的 TH0 和 TL0 应该怎么计算？

```
461-10=451
65536-451=65085=FE3DH(TH0=FEH, TL0=3DH)
```

因此，我们要把程序改写成下面的写法：

【例 9-8】 9-8-2.ASM

```
$MOD51
        ORG     0000H
START:  MOV     TMOD,#11H       ;SET TIMER0 TO  MODE1
        MOV     TH0,#0FEH       ;(65536-461)/256
        MOV     TL0,#33H        ;(65536-461)%256
        SETB    TR0             ;START TIMER0
;
LOOP:   JNB     TF0, LOOP       ;WAIT FOR OVERFLOW
        ACALL   T0_RELOAD       ;RELOAD TIMER0 SETTINGS
        CPL     P1.0            ;CPL P1.0
        SJMP    LOOP            ;WAIT FOR OVERFLOW AGAIN
;
TO_RELOAD:
        CLR     TR0             ;STOP TIMER0
        MOV     TH0,#0FEH       ;(65536-461)/256
        MOV     TL0,#3DH        ;(65536-461)%256
        CLR     TF0             ;CLEAR TIMER0 OVERFLOW FLAG
        SETB    TR0             ;START TIMER0
        RET
;
        END
```

再用示波器观察：998Hz。果然是 T0_RELOAD 所造成的误差，经过这样的修改，频率的误差就缩小啦！

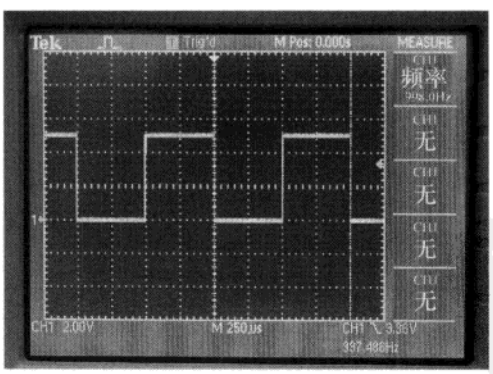


图 9-15　通过程序的修改，频率提高到 998Hz，很显然这个误差值是程序造成的

9.9　用 Timer 做串行的 Baud Rate 产生器

再来看看定时器不一样的应用吧！

8051 可以通过串行通信的功能，让两个 MCU 互相沟通，甚至可以让 MCU 跟计算机或其他仪器连线通信。在控制串行通信的传输速度时，我们必须通过 Timer1 来进行控制，先看看下面这一段程序吧。

【例 9-9】 9-9-1.ASM

```
$MOD51
DTART:  MOV     TMOD,#21H           ;TIMER1=MODE1           (1)
        MOV     TH1,#0FDH           ;9600BPS                (2)
        SETB    TR1                 ;START TIMER1           (3)
        MOV     SCON,#50H           ;SERIAL MODE1,UART ENABLE
        SETB    P1.0                ;P1.0=1
LOOP:   CLR     T1                  ;CLEAR T1
        MOV     A,#31H              ;OUTPUTDATA=31H
        CPL     P1.0                ;CPL P1.0
        MOV     SBUF,A              ;SEND 31H
WAIT:   JNB     TI,WAIT             ;WAIT FOR OUTPUT OK
        CPL     P1.0                ;CPL P1.0
        ACALL   DELAY               ;CALL DELAY
        SJMP    LOOP                ;SEND 31HAGAIN
DELAY:  MOV     R0,#00H             ;DELAY FOR A WHILE
        JNZ     R0,$
        RET
        END
```

这一段程序，是串行重复送出固定码的程序范例，我们所要关心的是程序里的前三行。

（1）设定 Timer1 为模式二。

（2）设定传输速率为 9600b/s。

（3）启动 Timer1。

在进行串行通信时，我们习惯把传输速度称为波特率（Baud Rate），关于波特率的相关说明，在稍后介绍串行通信时会有更详细的说明。在设定波特率时，请记得一定要使用 Timerl 的模式二做为波特率产生器，计算的方式如下：

$$\begin{aligned}\text{波特率} &= \frac{1}{32}\times\text{Timer1的溢位率}\\ &= \frac{1}{32}\times\frac{\text{石英晶振振荡频率}}{12\times(256-\text{TH1设定值})}\\ &= \frac{1}{32}\times\frac{11\,059\,200}{12\times(256-253(\text{FDH}))}=9600\text{b/s}\end{aligned}$$

假如要设定波特率为 4800 b/s，应该怎么修改呢？

$$\begin{aligned}4800\text{b/s} &= \frac{1}{32}\times\text{Timer1的溢位率}\\ &= \frac{1}{32}\times\frac{\text{石英晶振振荡频率}}{12\times(256-\text{TH1设定值})}\\ &= \frac{1}{32}\times\frac{11\,059\,200}{12\times(256-\text{TH1})}\\ &\rightarrow 256-\text{TH1}=6\\ &\rightarrow \text{TH1}=250=\text{FAH}\end{aligned}$$

9.10 与 Timer 有关的寄存器

最后，我们一起来看看跟 Timer 有关的两个很重要的寄存器：TMOD 和 TCON。

TMOD 模式控制寄存器解析

定时器的工作模式，最主要是由 TMOD 寄存器上的 M1 和 M0 两个位来决定。我们来看看 TMOD 各位的定义：

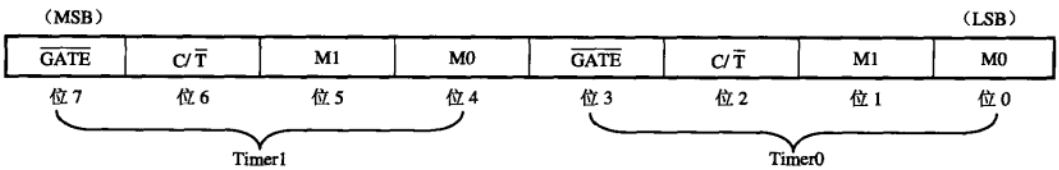


图 9-16 TMOD 各位的定义

M1=0，M0=1：

定时模式 1，Timer 被设定成 16 位的定时器，THx 和 TLx 串接，计时值最高可达 65 536。

M1=1，M0=0：

定时模式 2，Timer 被设定成 8 位自动重新载入的定时器，实际工作的 Timer 只有 TLx（8 位），每当 TLx 计时到产生溢位时，THx 上的值会自动载入 TLx 上，以方便 TLx 继续计时。

看到这里，有没有觉得 9.2 节的列表很熟悉？列表中所条列的程序写法，只要熟悉这个寄存器的结构，就可一目了然了。

TCON 控制寄存器解析：TCON 控制寄存器在定时功能上有两类很重要的标志，第一类是定时器的启动标志；第二类是计时溢位通知的标志，现在就来看看 TCON 各位的功能吧。

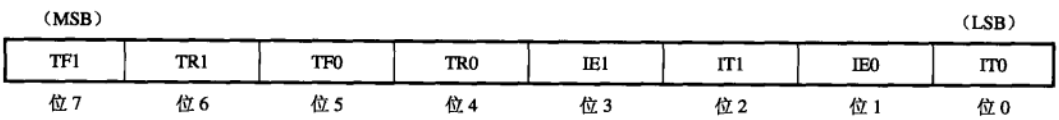


图 9-17 TCON 各位的定义

TRx(TCON.4、TCON.6)：

定时器的软件控制位（Run Control Bit），我们可以在程序中直接定址本位（TR0 或 TR1），以便控制 Timer 的开始或停止定时。

TFx(TCON.5、TCON.7)：

定时器上数到溢位发生时，本位被设成 1，请回头看图 9-4 的右方，本位实际上是一个对 CPU 要求中断的信号，当 CPU 反应并处理该中断时，TFx 会自动清除为 0。

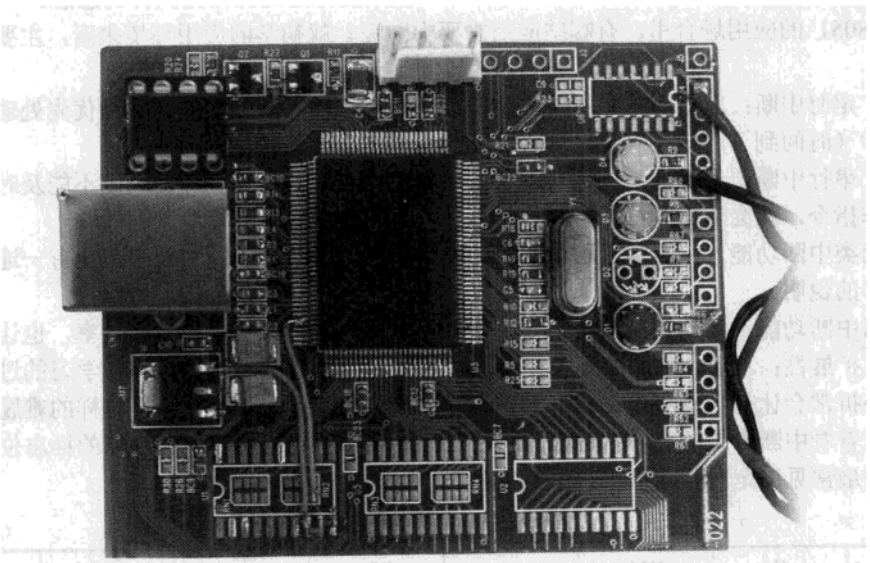
到此为止，Timer 的相关介绍就算告一段落。在本节的最后有提到一个“中断”专有名词，什么是中断？在下一章的内容里，我们会有更详细的介绍。

您可以从下列公司的网站取得更进一步的信息：

（1）www.chipware.com.tw：查询单片机相关应用的参考资料。

（2）www.flag.com.tw：查询“8051单片机彻底研究”系列丛书的相关介绍。
（3）www.agilent.com.tw：查询示波器的相关资料。
（4）www.tektronix.com：查询示波器的相关资料。
（5）www.atmel.com：制造8051的厂商，有很多MCU的相关参考资料。
（6）www.intel.com：查询8051（MCS51）的相关介绍。

10 8051中断的认识



中断是8051所有功能中最精华也最不容易理解的，它涉及到很多硬件工作上的细节，如果您对8051的硬件结构没有一定程度以上的认识，是很难将中断运用自如的。不过，现在您不用再担心这样的问题，通过本章所列举的程序范例，您可以轻松地运用8051的中断功能，并学会诊断程序错误的技巧，避开编写程序时所可能忽略的盲点，让中断功能不再是您学习8051时的梦想。

第 10 章　8051 中断的认识

10.1　中断的认识

中断就是暂时停下 MCU 正在执行的操作，让某些特定的操作先执行的意思。

在 8051 的应用场合里，有哪些应用需要中断呢？就初学的应用环境来看，主要分为两个部分：

（1）定时中断：就是有固定时间的紧迫性，不得不停下系统其他操作而优先处理的。例如：读秒（时间到了系统不停下来读秒，就会造成读秒的错误）的操作。

（2）串行中断：串行通信往往是用来下达操作指令用的，如果通信功能不能及时下达正确的操作指令，就会造成 MCU 的错误操作。

这两类中断功能，是灵活运用 MCU 的精华所在，在接下来的内容里，会有一连串的实例和详尽的说明。

使用中断功能，为的是让系统具备多处理的能力，让程序的操作更有效率，也让程序的应用更具扩展性；不过，如果中断使用不当，可能让系统资源严重困乏，在学习的过程中，这样的挫折常会让初学者止步不前，本章将以最浅显的例子，帮助你克服这样的难题。

先来看看中断专用的 IE 寄存器，在 IE 寄存器中，有许多启动中断功能的标志位，本书会介绍到最常见的定时中断与串列中断两种。

（MSB）							（LSB）
EA	X	ET2	ES	ET1	EX1	ET0	EX0
位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0

图 10-1　IE 中断控制寄存器

EA 标志位(IE.7)：这个位是 IE 寄存器中最重要的位，若 EA=0，则禁止系统中所有的中断请求，所以 8051 在系统重置后，一定要把此位清除为 0，假设 EA=1 时代表允许 CPU 响应中断，至于接受那个中断要求，则视其他 6 个位的设定值而定。

ES 标志位(IE.4)：决定是否允许串行传输端口的中断请求，0 代表 Disable，1 代表 Enable。

ET1 标志位(IE.3)：决定是否允许 Timer1 的中断请求，0 代表 Disable，1 代表 Enable。

ET0 标志位(IE.1)：决定是否允许 Timer0 的中断请求，0 代表 Disable，1 代表 Enable。

10.2　中断实例——定时中断

前一章刚介绍完定时功能，接下来，我们就先从定时中断开始，先来看看一个定时中断的例子：

【例 10-1】 10-2-1.ASM

```
$MOD51
        ORG         0000H
        LJMP        START           ;MAIN PROGRAM
        ORG         000BH
        LJMP        T0_ISR          ;TIMER0 INTERRUPT
;
        ORG         0030H
START:  SETB        P1.0            ;INITIALIZE P1.0=1
        MOV         TMOD,#11H       ;TIMER0=MODE1
        MOV         TH0,#0DBH       ;(65536-9217)/256
        MOV         TL0,#0FFH       ;(65536-9217)%256
        CLR         TF0             ;CLEAR TF0 FLAG
        SETB        TR0             ;START TIMER0
        SETB        ET0             ;ENABLE TIMER0 INTERRUPT
        SETB        EA              ;START INTERRUPT
LOOP:   SJMP        LOOP
;
T0_ISR: CPL         P1.0            ;CPL P1.0
        CLR         TF0             ;CLEAR TF0 FLAG
        MOV         TH0,# 0DBH      ;(65536-9217)/256
        MOV         TL0,#0FFH       ;(65536-9217)%256
        RETI                        ;RETURN FROM INTERRUPT
        END
```

这是一个每隔 10ms 将 P1.0 的电平反相一次的程序，使用振荡频率为 11.0592MHz 的石英晶振。从程序来看，我们发现程序操作的执行，应该是到 LOOP:SJMP LOOP 这一行就停止了，可是，为什么 MCU 还是会每隔 10ms 就把 P1.0 的电平反相一次呢?

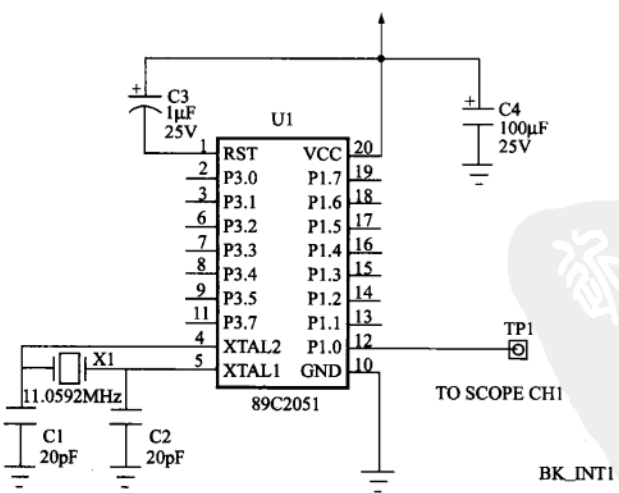


图 10-2　测试定时中断的电路图，P1.0 要接到示波器上

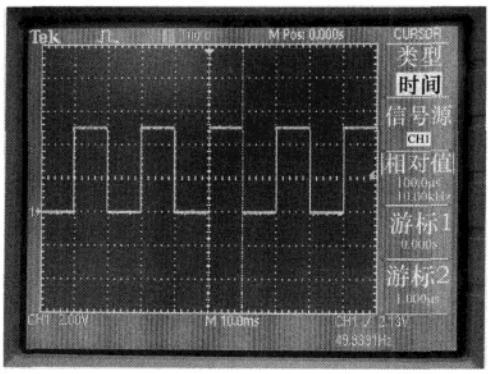


图 10-3　在示波器上观看到的 P1.0 的波形变化，每隔 10ms 反相一次

这就是中断功能的特色，接下来的章节，我们要以这个程序为主题，详细介绍工作的过程和需要留心的细节。

10.3　中断程序的标准范例

利用前一节提到的程序例，来看看程序的操作是如何工作的：

当我们启动定时中断之前，要先把定时器设定好，然后才启动中断功能。定时中断启动后，MCU 会去自动去检查 TF0 的状态，当 Timer0 产生溢位操作时，TF0 会被填成 1，而 Timer0 的中断就会产生。

当定时中断产生时，MCU 会到 000BH 的地方去执行程序，这个地址，我们称为 Timer0 的中断进入点，只要一发生 Timer0 中断，程序就一定会到这个中断进入点来执行我们指定的操作。等到中断操作完成后，MCU 会把控制权交还给主程序继续工作。

【例 10-2】 10-3-1.ASM

```
$MOD51
        ORG         0000H       ;PROGRAM START POINT                (1)
        LJMP        START       ;GOTO MAIN PROGRAM                  (2)
        ORG         000BH       ;TIMER0 INTERRUPT POINT             (3)
        LJMP        T0_ISR      ;GOTO TIMER0 INTERRUPT PROGRAM      (4)
;
        ORG         0030H       ;MAIN PROGRAM                       (5)
START:  SETB        P1.0        ;INITIALIZE P1.0=1                  (6)
        MOV         TMOD,#11H   ;TIMER0=MODE1                       (7)
        MOV         TH0,#0DBH   ;(65536-9217)/256                   (8)
        MOV         TL0,#0FFH   ;(65536-9217)%256                   (9)
        CLR         TF0         ;CLEAR TF0 FLAG                     (10)
        SETB        TR0         ;START TIMER0                       (11)
        SETB        ET0         ;ENABLE TIMER0 INTERRUPT            (12)
        SETB        EA          ;START INTERRUPT                    (13)
LOOP:   SJMP        LOOP        ;MAIN PROGRAM END                   (14)
;
T0_ISR: CPL         P1.0        ;CPL P1.0                           (15)
        CLR         TF0         ;CLEAR TF0 FLAG                     (16)
        MOV         TH0,#0DBH   ;(65536-9217)/256                   (17)
        MOV         TL0,#0FFH   ;(65536-9217)%256                   (18)
```

```
        RET1                    ;RETURN FROM INTERRUPT          (19)
        END
```

程序说明：

(1)～(2)

当 MCU 重置(RESET)后，程序会从 0000H 开始执行，因此，我们要在 0000H 的地方，用一个跳转指令，让主程序避开中断点的地址，从 0030H 开始，这样才不会干扰到中断程序的执行。

(3)～(4)

000BH 是 Timer0 的中断进入点，所以当中断产生时，MCU 会跳到这个地址执行程序，因此，我们通过一个跳转指令，将程序指向中断的子程序里，去执行固定时间要做的操作。

(5)

一般来说，现阶段会使用到的中断点都在 0030H 以前，所以我们将主程序由这里开始写起，就可以避开所有常用的中断程序跳转区。

(6)

主程序一开始先将 P1.0 的电平设定为 1，这是一个初始化的操作，让我们确认 P1.0 的起始电平，假如定时中断没有如期产生时，P1.0 应该是在 1 的电平不会变动的。

(7)～(11)

这一段程序是 Timer0 的设定，以产生 10ms 的定时中断为例，MCU 是选用 11.0592MHz 的石英晶振作为振荡器。

(12)

设定好 Timer0 之后，就可以将 Timer0 的中断功能打开。ET0 这个标志位的主要功能，就是管理 Timer0 的中断，填成 1 时，Timer0 的中断功能启动；填成 0 时，Timer0 的中断功能关闭。

(13)

将 Timer0 的中断功能打开后，就可以开始中断了。所有的中断功能，最终有一个 EA 标志位在管理，所以设定好中断功能后，最后要记得将这个标志位填成 1，这样才算正式启动中断。

(14)

主程序启动好定时中断功能后，就停在 LOOP：SJMP LOOP 这一行反复执行。

(15)～(18)

进入到中断后，我们将 P1.0 的电平反相，并且用程序重新填入 10ms 的计时值，就像前面介绍过的定时功能一样。

(19)

RETI 的指令功能，就是指示 MCU 离开中断程序返回主程序，这跟调用子程序后要返回主程序的 RET 指令是相似的，差别只在 RETI 是结束中断用，RET 则是结束子程序用。

	(MSB)							(LSB)
IE	EA	X	ET2	ES	ET1	EX1	ET0	EX0

(a)

	(MSB)							(LSB)
TCON	[TR1]	TR1	[TF0]	TR0	[IE1]	IT1	[IE0]	IT0

(b)

图 10-4　与定时中断有关的寄存器

图 10-4 中，除了前面介绍的 IE 寄存器以外，TCON 寄存器对定时中断来说也是十分重要的，因为 TCON 寄存器里有两个中断触发用的溢位标志位 TF0 和 TF1，中断的触发，是依这两个标志位的状态为触发条件的。

10.4 中断操作的观察

了解了定时中断的工作原理后，我们将程序稍为修改一下，并通过示波器来观看中断操作的执行。

【例 10-3】 10-4-1.ASM

```
$MOD51
MS10    EQU         -9217
TMH     EQU         (MS10)/256      ;10ms 定时值的声明
TML     EQU         (MS10)MOD 256   ;使用 11.0952MHz 石英晶振
;
        ORG         0000H           ;MCU 重置后的程序起点
        LJMP        START
        ORG         000BH           ;TIMER0 的中断点
        LJMP        T0_ISR
;
        ORG         0030H           ;主程序开始
START:  CLR         P1.0            ;P1.0=0
        MOV         TMOD,#11H       ;TIMER0=MODE1
        MOV         TH0,#TMH        ;填入重新定时值
        MOV         TL0,#TML
        CLR         TF0             ;清除 TF0 溢位标志
        SETB        TR0             ;TIMER0 开始定时
        SETB        ET0             ;启动定时中断功能
        SETB        EA              ;开始中断
LOOP:   CPL         P1.1            ;主程序将 P1.1 电平不断反相
        SJMP        LOOP
;
T0_ISR: SETB        P1.0            ;进入中断时把 P1.0 填成 1
        CLR         TF0             ;清除 TIMER0 的溢位标志
        MOV         TH0,#TMH        ;填入重新定时值
        MOV         TL0,#TML
        ACALL       DELAY           ;调用时间延迟的子程序
        CLR         P1.0            ;离开中断时把 P1.1 填成 0
        RETI
DELAY:  MOV         R0,#0           ;时间延迟的子程序
        DJNZ        R0,$
        RET
        END
```

观察中断过程，要从两个方面来看：第一是中断执行所花费的时间；第二是中断执行时，主程序的工作情况。在这个程序范例里，P1.0 是中断执行的时间观察，P1.1 则是观察系统工作的情况，我们用示波器的 CH1 观察 P1.0，CH2 观察 P1.1。

如果我们把定时中断的时间缩短为 100μs，程序改写如下，那么观察到的波形操作会是如何呢？

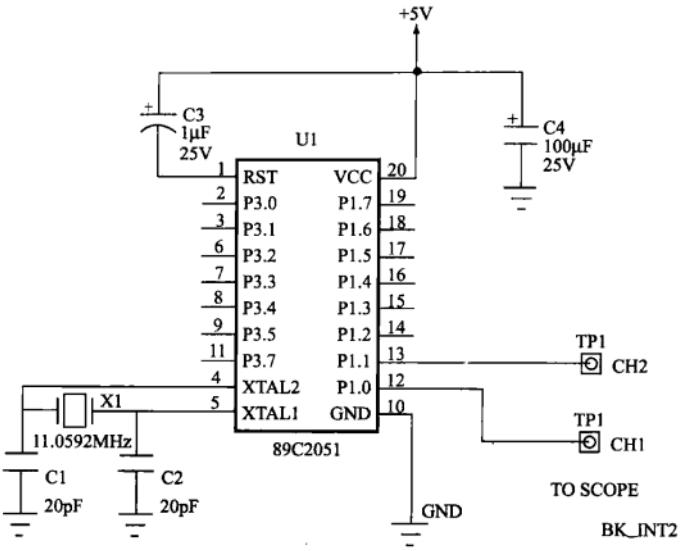


图 10-5　中断观察用了 P1.0 和 P1.1，这两点要接到数字式示波器上

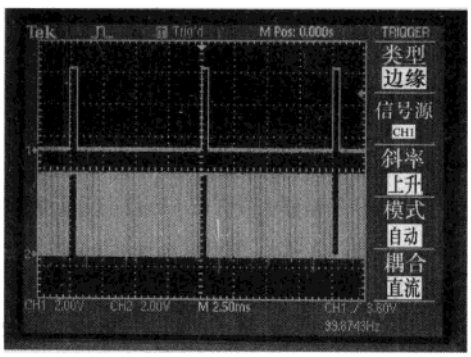


图 10-6　通过示波器所得到的两组信号波形，我们发现当定时中断执行的时候，主程序的工作是停下来的（P1.0 停止反相），等到中断离开之后，主程序才又恢复工作

【例 10-4】 10-4-2.ASM

```
$MOD51
TMR     EQU         (256-92)        ;100μs 定时值的声明
;
        ORG         0000H           ;MCU 重置后的程序起点
        LJMP        START
        ORG         000BH           ;TIMER0 的中断点
        LJMP        T0_ISR
;
        ORG         0030H           ;主程序开始
START:  CLR         P1.0            ;P1.0=0
        MOV         TMOD,#12H       ;TIMER0=MODE2
        MOV         TH0,#TMR        ;填入自动更新的 100μs 定时值
        CLR         TF0             ;清除 TIMER0 的溢位标志
        SETB        TR0             ;TIMER0 开始定时
        SETB        ET0             ;启动 TIMER0 的中断功能
```

```
        SETB    EA          ;中断开始
LOOP:   CPL     P1.1        ;主程序不断将 P1.1 的电平反相
        SJMP    LOOP
;
T0_ISR: SETB    P1.0        ;进入中断时将 P1.0 填成 1
        CLR     TF0         ;清除 TIMER0 的溢位标志位
        ACALL   DELAY       ;调用时间延迟的子程序
        CLR     P1.0        ;离开中断时将 P1.0 填成 0
        RETI
;
DELAY:  MOV     R0,#0       ;时间延迟子程序
        DJNZ    R0,$
        RET
        END
```

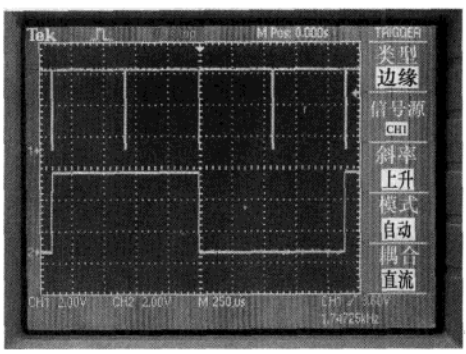


图 10-7　程序改写后所得到的波形。我们意外地发现：程序不但没有按我们所想象的 100µs 就产生一次中断。连主程序的操作也没有在中断结束后就继续执行，为什么？

10.5　中断对主程序的影响

为什么改成 100µs 的定时中断后，MCU 的操作会完全不一样了呢？这是因为定时中断中所要执行的操作，超过 100µs 的原因。一般来说，中断所要处理的操作是比较急迫且必须优先处理的特殊条件，它会强迫主程序的操作停止，直到这些特殊条件都处理完之后，再把 MCU 的控制权还给主程序。

以前一节所介绍的两个程序例来说明：第一个程序，中断服务例程所执行的时间仅占了不到 1ms 的时间，相对于 10ms 的中断间隔，还有绝大多数的时间可以把控制权还给主程序，不会占用系统太多的资源，可以让主程序的操作保持流畅；再看第二个程序，中断执行的时间花了 500µs 以上，远远超过 100µs 的中断间隔时间，因此系统资源一直被中断程序所占用，导致主程序无法顺利执行它所该执行的操作。

因此，当我们在设计定时中断的时候，首先要把握的重点就是中断程序执行的时间要越短越好。中断执行的时间越久，主程序等待的时间就越久，主程序的执行效率就越差。所以，我们整理了下面一些中断设计的原则，让你在编写定时中断时可以更方便。

（1）定时中断的时间间隔，以 2～10ms 之间较为恰当。太频繁的中断次数，会让系统的执行效率变差；中断间隔太长，又会降低系统对紧急信号的敏感度。

（2）中断程序的执行时间，不要超过中断间隔时间的 1/3。比如说 10ms 的定时中断，执行的中断程序要控制在 3ms 以内完成。

（3）规划程序结构时，如果跟时间有相关的操作，要放到定时中断里去执行，因为中断产生一定会造成主程序操作的延迟。

掌握这些原则，就可以让系统更灵活并更有效率。

如果把上一节第二个程序中的调用 DELAY 程序取消掉，程序就正常了，这也就是说在中断程序内尽量要避免程序空转（如 DELAY 占用时间）的写法，这样才能保证程序的正常执行。

【例 10-5】 10-5-1.ADM

```
$MOD51
TMR      EQU        (256-92)        ;100μs 定时值的声明
;
         ORG        0000H           ;MCU 重置后的程序起点
         LJMP       START
         ORG        000BH           ;Timer0 的中断点
         LJMP       T0_ISR
;
         ORG        0030H           ;主程序开始
START:   CLR        P1.0            ;P1.0=0
         MOV        TMOD,#12H       ;Timer0=MODE2
         MOV        TH0,#TMR        ;填入自动更新的 100μs 定时值
         CLR        TF0             ;清除 Timer0 溢位标志
         SETB       TR0             ;开始定时
         SETB       ET0             ;Timer0 的中断功能
         SETB       EA              ;中断开始
LOOP:    CPL        P1.1            ;主程序不断将 P1.1 的电平相反
         SJMP       LOOP
;
T0_ISR:  SETB       P1.0            ;进入中断时将 P1.0 填成 1
         CLR        TF0             ;清除 Timer0 的溢位标志位
;XXXXX   ACALL      DELAY           ;删除此行
         CLR        P1.0            ;离开中断时将 P1.0 填成 0
         RETI
;
DELAY:   MOV        R0,#0           ;时间延迟子程序
         DJNZ       R0,$
         RET
         END
```

10.6 中断种类和使用时机

中断的种类很多，除了刚刚介绍的定时中断，还有接下来要介绍的串行中断。

所谓的串行中断，就是当串行传送与接收数据时，系统会停下来优先把串行传输的操作完成，再执行其他的操作。和定时中断一样，串行中断的规划，是要让系统的运作更具扩展性，更有效率。通过定时中断的程序范例，我

表 10-1 与中断对应的地址表

中断进入点地址	中断源名称
0003H	外部 INT0 中断
000BH	内部计时/计数器 0 中断
0013H	外部 INT1 中断
001BH	内部计时/计数器 1 中断
0023H	串行传输中断

们知道 Timer0 的中断点进入地址是 000BH，那其他中断点的地址在哪里呢？表 10-1 可以让你一目了然。

有了这个列表，在我们写程序的时候，如果要引用这些中断，只要在程序的开始先声明，再指定到对应的子程序去执行就可以了。我们先来看一个中断的范例：

【例 10-6】 10-6-1.ASM

```
$MOD51
MS10    EQU         -9217
TMH     EQU         (MS10)/256
TML     EQU         (MS10) MOD 256
B_RATE  EQU         0FDH
;
        ORG         0000H
        LJMP        START
        ORG         000BH
        LJMP        T0_ISR
        ORG         001BH
        LJMP        T1_ISR
        ORG         0023H
        LJMP        SERIAL
;
START:  MOV         TMOD,#21H
        MOV         TH0,#TMH
        MOV         TL0,#TML
        SETB        TR0
        MOV         TH1,#B_RATE
        SETB        TR1
        MOV         SCON,#50H
        SETB        ET0
        SETB        ET1
        SETB        ES
        SETB        EA
LOOP:   SJMP        LOOP
;
T0_ISR: CLR         TF0
        MOV         TH0,#TMH
        MOV         TL0,#TML
        RETI
;
T1_ISR: CLR         TF1
        RETI
;
SERIAL: CLR         RI
        CLR         TI
        RETI
        END
```

这个程序范例是引用各个中断的标准写法，不过程序中犯了一个很严重的错误，这是我们事先未发现的。

在介绍定时器的时候，我们提到 Timer1 是用来产生串行波特率的专用定时器，所以当 Timer1 设定为串行波特率产生器时，就无法使用 Timer1 的定时中断。

正确的写法应该只留下 Timer0 中断和串行中断：

【例 10-7】10-6-2.ASN

```
$MOD51
MS10        EQU         -9217
TMH         EQU         (MS10)/256
TML         EQU         (MS10) MOD 256
B_RATE      EQU         0FDH
;
            ORG         0000H
            LJMP        START
            ORG         000BH
            LJMP        T0_ISR
            ORG         0023H
            LJMP        SERIAL
START:      MOV         TMOD,#21H
            MOV         TH0,#TMH
            MOV         TL0,#TML
            SETB        TR0
            MOV         TH1,#B_RATE
            SETB        TR1
            MOV         SCON,#50H
            SETB        ET0
            SETB        ES
            SETB        EA
LOOP:       SJMP        LOOP
;
T0_ISR:     CLR         TF0
            MOV         TH0,#TMH
            MOV         TL0,#TML
            RETI
;
SERIAL:     CLR         RI
            CLR         TI
            RETI
            END
```

或者是留下 Timer0 和 Timerl 中断：

【例 10-8】10-6-3.ASN

```
$MOD51
MS10        EQU         -9217
TMH         EQU         (MS10)/256
TML         EQU         (MS10) MOD 256
RATE        EQU         (256-92)
;
            ORG         0000H
            LJMP        START
            ORG         000BH
            LJMP        T0_ISR
            ORG         001BH
            LJMP        T1_ISR
;
START:      MOV         TMOD,#21H
            MOV         TH0,#TMH
            MOV         TL0,#TML
```

```
            SETB        TR0
            MOV         TH1,#RATE
            SETB        TR1
            SETB        ET0
            SETB        ET1
            SETB        EA
LOOP:       SJMP        LOOP
;
T0_ISR;     CLR         TF0
            MOV         TH0,#TMH
            MOV         TL0,#TML
            RETI
;
T1_ISR:     CLR         TF1
            RETI
            END
```

10.7 程序模块 1——定时中断

从范例程序学习中断是最快的方法，例 10-9 是每隔一秒就把 CNT 缓冲区的显示值加 1，并从 P1.0 显示的程序。

【例 10-9】 10-7-1.ASM

```
$MOD51
MS10        EQU         -9217
TMH         EQU         (MS10)/256
TML         EQU         (MS10) MOD 256
CNT         DATA        030H
            ORG         0000H
            LJMP        START
            ORG         000BH
            LJMP        T0_ISR
            ORG         0030H
START:      MOV         CNT,#0
            MOV         R0,#100
            MOV         TMOD,#11H
            MOV         TL0,#TMH
            CLR         TF0
            SETB        TR0
            SETB        ET0
            SETB        EA
LOOP:       SJMP        LOOP
T0_ISR:     MOV         TL0,#TML
            MOV         TH0,#TMH
            DJNZ        R0,EXIT
            MOV         R0,#100
            INC         CNT
            MOV         A,CNT
            CPL         A
            MOV         P1,A
EXIT:       RETI
            END
```

通过定时中断的写法，可以让原先通过程序重新载入计时值的误差大大降低，从示波器观看 P1.0 的波形时，信号是每秒钟规则地变化着，而 LED 的显示也每秒更新一次。

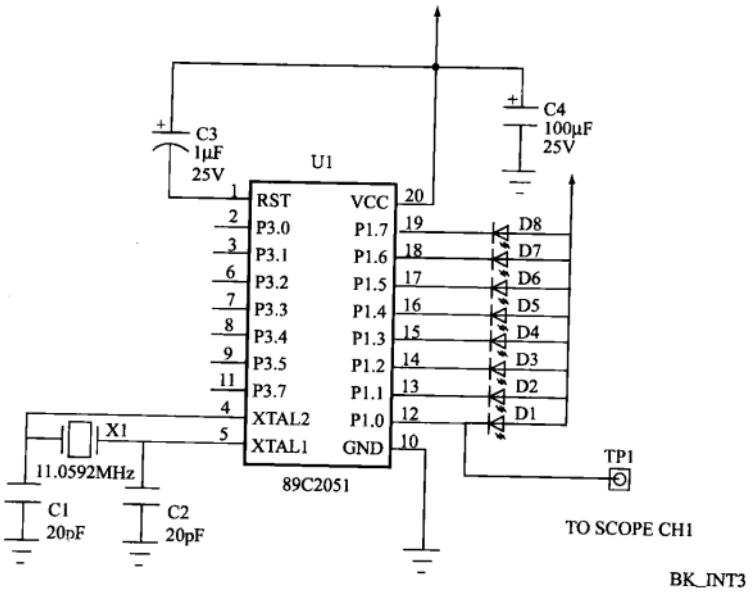


图 10-8　除了 8 个 LED 外，P1.0 端也接到示波器的 CH1 输入点

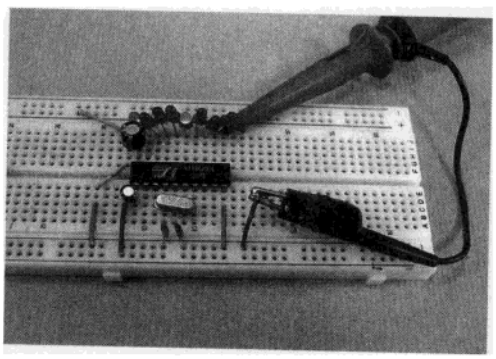


图 10-9　将程序编译好烧录到 MCU 中，看到 P1 端口的八个 LED 灯正以 1Hz 的速度更新显示

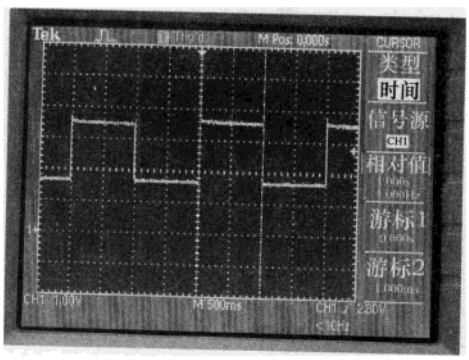


图 10-10　从示波器上看到 P1.0 的信号，电平的变化正好是每秒一次

此外，我们在 T0_ISR 这段中断程序中并没有 CLR TF0 的操作，但是系统还是准时地产生中断信号，这是因为中断产生的时候，MCU 会自动将 TF0 清成 0 的原因，因此我们就不必要再多花一个指令时间去执行 CLR TF0 的操作了。

10.8　中断范例 2——用定时中断更新显示

在使用定时中断时，尽量把跟时间有关的操作放到中断中去执行，没有时间上迫切需要

的操作，就放在主程序区里执行。我们来看看如下范例：

【例 10-10】 10-8-1.ASM

```
$MOD51
MS10        EQU         -9217
TMH         EQU         (MS10)/256
TML         EQU         (MS10) MOD 256
KEY         BIT         P3.4
CNT         DATA        030H
;
            ORG         0000H
            LJMP        START
            ORG         000BH
            LJMP        T0_ISR
;
            ORG         0030H
START:      SETB        KEY
            MOV         CNT,#0
            MOV         TMOD,#11H
            MOV         TL0,#TML
            MOV         TH0,#TMH
            CLR         TF0
            SETB        TR0
            SETB        ET0
            SETB        EA
LOOP:       JB          KEY,LOOP
            JNB         KEY,$
            INC         CNT
            ACALL       DELAY
            SJMP        LOOP
;
DELAY:      MOV         R0,#100
L1:         MOV         R1,#0
            DJNZ        R1,$
            DJNZ        R0,L1
            RET
;
T0_ISR:     MOV         TL0,#TML
            MOV         TH0,#TMH
            MOV         A,CNT
            CPL         A
            MOV         P1,A
            RETI
            END
```

这个程序是通过 10ms 的定时中断来更新 LED 的显示操作，每当我们按一下按键时，CNT 的值会加 1，而定时中断每隔 10ms 会把 CNT 的值送到 LED 上显示。假如我们快速连续按两下按键，却发现显示只增加 1，这是因为我们在程序中加入按键操作的判断，每按一下一定要经过一小段时间的延迟才可以再按下一个。

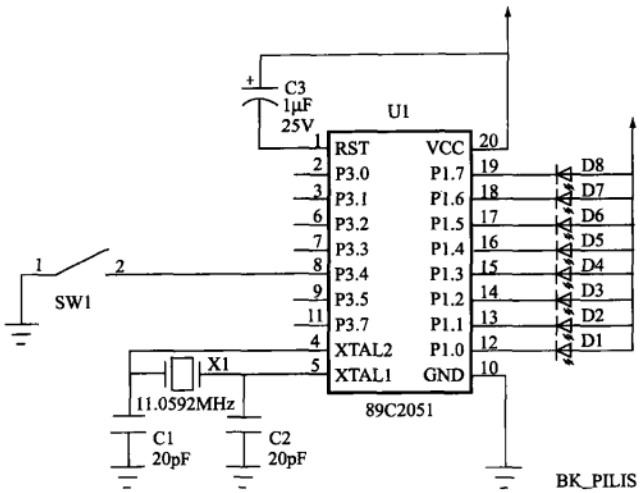


图 10-11　测试用线路图，SW1 按键每按一下，LED 灯的状态就会改变一次

10.9　中断范例 3——串行中断

例 10-11 这一段程序在示范利用串行中断传送一个固定码 31H 的范例。

【例 10-11】 10-9-1.ASM

```
$MOD51
        ORG         0000H
        SJMP        START
        ORG         0023H
        SJMP        SERIAL
;
        ORG         0030H
START:  MOV         TMOD,#21H
        MOV         TH1,#0FDH
        SETB        TR1
        MOV         SCON,#50H
        SETB        ES
        SETB        EA
        SETB        TI
STOP:   CPL         P1.0
        SJMP        STOP
;
SERIAL: CPL         P1.1
        JNB         TI,EXIT
        CLR         TI
        MOV         A,#31H
        MOV         SBUF,A
        ACALL       DELAY
EXIT:   CPL         P1.1
        RETI
;
DELAY:  MOV         R0,#0
```

```
JNZ         R0,$
RET
END
```

主程序部分没做什么事，只是一直把 P1.0 的状态反相。串行中断时，刚进入串行中断服务程序时，会先把 P1.1 反相，而离开中断前还会再把 P1.1 反相一次，所以只要观察 P1.1 的时间就可以知道执行串行中断所花的时间。另一方面串行中断时会从串行端口送出 31H 的固定值，这个状态就出现在 P3.1 上。

在这个程序范例中，我们要观察 P1.0(CH2)、P1.1(CH3)和 P3.1(CH1)的电平变化，其中，CH1 是串行信号，CH2 是系统操作，CH3 是中断执行时间。从示波器上的信号来看，我们发觉当串行数据在传送时并不会与主程序的操作发生干扰，因为串行传输的操作是由 MCU 的硬件自行控制的，但中断执行的时间越久，对主程序的影响就越大，因此在编写中断程序时，还是要把握执行时间越短越好的原则。

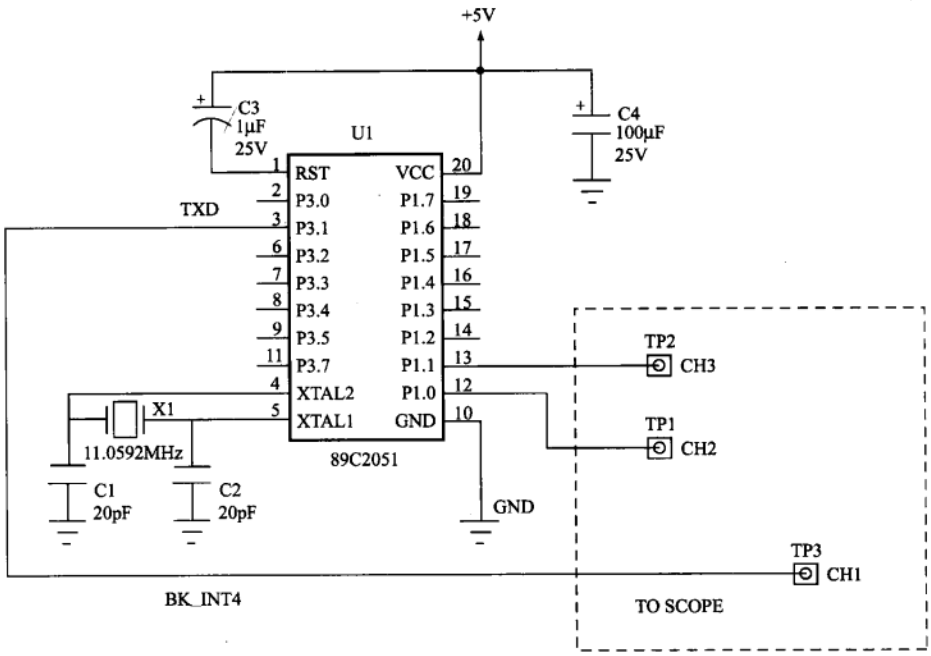


图 10-12　串行中断的观察，在这里我们同时监视三组数字状态

图 10-13 中解释如下：

P3.1（CH1）由串行端口所送出固定的 31H 串行数据串。

P1.0（CH2）主程序部分，一进入中断后就停止反相。

P1.1（CH3）每次中断所花的时间，由图中看来中断约耗掉 500μs 的时间。

比较 CH2 与 CH3，当中断发生时（CH3 电平下降），主程序的操作就停止了（CH2 停止变化），但是串行传输的部分，则完全不会干扰主程序操作，在串行中断一发生不久后，就固定把数据送出去（CH1 的信号变化，都在 CH3 的电平下降不久就产生）。

由于串行传输在下一章才会介绍，因此在这里仅对中断的效率做分析，等到我们对串行传输的基本原理有了初步的了解后，可以再回来探讨这个操作的流程，应该会有更深刻的体验。

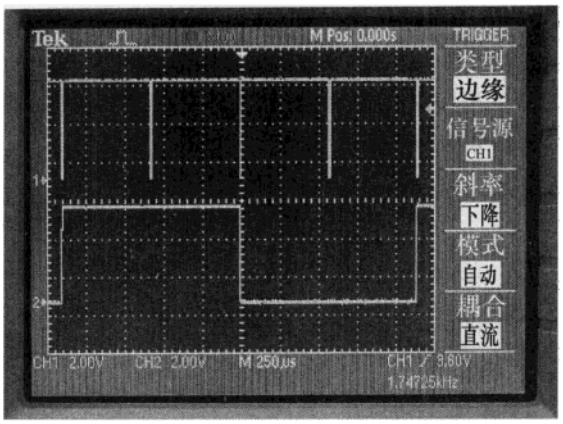


图 10-13　用示波器观察 P1.0、P1.1 和 P3.1（由上到下）的重要波形

10.10　随时将待测状态值送到 P1 上

在编写中断程序时，我们强烈建议使用 P1 当作除错的参考依据，这比任何软件除错的工具要实用得多。本书大多数的程序范例，都是通过 P1 端口来显示状态值，不管是执行时间的确认，或是寄存器的数值，只要善用 P1 端口，就可以省去很多除错的时间。例 10-12 是一个用 P1 端口显示串行中断所接收数据的例子。

【例 10-12】 10-10-1.ASM 接收端

```
$MOD51
CNT     DATA        030H
;
        ORG         0000H
        SJMP        START
        ORG         0023H
        SJMP        SERIAL
        ORG         0030H
START:  MOV         CNT,#0
        MOV         TMOD,#21H
        MOV         TH1,#0FDH
        SETB        TR1
        MOV         SCON,#50H
        SETB        ES
        SETB        EA
STOP:   MOV         A,CNT
        CPL         A
        MOV         P1,A            ;把收到的数据反应在 P1 端口上
        SJMP        STOP
;
SERIAL: JNB         RI,EXIT
        CLR         RI
        INC         CNT
EXIT:   RETI
        END
```

有了接收程序，还需要一个传送程序，例 10-13 为我们提供了一个可以传送 99 字节的传送程序，你可以烧录到 MCU 里做测试，看看 P1 显示的是不是 99(63H)。当然你还可以用刚刚介绍过的重复传送 31H 的程序来观看，不过因为重复传送的操作是不会停止的，所以你所观看到的接收数据量是一直变动的。

【例 10-13】 10-10-2.ASM 传送端

```
$MOD51
        ORG         0000H
        SJMP        START
        ORG         0023H
        SJMP        SERIAL
;
        ORG         0030H
START:  MOV         R2,#100
        ACALL       DELAY
        ACALL       DELAY
        MOV         TMOD,#21H
        MOV         TH1,#0FDH
        SETB        TR1
        MOV         SCON,#50H
        SETB        ES
        SETB        EA
        SETB        TI
STOP:   SJMP        STOP
;
SERIAL: JNB         TI,EXIT
        CLR         TI
        DJNZ        R2,SEND
        SJMP        EXIT
SEND:   MOV         A,#31H
        MOV         SBUF,A
EXIT:   RETI
;
```

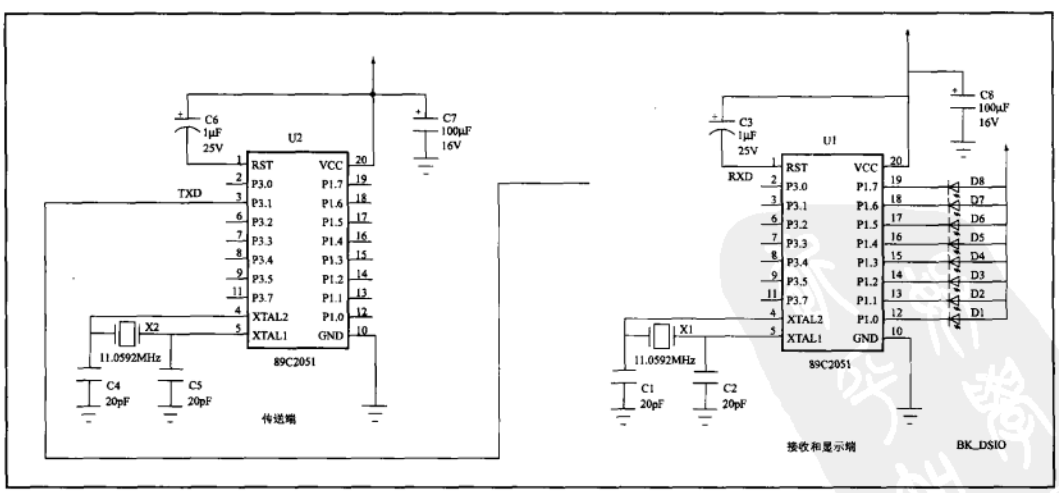


图 10-14　两个 AT89C2051 同时工作，左边的 MCU 一直从 P3.1 上送出 31H 的串行数据右边的 MCU 则把从 P3.0 串行端口收到的数据反应在八个 LED 上

两个 MCU 要能相互传递串行数据还是要有一些前提的：

（1）相同的供应电压。

（2）GND 接在一起。

（3）相同的传输速度（Baud rate）。

（4）串行数据传送端由 TxD（P3.1）送出，接收端由 RxD（P3.0）接收。

（5）MCU 之间的互相协调以免漏接数据，这方面要靠程序配合。

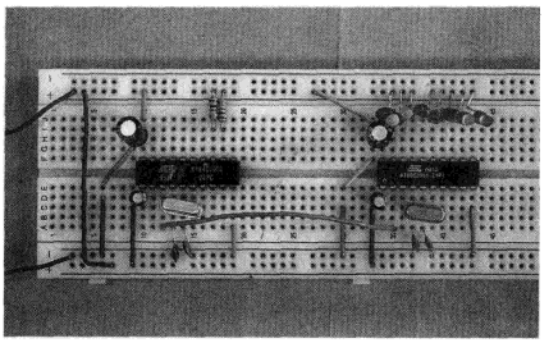


图 10-15　通过 P1 得到串行接收到的字节数为 99 个（63H）

线路配接的方式，请参考下一章串行接收的线路配接法

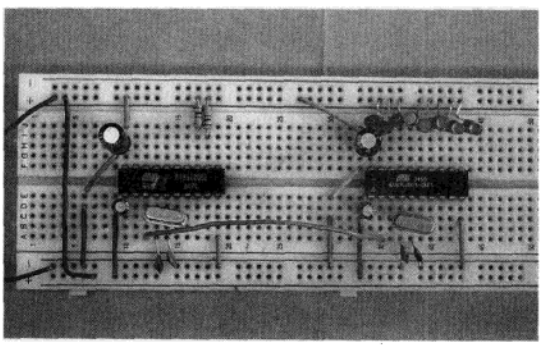


图 10-16　如果传送程序改用前一节所介绍的重复传送，显示值会一直变化

```
DELAY:  MOV     R0,#0
L1:     MOV     R1,#0
        DJNZ    R1,$
        DJNZ    R0,L1
        RET
        END
```

我们发现，通过 P1 端口外接 LED 的方式，可以帮助我们观察很多有用的信息。所以在编写程序时，多利用这样的方式进行除错，绝对是有显著效果的。

中断介绍到此算告一个段落，如果您对中断想了解得更多，可以参考一下由旗标公司所出版的《8051 单片机彻底研究——基础篇》一书。

您可以从下列公司的网站取得更进一步的信息：

（1）www.chipware.com.tw：查询单片机相关应用的参考资料。

（2）www.flag.com.tw：查询“8051 单片机彻底研究”系列丛书的相关介绍。

（3）www.tektronix.com：查询示波器的相关资料。

（4）www.atmel.com：制造 8051 的厂商，有很多 MCU 的相关参考资料。

（5）www.intel.com：查询 805I（MCS51）的相关介绍。

11 串行通信

本章要介绍的是 8051 的串行通信，它是 8051 与 PC 或仪器设备沟通的重要桥梁。要写一个 8051 的串行通信程序并不难，但是要写一个好的串行通信程序却需要注意很多环节。本章的内容着重在程序编写上的技巧说明，列举初学者最容易混淆或忽略的串行通信设定方式，并学会以示波器除错的方法，养成用示波器看波形并除错的习惯，如果你不会用示波器看串行数据的话，说真的，你对 8051 单片机的理解顶多算一半。

第 11 章　串行通信

11.1　送出一个串行数据的程序范例

串行通信是 8051 中很好玩也是很重要的部分，它是 8051 与外界沟通的桥梁，通过下面实际的例子，了解一下串行通信在工作时的情况。下面是利用串行通信送出一段字符串的范例。

【例 11-1】 11-1-1.ASM

```
$MOD51
        ORG         0000H
START:  MOV         TMOD,#20H           ;设定定时器的模式
        MOV         TH1,#0FDH           ;设定传送的波特率 9600b/s
        SETB        TR1                 ;定时开始
        MOV         SCON,#50H           ;设定串行传输的模式
        SETB        P1.0                ;设定观察对照点
;
LOOP:   CLR         TI                  ;清除 TI 标志位
        MOV         A,#31H              ;选择要送出的码
        CPL         P1.0                ;传送起始的参考点
        MOV         SBUF,A              ;从串行通信专用的寄存器送出
;
WAIT:   JNB         TI,WAIT             ;等待传送完成
        CPL         P1.0                ;传送完成的参考点
        ACALL       DELAY               ;延迟的时间
        SJMP        LOOP                ;继续传送同样的码来观察
;
DELAY:  MOV         R0,#00H             ;延迟用子程序
        DJNZ        R0,$
        RET
        END
```

如果您是第一次写串行通信的程序，我们建议您对本章的所有程序范例一定要全部试过一次，这样才会对 8051 的串行通信有全面的理解。一开始我们先来看看如何通过 8051 的串行通信端口 SBUF，将数据源源不断地发送出去，整个过程的确有点复杂，但弄懂后就不会不知所措了。一定要先确认送的数据是对的，稍后再探讨如何接收串行的数据。

有了传送的范例程序后，该怎样观察传送出来的数据呢？最好的方法是用示波器，而观察点是 MCU 上的 P3.1(TxD)及 P1.0，也就是 AT89C2051 的第 3 引脚跟第 12 引脚，第 3 引脚是串行传输所送出的数据，而第 12 引脚则是用来对照传输的开始和结束用的，下面是通过示波器所观察到的波形。

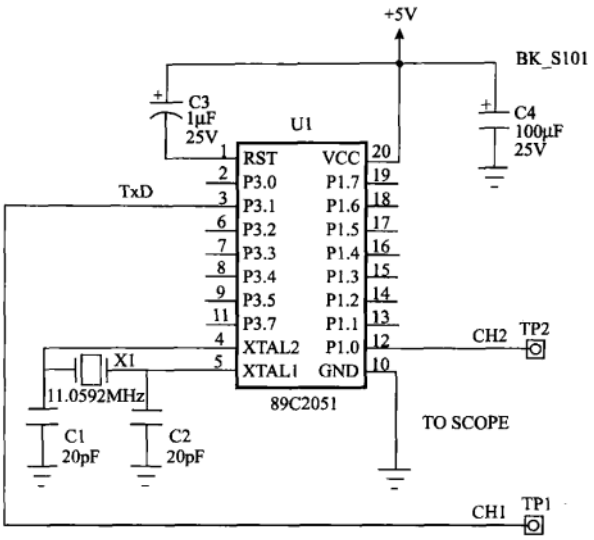


图 11-1　串行通信所安排的实验线路图

图 11-1 中，在电路的安排上共有两个信号要接往示波器上，P3.1 接示波器的 CH1，直接观察串行数据送出的情况。P1.0 接 CH2，代表传送一笔串行数据所要花的时间。请使用数字式示波器以便观察数字信号的变化。

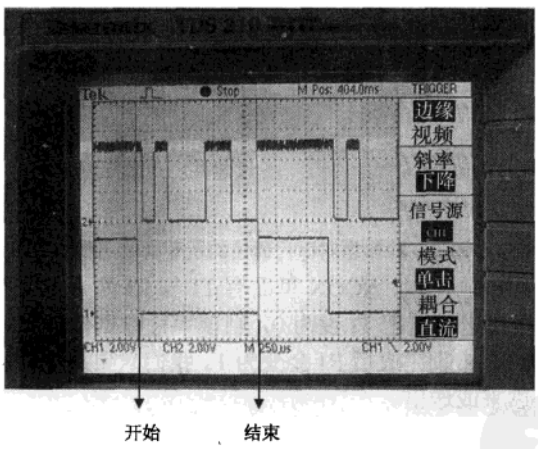


图 11-2　示波器的信号图

图 11-2 中，示波器捕捉到由 TxD 引脚所送出来的信号，上方波形显示的数据 P3.1 送出来的信号 31H，而下方显示的波形是由 P1.0 所送出。屏幕中的横向代表时间轴（目前状态为 250μs/DIV），纵向代表电压电平（目前状态 2.00V/DIV）

P1.0 由 1 变成 0 时正好开始传送数据，而由 0 变成 1 时代表串行数据刚好传完，同样的时间内 P3.1 引脚也从开始执行传送 31H 串行数据的操作。

串行传输的基本操作，在这个程序中做了最完整且简单的示范，按程序的内容可以分为两个部分：第一个部分是从 START 开始到分号结束，这一段程序是用来设定 8051 串行传输

的波特率和传送格式；第二个部分是从 LOOP 开始一直到 END 结束，其功能是用来连续传送同一个字节的数据，程序上所示范的是送出 31H。

8051 的串行通信绝对不是一行 MOV SBUF，A 就交待完了，还有一大堆寄存器的设定要弄清楚的。我们的问题来了：程序中第一段的设定值是怎么来的？什么是波特率？串行的传送格式有哪些？如果一次要传送好几个字节的数据该怎么传送？示波器上的信号该怎么解读？有没有其他方法可以接收这些串行信号？要知道这些问题的答案，先要知道串行传输在 8051 中是如何实现的。

11.2 串行通信有关的寄存器 SCON 和 SBUF

在 8051 中，有两个专门提供作为串行通信用的寄存器，一个叫 SCON，另一个叫 SBUF。SCON 所掌管的是通信格式的设定，而 SBUF 则是传输数据的缓冲区，先来看看 SCON 有哪些功能吧！

(MSB)							(LSB)
SM0	SM1	SM2	REN	TB8	RB8	TI	RI
位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0

图 11-3 SCON 寄存器的内容和各个位的名称

SCON 最主要的功能就是设定串行通信的格式，以及串行数据的传输控制。它必须搭配定时器 Timer1 才能顺利完成串行通信的设定。如果要使用串行通信，我们该怎么设定才好呢？答案就在 11-1-1.ASM 的 START 部分，只要依照这样设定，我们就可以进行串行通信了。

```
START:  MOV     TMOD,#20H       ;设定定时器
        MOV     TH1,#0FDH       ;设定波特率 9600 b/s
        SETB    TR1             ;开始定时
        MOV     SCON,#50H       ;设定并启动串行通信
```

注意事项：

此段设定不建议初学者自己修改，除非您对串行通信的所有功能已经十分熟悉。

再来看看串行数据的传输控制。SCON 里面有 TI 与 RI 两个位，它们是串行数据传输的控制点，只要在串行数据传送前先把 TI 设为 0，则该串行数据传送完成后 TI 会变为 1；在接收串行数据前把 RI 设为 0，接收完成后 RI 会变为 1。我们只要用程序检查 RI 或 TI 位就可以知道数据串已经完整地进来或送出。

回头看看 11-1-1．ASM 这个程序里 LOOP 部分的片段：

```
LOOP;   CLR     TI              ;清除 TI 标志位
        MOV     A,#31H          ;选择要送出的码
        CPL     P1.0            ;传送起始的参考点
        MOV     SBUF,A          ;从串行通信专用的寄存器送出
WAIT;   JNB     TI,WAIT         ;等待串行传送完成
```

其中的 P1.0 是为了配合示波器观看用的，在实际的应用场合可以省略，但是请一定要学会用这个方法进行除错和波形观察。

这就是判断串行数据是否传输完成的做法，而 RI 的用法与 TI 相同，等后面介绍到接收

时再来探讨吧。

如果我们要改变 SBUF 的输出值，应该在哪里做修改呢？请先找到 11-1-1.ASM 中 MOV SBUF，A 这行指令，我们知道输出的值是由累加器传至 SBUF 的，再往上看 MOV A，#31H，原来累加器的值是用立即寻址的方式送进去的。所以，若我们想改变 SBUF 的输出状态，只要将#31H 改成我们需要的值就可以了，如果现在输出的串行数据要改为 58H，那程序的变动部分只是将 MOV A，#31H 这行改变成 MOV A,#58H。

我们看看程序更改后，波形是不是又不一样了呢？

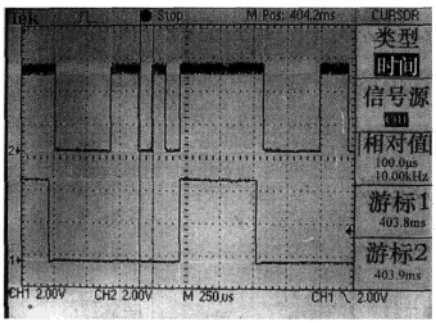


图 11-4　上面的波形是 P3.1 送出来串行数据 58H，而下方显示的波形是由 P1.0 送出

果然跟以前看到送 31H 信号时的波形不大一样，下方是由 P1.0 所产生的波形，有两个凹型波形：当信号开始向下凹，代表数据传送开始；当信号向上回到原来的水平位置时，代表信号传送结束，之所以会看到重复出现凹型信号波形，这是因为程序的循环设定的关系，同时也方便我们观察用，因此每看到下方出现一个凹型波形，就代表串行端口 SBUF 又完成一笔数据的传送了。请对照着看上方 P3.1 的波形，在 P1.0 的凹型时间间隔里，是不是每次的波形都一样？

图 11-4 上方传的是 58H 的数字数据串，你看得出来吗？很难，真的很难看出其中的情况。其实，串行数据何时开始产生是很难去指定的，所以串行端口在送出真正的第 1 位数据前，它会先传送一个数字 0 的信号，所以在波形上我们会先看到 LOW(0)的状态，这个 0 的信号我们称为“Start Bit”，通知接收端它要开始传送第一位的数据。串行数据结束时也会有一位的时间，其状态一定是 1，以便通知接收端数据已经传送完了，这个位就称为“Stop Bit”。所以 8051 的串行端口传送 1 个字节数据时，实际上却是送出 10 位的数据串。

对照图 11-4 看，来 P3.1 较 P1.0 开始的时间较晚了一点，这是因为编写程序时，CPL P1.0 在 MOV SBUF,A 的前面，所以执行时当然就会有先后的差别，但结束的时间就几乎一样了。

11.3　串行波形的观察

示波器显示的波形，到底哪些才是传送出来的信号，这些信号要如何去观察和判断呢？

要使用示波器观察前，我们需要先将下面的元器件准备好：

（1）面包板。

（2）烧录好 11-1-1.ASM 程序范例的 AT89C2501 芯片。

（3）1μF 50V 和 10μF 50V 电解电容 X1。

（4）10kΩ 电阻 X1。
（5）11.0592MHz 石英晶振 X1。
（6）20pF 陶质电容 X2。
（7）跳线数条。

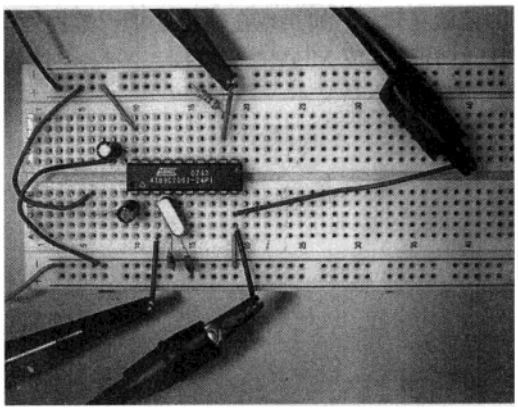


图 11-5　跳线接法图

图 11-5 中，我们预留出来六条跳线，除了最左方两条是预留给电源供应器供应+5V，其他四条两两为一组，分别接上两个测棒，由示波器 CH1 接出来的测棒正接 P1.0 接出来的跳线，另一端接 GND 接出来的跳线，由示波器 CH2 接出来的测棒，正接 P3.1(TxD)引脚接出来的跳线，另一端接 GND 接出来的跳线。

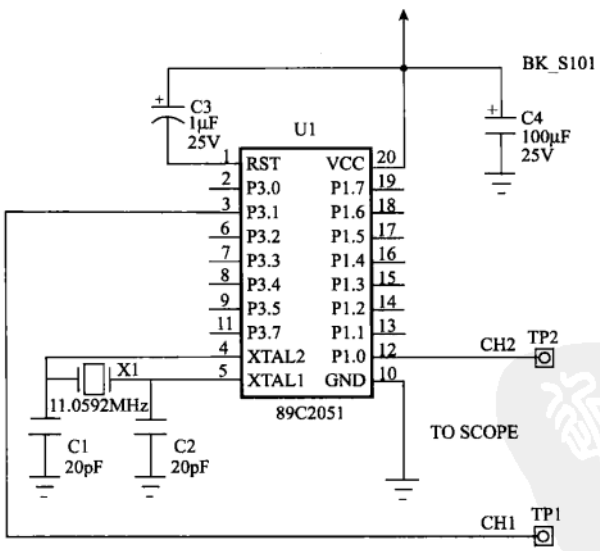


图 11-6　线路图

在示波器上除了看 P1.0 的波形外，最重要的就是传送数据的波形，也就是由 P3.1 送出来的波形。到底要由哪边看起，这一串波形中哪些信号是我们传送出去的，我们又可以从示波器上得到哪些信息呢？

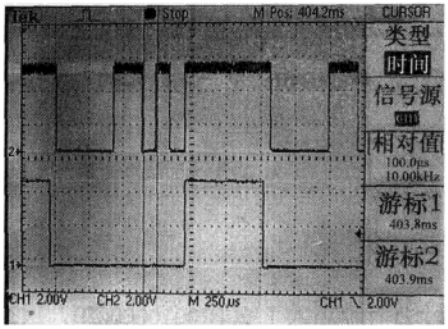


图 11-7　上方的波形是 P3.1 引脚传送 31H 所产生的形状，而下面是 P1.0 传送出来的波形，平时没有传送数据，波形会一直在 High 状态

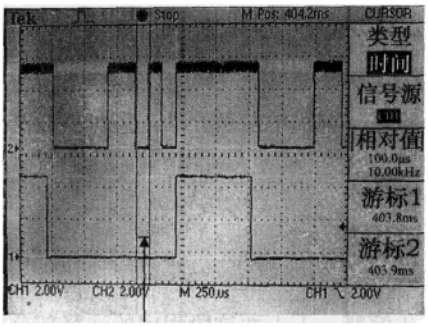


图 11-8　上方的波形是 P3.1 引脚传送 58H 所产生的形状，而下方一样是 P1.0 传送出来的波形，主要用来对照 P3.1 送出数据开始和结束的时间

示波器记录波形是由左向右存放的，所以靠左边的波形传送的时间较早，在观察波形前先做以下设定：垂直电压轴设定成 2V/DIV，而横向的时间轴设定成 250μs/DIV。P1.0 的波形由开始到结束，一共占 4 大格（1 大格=5 小格）。而 P3.1 和 P1.0 两个引脚的执行时间因为程序先后顺序不同大约差 1 小格，所以 P3.1 引脚从开始到结束约占 3 大格 4 小格，一次执行完的时间约 lms。那么 31H 和 58H 的波形该怎么判断呢？

58H=01011 000B，串行端口传送由低位开始，所以为“00011010”，再加上 Start 位和 Stop 位两个增添的位，最后在示波器上看到的波形应该为“0 00011010 1”共 10 个状态：

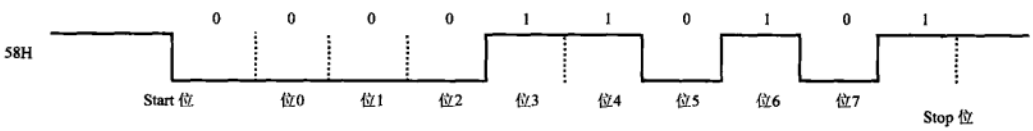


图 11-9　波形图

31H=0011 0001B，再加上 Start 位，所以我们在 31H 的串行数据传送中会看到“0 1000 1100 1”，如图 11-10 所示。

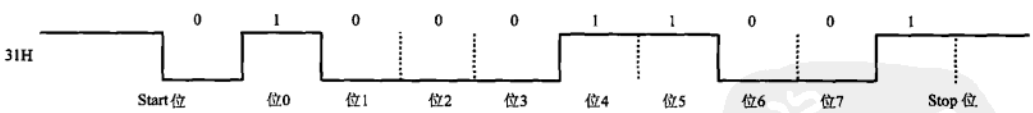


图 11-10　波形图

在传送数据的开始前和结束后，P3.1 即 8051 的 TxD 会保持在 High 状态，这个状态在看 P1.0 的波形时最明显，因为屏幕上 P1.0 总共产生两个凹型波形，各代表执行一次完整的程序，而两个波形间，有约两大格的空间处在 High 状态，这就是在等待传送前的状态。

到底为何利用 P1.0 来做起始的对照点呢？如果今天我们传送的数据为 F0H，这时的 Start 位和 Stop 位就很容易被我们忽略。因为 F0H 会送出“0 0000 1111 1”的波形，这时对照用的 P1.0 就可以从旁协助我们观察波形的开始与结束点，所以不能因为偷懒而把 P1.0 对照的操作给省略了。

11.4 发送程序模块说明（一）

看过了前面的范例介绍，您可能会产生一些疑问，串行的设定值是怎么来的？又该如何设定呢？

```
MOV     TMOD,#20H       ;设定定时器                    (1)
MOV     TH1,#0FDH       ;设定波特率 9600b/s            (2)
SETB    TR1             ;开始定时                      (3)
MOV     SCON,#50H       ;设定并启动串行通信            (4)
```

先解释一下上面这段程序的注释部分：

（1）在设计定时器这一行，我们将 TMOD 这个寄存器填入了 20H(00100000B)，这个操作就是告诉 MCU，我们要使用 Timer1 来控制串行传输。

（2）所谓的波特率（Baud Rate），指的是单一位串行传输的速率，单位是 b/s(bit per second)，也就是每秒钟传送的位数，而 9600b/s 指的就是每秒钟传送 9600 位的传输速度。要设定这个波特率的前题是我们要选用频率为 11.0592MHz 的石英晶振，并且在 TH1 填入 FDH 的值，这样 MCU 就知道要以 9600b/s 的速度来传送了。至于为什么要这样设定，等正式介绍波特率的时候，我们再来深入了解。

（3）开始定时这一行，更正确的说法应该是启动定时器 Timer1。此时的 Timer1 也可以称为波特率发生器。

（4）万事具备，只欠东风。当我们把所有串行传输的条件都设定好之后，别忘了要把串行传输的功能启用。这一行的操作，除了设定串行传输的模式外，还包含了启用串行功能的操作。关于串行传输的模式，我们在稍后的章节会有更详尽地介绍。

有了这个串行传输设定模块后，我们就可以轻易地在程序中引用串行传输功能了。

使用时机：

凡是所有需要使用串行通信的场合，都可以使用这个程序模块来做设定操作，这里示范的是传输速率 9600b/s 的写法。

使用限制：

因为使用串行通信会占用 Timer1 的资源，并且一定要通过 P3.0 和 P3.1 来进行串行传输操作，所以如果使用这个设定模块后，就不能再使用 P3.0、P3.1 和 Timer1 作为其他的用途，否则很可能会造成串行传输的异常或系统的错误操作。

11.5 发送程序模块说明（二）

设定好串行传输的条件，再来看看如何传送数据。

```
CLR     TI              ;清除 TI 标志位                (1)
MOV     A,#31H          ;选择要送出的码                (2)
MOV     SBUF,A          ;从串行通信专用的寄存器送出    (3)
JNB     TI,$            ;等待传送完成                  (4)
```

程序注释说明：

（1）TI 标志位是串行通信功能中管理数据传送的控制点。当 MCU 将数据传输完成后，TI 标志位会被设成 1，不过，设成 1 之后并不会自动恢复成 0 的状态。因此，当我们要传送串行数据的时候，第一项操作就是要先把 TI 标志位设成 0，这样才能够通过 TI 标志来确认我们所要传送的数据是否已经完整送出。

（2）串行传输一次只能传送一个字节，在传送前要先把数据放在 ACC 累加器中，这里我们所要送出的码是 31H，如果要传送其他的码，只需在这里更改就好了。实际应用时的写法不会如此简单，应该会更复杂一些。

（3）要传送的数据确定填入 ACC 累加器之后，就可以通过 SBUF 把数据送出去。

（4）前面提到 TI 标志会在串行数据传送完成后被填成 1，如果数据还没送出，TI 会是我们一开始设定的 0。此时通过一个 JNB TI,$的操作，强迫 MCU 在这里等待，直到串行数据完整送出后才继续下一项操作。

使用时机：

传送串行数据。

使用限制：

要把 P3.1(TxD)空出来专门给串行传输用。

11.6 波特率产生器

介绍了两个串行通信的程序模块后，我们来了解一下波特率是怎么产生的。波特率（Baud Rate）就是串行通信的位传输速率，串行通信如果没有设定波特率，在通信时 MCU 就无法判断数据各个位的时间间隔为多久，不知道数据何时开始、何时结束，也就无法正确传输串行数据。因此波特率的设定，对串行通信来说是相当重要的一环。

波特率的产生方式，最常用的就是由 MCU 的定时器 Timer1 来控制，只要波特率的定义正确，就可以轻易地跟计算机系统或仪器设备连接。

Timer1 的定时跟石英晶振的振荡频率有关，标准的串行通信会使用 11.0592MHz 的石英晶振作为 MCU 的振荡频率。

问题来了：石英晶振为什么要选 11.0592MHz 呢？

从串行通信的硬件框图来看，波特率的计算推导出的公式：

$$\begin{aligned}\text{波特率}&=\frac{1}{32}\times\text{Timer1的溢位率}\\&=\frac{1}{32}\times\frac{\text{石英晶振振荡频率}}{12\times(256-\text{TH1设定值})}\end{aligned}$$

根据前面的程序范例 1，TH1 的设定值为 253(FDH)，石英晶振的振荡频率为 11.0592MHz，我们来推导一下传输的波特率应该：

$$\begin{aligned}\text{波特率}&=\frac{1}{32}\times\frac{11059200}{12\times256-253}\\&=\frac{1}{32}\times\frac{11059200}{36}=9600\,\text{b/s}\end{aligned}$$

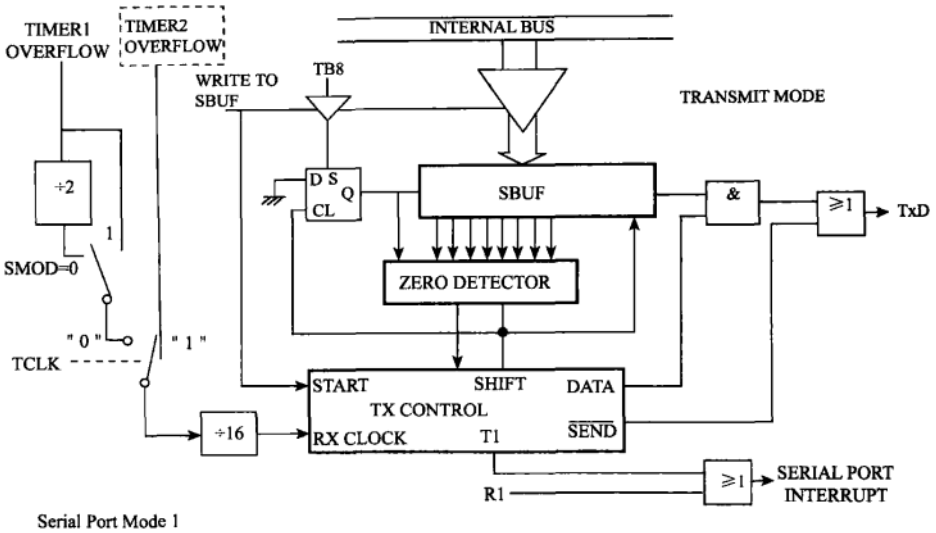


图 11-11　串行通信框图

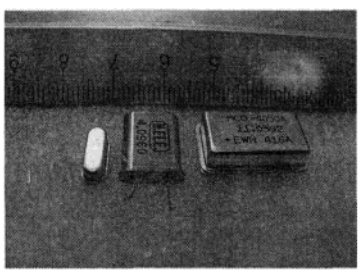


图 11-12　几种常见的石英晶振外观，右边的是所谓的 OSC 振荡器，直接通电就可以产生方波

看到了吗？11.0592MHz 这个数值是常用通信波特率（19200、9600、4800、2400、1200）的公倍数，这并不是随机挑选的。

如果 MCU 所应用的场合必须与计算机或其他仪器连线时，请记得一定要选用 11.0592MHz 的石英晶振。假如 MCU 可以支持较高的振荡频率（如 24MHz 或 40MHz），我们也可以选用多一倍的 22.1184MHz 石英晶振，只要把 TH1 的设定值修正一下就可以了。

图 11-13 是常用波特率的列表：

波特率（b/s）	振荡频率（MHz）	TH1 设定值
9600	11.0592	FDH
4800	11.0592	FAH
2400	11.0592	F4H
1200	11.0592	E8H
19200	22.1184	FDH
9600	22.1184	FAH
4800	22.1184	F4H
2400	22.1184	E8H
1200	22.1184	D0H

图 11-13　使用 11.0592MHz 石英晶振时，常用的波特率值与 TH1 设定值

假设 MCU 只能支持 11.0592MHz 的石英晶振(20MHz 以下),可是串行通信却需要 19200 b/s 的波特率，有没有可以变通的方式呢？

在 8051 中另外还有一个 SMOD 标志位，它位于 PCON 寄存器中，当系统重置（RESET）时，SMOD 会被填成 0，此时串行的速率就同上面的列表一样。如果将这个标志设成 1，则串行传输的速率会加快一倍，原本 9600b/s 的波特率，就会变成 19200b/s 了。

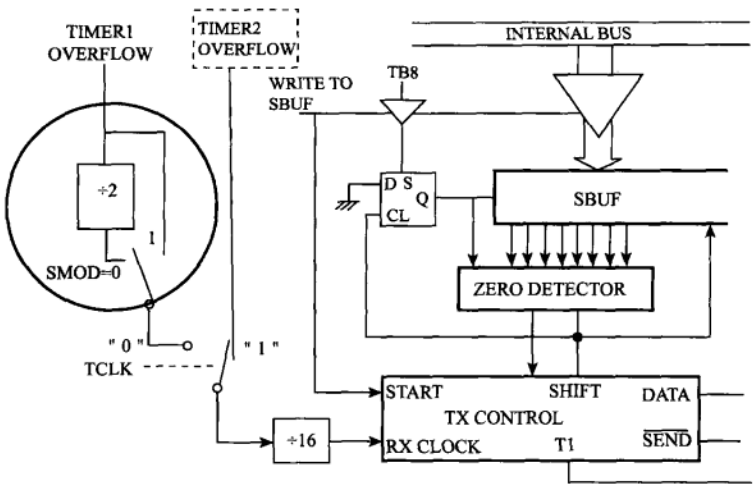


图 11-14　设定 SMOD 位为 1 就可以让传输速率增加 1 倍达到 19200b/s

11.7　SCON 寄存器

前面提到，SCON 是控制串行通信很重要的寄存器，究竟 SCON 有哪些功能呢？

SCON.0（RI）、SCON.1（TI）：原来我们用来判断串行接收与传送的两个重要标志位，就是 SCON 的成员之一，分别占用了位 0 与位 1，所以 SCON 的主要功能，包含了串行传输的控制。

SCON.4（REN）：这个标志位是用来启用串行接收功能，如果填成 1，则串行接收才启用；填成 0 就无法接收串行数据了。

SCON.6（SM0）、SCON.7（SM1）：这两个位是用来设定串行传输模式的，一般最常用的就是模式一（SM0=0，SM1=1）。

所谓的串行传输模式一，就是指串行数据以最常见的 8 位格式传送，而且波特率可以通过程序设定。这一章所提到用到的设定波特率，都是在模式一的条件中使用。

（MSB）　　（LSB）

SM0	SM1	SM2	REN	TB8	RB8	TI	RI
位 7	位 6	位 5	位 4	位 3	位 2	位 1	位 0

图 11-15　SCON 寄存器中各位定义

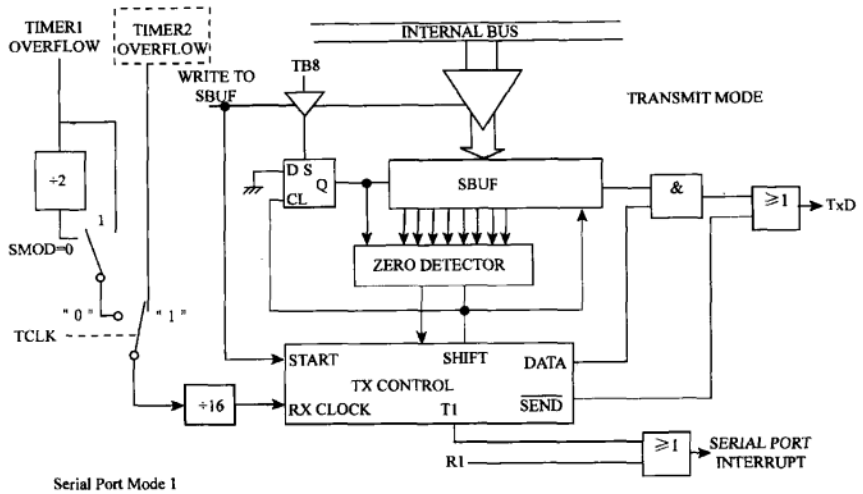


图 11-16　串行模式一的传送框图

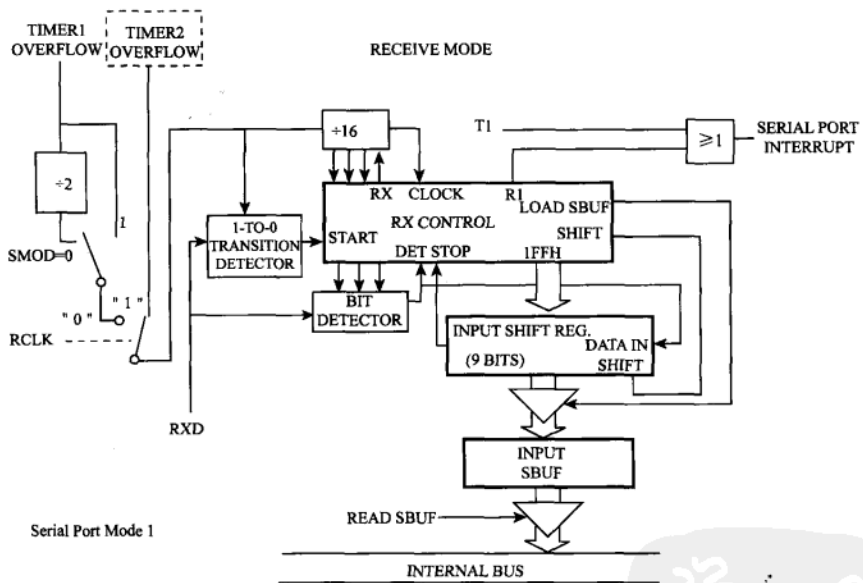


图 11-17　串行模式一的接收框图

11.8　接收串行数据的程序范例

介绍过串行传送的方式，现在来认识一下串行接收的程序写法。

【例 11-2】 11-8-1.ASM

```
$MOD51
        ORG         0000H
START:  MOV         TMOD,#20H           ;TIMER1=MODE2
```

```
        MOV         TH1,#0FDH           ;9600 b/s
        SETB        TR1                 ;START TIMER1
        MOV         SCON,#50H           ;SERIAL MODE1,ENABLE UART
        CLR         RI                  ;RI=0
;
WAIT:   JNB         RI,WAIT             ;WAIT FOR RECEIVE OK
        CLR         RI                  ;RI=0
        MOV         A,SBUF              ;READ DATA
        CPL         A                   ;CPLACC
        MOV         P1,A                ;SHOW DATA ON P1
        SJMP        WAIT                ;WAIT NEXT BYTE
;
        END
```

这个程序是把串行传输所接收到的数据，直接通过 P1 端口表现出来，在 P1 端口接上 LED 显示，就可以清楚地看到串行传输所接收的数据了。不过，这个程序如果连续接收太多字节的数据，LED 在接收中会一直闪烁，闪烁结束后所显示的只是最末字节的数据。

马上来试试串行接收的效果吧!利用本章第一节的 DEMO 程序，当作串行数据来源。烧录好两组程序后，依照下图的连接方式，通电后就可以验证结果了。

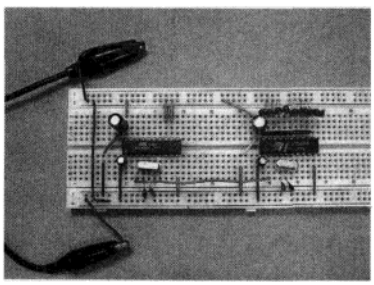


图 11-18　利用两个 MCU 测试串行通信，要把传输端的 P3.1（TxD），连接到接收端的 P3.0（RxD），并利用 LED 显示接收值

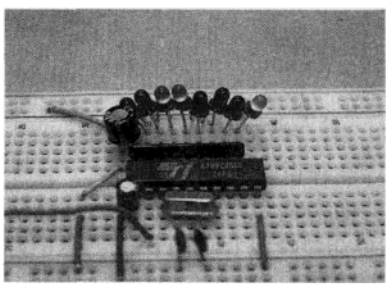


图 11-19　测试时 LED 显示 0011 0001(31H)，代表收到的传送码为 31H

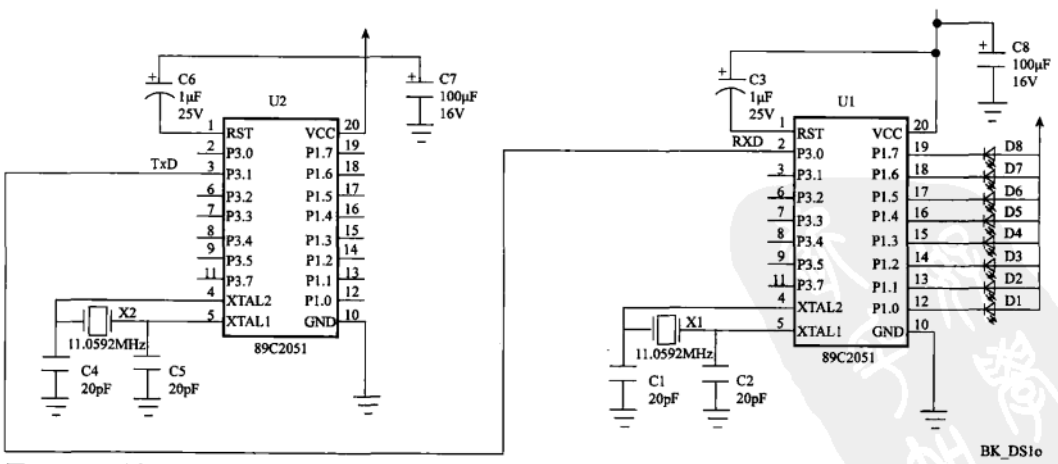


图 11-20　两个 MCU 传送串行数据线路图，左边是传送端，右边是接收端，接收端会把收到的值用 LED 显示出来。一定要确认发送端送出正确的格式后，再来考虑接收端的程序

```
LOOP:   CLR     TI          ;清除 TI 标志位
        MOV     A,#31H      ;选择要送出的码
        CPL     P1.0        ;传送开始的对照点
        MOV     SBUF,A      ;从串行通信专用的寄存器送出
WAIT:   JNB     TI,WAIT     ;等待串行传送完成
LOOP:   JNB     RI,WAIT     ;清除 RI 等待串行数据进入
        CLR     RI          ;此时已经有一笔数据进入
                            ;SBUF 了, RI=0
        MOV     A,SBUF      ;A=SBUF 读到数据
        CPL     A           ;CPL ACC
        MOV     P1,A        ;反相后送到 LED 上
        SJMP    LOOP        ;等待下一笔数据
```

11.9 接收程序模块说明

看完串行接收的范例程序，我们来了解一下工作原理。

```
        CLR   RI              ;CLEAR RI                    (1)
WAIT:   JNB   RI,WAIT         ;WAIT FOR RECEIVE OK         (2)
        CLR   RI              ;CLEAR RI                    (3)
        MOV   A,SBUF          ;READ DATA                   (4)
```

程序注释说明：

（1）RI 标志位是串行通信功能中管理数据接收的控制点，当 MCU 将数据完整接收后，RI 标志位会被设成 1，同 TI 一样，RI 设成 1 之后并不会自动恢复成 0 的状态，必须用通过程序将其清成 0。所以，在启动串行功能之前，请先行将 RI 填成 0，再通过此行程序的判断，来得知现在 MCU 是否收到串行数据。

（2）等待并确定串行数据接收完成。

（3）数据接收完成后，请记得要把 RI 再填为 0，以方便下一个字节数据的接收判断用。

（4）把接收到的数据读取出来，存放在 ACC 累加器中。接收串行数据跟传送串行数据一样，都必须尽量通过 ACC 累加器来执行，用其他的寄存器也可以，但是无法像累加器这么方便。

使用时机：

接收串行数据。

使用限制：

要把 P3.0(RxD)空出来给串行传输使用。

11.10 串行程序的设定流程

经过前面程序范例的说明，以及串行相关寄存器的介绍，我们整理出一个编写程序的流程，只要按照这样的顺序，就可以在 MCU 上轻松应用串行通信的功能。

（1）设定 Timer1 为模式二。

（2）设定波特率。

（3）启动 Timer1。

（4）设定串行模式一并启动串行接收功能。

（5）清除 RI 与 TI 标志位。

（6）如要接收数据，则等待串行接收完成后，清除 RI 标志位，并将 SBUF 的内容读取到 ACC 累加器中。

（7）如要传送数据，先将想要传送的码填入 ACC 累加器中，并通过 SBUF 传送，传送时必须等待传送完成再执行下一步操作。

11.11 ASCII 的介绍

在进行串行通信时，我们常常会使用一种特定格式的编码，用来表示英文字母、数字、标点符号等等，这个编码格式叫做 ASCII。

所谓的 ASCII（American Standard Code for Information Interchange，美国信息交换标准编码），是基于罗马字母表的一套计算机编码系统，主要用于显示英语和其他西欧语言，是现今最完整的 1 字节编码系统，图 11-21 是编码列表（ASCII 表）。

如何使用这个列表呢？举几个例子说明：

（1）查询大写的“E”，其十六进制值应为 45H，十进制值为 69。

（2）由十六进制值 71H 查询 ASCII 值，对应的应该是“q”。

（3）由十进制值 32 查询 ASCII 值，对应的是“SP”，也就是空白(SPACE)的意思。

	0	1	2	3	4	5	6	7
0	NUL 00 0	DEL 10 16	SP 20 32	0 30 48	@ 40 64	P 50 80	` 60 96	p 70 112
1	SOH 01 1	DC1 11 17	! 21 33	1 31 49	A 41 65	Q 51 81	a 61 97	q 71 113
2	STX 02 2	DC2 12 18	" 22 34	2 32 50	B 42 66	R 52 82	b 62 98	r 72 114
3	ETX 03 3	DC3 13 19	# 23 35	3 33 51	C 43 67	S 53 83	c 63 99	s 73 115
4	EOT 04 4	DC4 14 20	$ 24 36	4 34 52	D 44 68	T 54 84	d 64 100	t 74 116
5	ENQ 05 5	NAK 15 21	% 25 37	5 35 53	E 45 69	U 55 85	e 65 101	u 75 117
6	ACK 06 6	SYN 16 22	& 26 38	6 36 54	F 46 70	V 56 86	f 66 102	v 76 118
7	BEL 07 7	ETB 17 23	' 27 39	7 37 55	G 47 71	W 57 87	g 67 103	w 77 119
8	BS 08 8	CAN 18 24	(28 40	8 38 56	H 48 72	X 58 88	h 68 104	x 78 120
9	HT 09 9	EM 19 25	) 29 41	9 39 57	I 49 73	Y 59 89	i 69 105	y 79 121
A	LF 0A 10	SUB 1A 26	* 2A 42	: 3A 58	J 4A 74	Z 5A 90	j 6A 106	z 80 122
B	VT 0B 11	ESC 1B 27	+ 2B 43	; 3B 59	K 4B 75	[5B 91	k 6B 107	{ 7A 123
C	FF 0C 12	FS 1C 28	, 2C 44	< 3C 60	L 4C 76	\ 5C 92	l 6C 108	| 7B 124
D	CR 0D 13	GS 1D 29	- 2D 45	= 3D 61	M 4D 77	] 5D 93	m 6D 109	} 7C 125
E	SO 0E 14	RS 1E 30	. 2E 46	> 3E 62	N 4E 78	^ 5E 94	n 6E 110	~ 7D 126
F	SI 0F 15	US 1F 31	/ 2F 47	? 3F 63	O 4F 79	_ 5F 95	o 6F 111	DEL 7F 127

高位

低位

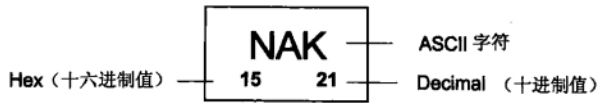


图 11-21　ASCII 表单格说明

	0	1	2	3	4	5	6	7
0	NUL 00 0	DEL 10 16	SP 20 32	0 48	@ 40 64	P 50 80	` 60 96	p 70 112
1	SOH 01 1	DC1 11 17	! 21 33	1 31 49	A 41 65	Q 51 81	a 61 97	q 71 113
2	STX 02 2	DC2 12 18	" 22 34	2 32 50	B 42 66	R 52 82	b 62 98	r 72 114
3	ETX 03 3	DC3 13 19	# 23 35	3 33 51	C 43 67	S 53 83	c 63 99	s 73 115
4	EOT 04 4	DC4 14 20	$ 24 36	4 34 52	D 44 68	T 54 84	d 64 100	t 74 116
5	ENQ 05 5	NAK 15 21	% 25 37	5 35 53	E 45 69	U 55 85	e 65 101	u 75 117
6	ACK 06 6	SYN 16 22	& 26 38	6 36 54	F 46 70	V 56 86	f 66 102	v 76 118
7	BEL 07 7	ETB 17 23	' 27 39	7 37 55	G 47 71	W 57 87	g 67 103	w 77 119
8	BS 08 8	CAN 18 24	(28 40	8 38 56	H 48 72	X 58 88	h 68 104	x 78 120
9	HT 09 9	EM 19 25	) 29 41	9 39 57	I 49 73	Y 59 89	i 69 105	y 79 121
A	LF 0A 10	SUB 1A 26	* 2A 42	: 3A 58	J 4A 74	Z 5A 90	j 6A 106	z 80 122

图 11-22　查询实例

11.12　将数值数据转为 ASCII 码

假如今天我们在 MCU 中得到一个运算值 7，那如何将 7 以 ASCII 编码方式传送呢？在做仪器连线时这种例子是很常见的。

我们查一下 ASCII 编码，“7”所代表的是 37H，所以如果要将数值 7 以 ASCII 方式传送的话，程序必须这么写：

```
MOV A,#37H
MOV SBUF,A
```

所以我们可以建一个表格，用 DPTR 变址寻址方式，把需要的数值找出来：

```
        MOV         A,#7
        MOV         DPTR,#TABLE
        MOVC        A,@A+DPTR
        MOV         SBUF,A
        SJMP        SOMEWHERE
;
TABLE:  DB          30H             ;ASCII 0
        DB          31H             ;ASCII 1
```

```
        DB          32H             ;ASCII 2
        DB          33H             ;ASCII 3
        DB          34H             ;ASCII 4
        DB          35H             ;ASCII 5
        DB          36H             ;ASCII 6
        DB          37H             ;ASCII 7
        DB          38H             ;ASCII 8
        DB          39H             ;ASCII 9
;另一种 TABLE 表的写法
TABLE2: DB          '0', '1', '2', '3', '4'
        DB          '5', '6', '7', '8', '9'
```

不过，如果每一个数值都这样查，程序的写法会变得很复杂，有没有比较好一点的方式呢？答案就是把数值加上“30H”。为何要加 30H？因为 30H 所代表的就是 ASCII 的“0”，请翻到前一页的 ASCII 表说明。而 0～9 的编码又刚好是依序递增的，所以如果以 ASCII 的格式传送 ASCII 码，就可以通过下面的写法来达成：

```
MOV A,#7
ADD A,#30H
MOV SBUF,A
```

而 MOV A,#7 这一行，你可以把 7 改为 0～9 的任意数值。

除此之外，你也可以这么写：

```
MOV A,#7
ADD A,#'0'
MOV SBUF,A
```

这个写法是编译器的特殊功能，ASM51 会自己判断“0”就是 ASCII 的 30H，并将 30H 加到程序中。

11.13 串行传输的考虑

串行通信介绍到此，已经把大多数常用的功能都介绍完了，不过，当你在使用串行功能的时候，我们还有一些小细节要提醒你，这些细节可以协助你在串行功能无法顺利执行时，迅速找出问题所在。

重点 1．波特率与串行模式的设定要正确

波特率与串行模式的设定是串行通信的首要工作，如果传送跟接收的双方设定不同，在数据通信时就会发生错误。无论你是跟其他装置或 PC 连接，这个设定一定要相同，否则双方是完全无法沟通的。

重点 2．石英晶振的振荡频率要兼容

石英晶振的选用是另外一项要素。兼容的石英晶振，才调得出误差极小的波特率，有些时候串行通信的异常，是石英晶振振荡不稳定造成的，在我们实际遇到的例子里，就发现过 11.0592MHz 的石英晶振竟然以 33MHz 振荡，因此，一定要确认石英晶振的频率是兼容的。比如说 11.0592MHz 可以跟 22.1184 兼容，12MHz 可以跟 4MHz 兼容，但是 11.0592MHz 就

一定不要跟 12MHz 兼容，就算串行可以工作，动起来也不太正常。串行通信时波特率稍有误差是可以接受的，主要的原因都是石英晶振造成的。不过，11.0592MHz 的石英晶振并不难买，最好还是用相同的频率规格，以免发生传输错误。

重点 3. 要学会用示波器观看串行波形

示波器对学习串行通信来说，一定有很大的助力而不是阻力，因为示波器可以将很多波形的特性显示出来，不管是波特率、石英晶振的振荡频率、传输时的码不小心送错了等等，只要示波器在手，就可以很快看出问题。

重点 4. 先传送验证然后才是接收数据的验证

如果送出的串行数据不对的话，接收端再怎么写也无法还原传输的串行数据。所以我们一定要先确认所有送出来的串行数据的正确性，而最好的验证工具就是示波器了。

重点 5. 传送与接收数据之间要有一小段反应延迟时间

串行通信会受到传输距离的影响，而加长了传输的时间，也容易因外界电器干扰，让信号的传输能力降低，因此，在规划传送与接收操作时，一定要有适当的等待时间，让对方有时间来反应。

重点 6. 传送与接收是要时间去完成的

以最常见的传输速度 9600b/s 为例，传送 1 字节数据要花将近 1ms 即千分之一秒的时间，这也就是说一秒钟顶多能传送 1000 字节的数据，这个串行传送的速率远比 MCU 的指令执行速度要慢许多，如果没有考虑到这个细节，程序就容易发生莫名的错误操作。当我们发出了 MOV SBUF,A 指令之后，串行数据就立刻送出去了吗？不是，应该是过了千分之一秒的时间后，串行数据才真的送完。而这段时间可能足够让 8051 单片机多执行 1000 行上下的指令。所以，比较好的串行传输的程序应该用中断服务程序来处理。

11.14 中断接收程序

前面所提到的范例，我们都必需通过程序去检查 TI 与 RI 标志位的状态，如果因为执行某段特定的运算，因而拖延了超过二个字节的传输时间后才接收数据，就会造成数据接收的缺漏。为了有效改善这个问题，我们可以通过前一章所提到的串行中断方式来接收，首先改写的是接收程序。例 11-3 是改写后的接收程序：

【例 11-3】 11-14-1.ASM

```
$MOD51
        ORG         0000H
        SJMP        START
        ORG         0023H
        SJMP        SERIAL          ;(1)
;
        ORG         0030H
START:  MOV         TMOD,#21H
        MOV         TH1,#0FDH
```

```
        SETB    TR1
        MOV     SCON,#50H
        SETB    ES              ;(2)
        SETB    EA              ;(3)
STOP:   SJMP    STOP
;
SERIAL: JNB     R1,EXIT         ;(4)
        CLR     RI
        MOV     A,SBUF
        CPL     A
        MOV     P1,A
EXIT:   RETI                    ;(5)
;
        END
```

串行中断的工作，就是通过 TI 和 RI 的状态被设成 1 来决定的。当 TI 或是 RI 被设成 1 时，就会产生串行中断请求，此时系统会进入串行断点去执行服务程序，完成接收及 RETI 后才回到主程序继续刚刚没做完的工作。

程序改写说明：

（1）这一行是断点的声明，当进入串行中断时，要优先执行 SERIAL 的服务程序。

（2）声明好断点，要启动串行中断的功能。

（3）完成串行中断的设定后，就开始中断功能的执行。

（4）串行中断产生时，有可能是 TI 被填成 1，为了确定是 RI 被填成 1 所产生的中断，于是加入这一行判断，如果不是 RI 所产生的中断，就回到主程序。

（5）中断操作完成后，将控制权交还给系统。

同样通过第一节的程序范例来当做数据来源，改成这样的写法后，读取到的值是 31H，不过改写成这样后，却会让系统的运作更有效率，也更有相对的扩展性。

11.15 中断传送程序

改写好接收程序，接着改写传送程序。我们把第一节的传送程序拿来改写，改写后的程序如例 11-4 所示。

【例 11-4】 11-15-1.ASM

```
$MOD51
        OGR     0000H
        SJMP    START
        ORG     0023H
        SJMP    SERIAL          ;(1)
;
        ORG     0030H
START:  MOV     TMOD,#21H
        MOV     TH1,#0FDH
        SETB    TR1
        MOV     SCON,#50H
        SETB    SE              ;(2)
        SETB    EA              ;(3)
```

```
        SETB    TI              ;(4)
STOP:   SJMP    STOP
;
SERIAL: JNB     TI,EXIT         ;(5)
        CLR     TI
        MOV     A,#31H
        MOV     SBUF,A
        ACALL   DELAY           ;(6)
EXIT:   RETI                    ;(7)
;
DELAY:  MOV     R0,#08H
        DJNZ    R0,$
        RET
        END
```

程序改写说明：

（1）声明串行中断进入点。

（2）启动串行中断功能。

（3）开始中断。

（4）通过程序将 TI 填成 1，产生第一次中断。

（5）进入中断时，如果不是传送所产生的中断，则离开中断传送。

（6）送一个字节的数据，中间要间隔一小段时间。

（7）中断操作完成后，将控制权交还给系统。

使用中断传送的功能，一旦将 TI 填为 1，就会开始送出串行数据，如果传送完成，又会自动将 TI 填为 1，下一个串行中断又会继续产生，所以就会不断重复地送出数据。

11.16 串行中断的注意事项

使用串行中断功能来进行传输时，有几个必须要掌握的原则：

（1）先判别是传送 TI 或接收 RI 所产生的串行中断，再分别完成传送或接收操作。

（2）串行中断在进行数据传送时，字节与字节之间还是要有一小段时间的延迟。

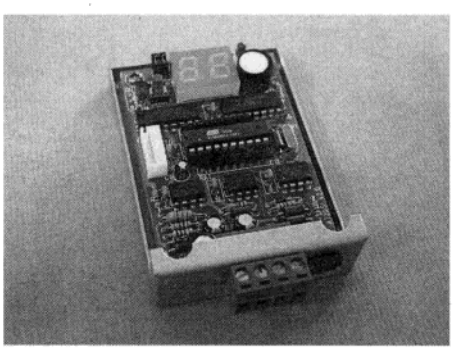


图 11-23 AT2051 控制板是用串行传输 MODE1 与 PC 通信

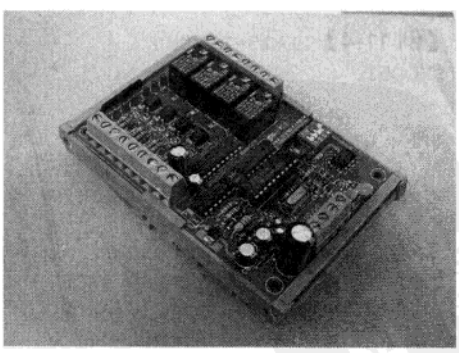


图 11-24 旗威科技公司的 DIO 板也是用串行传输 MODE1 与 PC 通信

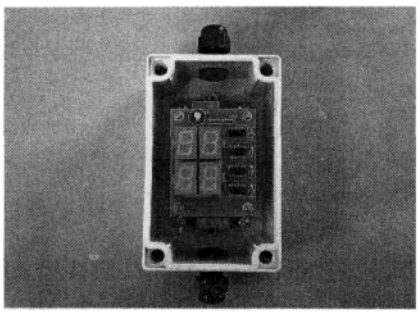


图 11-25　TH2040 温湿度计内部的 AT89C2051 也是串行 MODEl 的应用实例

（3）串行中断期间的操作要越短越好，避免受到其他中断的干扰，导致数据的漏失；如果不能避免干扰，则进入中断时暂时将中断功能关闭，等中断工作完成后再启动中断，或是放弃该次的数据。

（4）无论是一般串行传输，还是使用中断来进行串行传输，都要会善用示波器进行除错的工作，因为只有示波器才能明明白白将传送的数据串显示出来。

11.17　一个完整的串行程序

这是一个完整的串行传输程序，为使系统有更好性能，串行传送都是以中断方式处理。传送的部分是 0～255 不断送出的串行数据，接收则是把收到的值显示在 LED 上。

【例 11-5】11-17-1.ASM

```
$MOD51
            ORG             0000H
            SJMP            START
            ORG             0023H
            SJMP            SERIAL
;
            ORG             0030H
START:      MOV             TMOD,#21H
            MOV             TH1,#0FDH
            SETB            TR1
            MOV             SCON,#50H
            MOV             R1,#0
            SETB            ES
            SETB            EA
            SETB            TI
STOP:       SJMP            STOP
;
SERIAL:     JNB             R1,NEXT
            CLR             R1
            MOV             A,SBUF
            CPL             A
            MOV             P1,A
NEXT:       JNB   TI,EXIT
            CLR             TI
            ACALL           DELAY
            MOV             A,R1
            MOV             SBUF,A
            INC             R1
```

```
EXIT:       RETI
;
DELAY:      MOV         R1,#0
L1:         MOV         R3,#0
            DJNZ        R3,$
            DJNZ        R2,L1
            RET
            END
```

把这个程序烧录到 MCU 中（称为 MCU1），当我们把 TxD 与 RxD 的引脚接起来的时候，从 TxD 送出来的数据会直接进到 RxD 并显示在 LED 上。当我们通过另一个 MCU（称为 MCU2）来验证 MCU1 的操作时，若要验证传送功能，可以在 MCU2 中烧录第八节的串行接收程序，用来验证 MCU1 传送的数据是否正确变化；若要验证接收功能，则在 MCU2 里烧录第一节的串行传送程序，用以验证 MCU1 是否会将接收值正确显示在 LED 上。

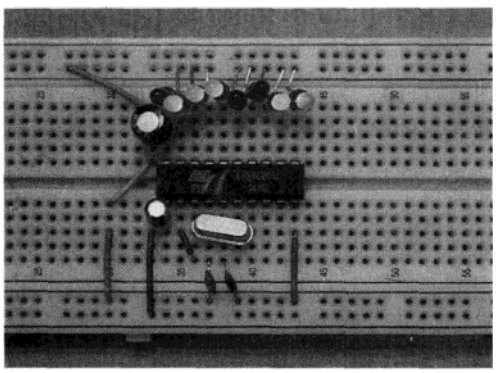


图 11-26　把 MCU1 的 TxD 与 RxD 接在一起的通信状况，LED 显示时很像交通信号灯

您可以从下列公司的网站取得更进一步的信息：

（1）www.chipware.com.tw：查询单片机相关应用的参考资料。

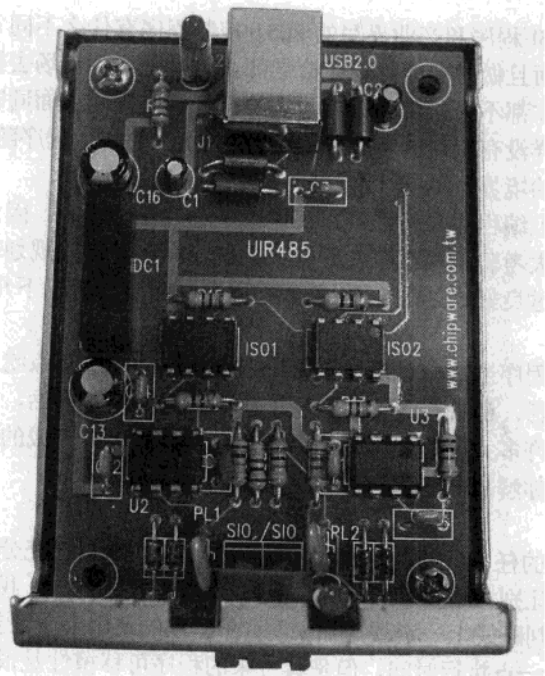
（2）www.flag.com.tw：查询“8051 单片机彻底研究”系列丛书的相关介绍。

（3）www.tektronix.com：查询示波器的相关资料。

（4）www.atmel.com：制造 8051 的厂商，有很多 MCU 的相关参考资料。

（5）www.intel.com：查询 8051（MCS51）的相关介绍。

12 写一个好程序



在本书的所有实验中，每次使用的元件数量最多也不会超过 20 个，但是每章学习的主题却都不一样，其实就是程序内容不一样，你烧录的程序不一样，当然结果也会不同。看了前面十一章后您的感想如何？写一个能工作的 8051 汇编语言程序不难，真的不难。但是，要写一个好的程序，写一个三个月后您我都还看得懂的汇编语言程序，那就需要一点技巧了。本章是本书的最后一章，所有 8051 的精华会在这里再做总整理，希望您能多花一些时间仔细研读并配合必要的 DIY 实验。

第12章　写一个好程序

12.1　好程序的定义

学校专题写的 8051 程序和产业界写的 8051 控制程序有什么不同？前者是为了要毕业的报告，只要能够工作而且做对了就可以交给老师。后者是要到市场去接受考验的，不仅要做对操作而且要怎么操作都不会死机，这方面就要软件和硬件两方面同时配合才行。我们认为写程序就好像画画一样没有绝对的 100 分，只要时间允许的话，程序随时都可以再改得更好，完全没有最完美最好的境界。

我们认为 8051 汇编程序写得好，写其他程序一定得心顺手。因为汇编程序对人而言，绝对是逻辑思维能力大考验。写程序就好像下围棋一样，初步的规划若错了，到最后是很难修正回来的。一个经过良好规划的 8051 单片机程序，应该会有以下几个特点：

（1）好看

自己写的 8051 程序当然会看得懂，如果别人看您的程序有点吃力，如果看了几十行下来，有点昏头转向的话，那就有改善的空间了。如果再不修正的话，可能三五个月以后，连你自己也看不懂了。许多大型的程序都是由好几个人分头进行完成的，如果别人看不懂你的程序，可能也无法帮你解决问题了。

（2）好懂

程序要好懂首先的任务就是加入足够的说明和注释，让自己充分理解其中的逻辑判断走向，当然同时也可以让别人适度理解。所以在写 8051 汇编程序中，我们一定要求要加程序注释，尤其在程序进行判断式上一定要详加说明。在例程的调用上一定要保持一进一出的原则，虽然这种做法会增加一点执行时间，但是其带来的程序可移植性是很难去评估的。在“8051 彻底研究”一系列书中，至少有 30 个以上的例程（Routine）你可以不做任何修改就可引用，其主要的优点除了说明清楚以外，这些例程都保持一进一出的程序结构。

（3）产生的程序代码精简

如果你对 CPU 的结构熟悉，可以对程序的掌握度更高，你的程序将会更精简，使用几个简单指令的组合，去完成重要的程序处理过程。8051 单片机的相关参考资料相当多，唯有越深入了解 8051，你才能更精通掌握。

（4）执行速度快

8051 程序写得好，除了系统稳定以外，执行速度加快也是重点之一。某些特定的环境就是要求反应速度要快，例如条形码辨别、IR 红外线接收、机车的电子式引擎点火控制以及高速电动机的控制等等。上述的场合信号可能一下子就不见了，8051 此时就要很好地处理所有进来的信号，并保留足够的反应时间。你写的程序若执行不够快的话，马上就会发生问题。

（5）保留足够的除错点

程序无论怎么写还是有出错的机会，面对类似的问题我们认为在程序上保留除错点才对。程序不论大小都要有一些检测的机制来测试程序的流程或走向是否如事先预期的那样。

若程序运行错了一定是某个环节出了问题，这时就可以靠除错点的加入来帮忙了，当然配合的除错说明文件也要妥善保留，你写的 8051 程序如果两年后才出问题，这些除错点就是这时的救命仙丹了。

12.2 AT89C2051 重点资料

写 8051 汇编程序需要准备那些资料才够呢？以下是我们重点整理的资料：

（1）8051 寄存器的说明。

（2）可位寻址寄存器的说明。

（3）ASCII 符号表。

（4）8051 指令集。

（5）影响标志位的指令。

准备好以上资料可以让我们遇到 8051 的程序问题时，立即找到说明和解答。在本书的附录上都有类似的资料，所以只要摆上一本《8051 单片机彻底研究——入门篇》就够了，除此之外还要有：

（1）AT89C2051 或 AT89S51/52 的原厂 Data Sheet。

（2）ASM51 原厂提供的使用与操作手册（自行印出并装订）。

（3）一台可以写程序的 PC 个人电脑。

（4）ADSL 或 MODEM 调制解调器随时可以上网找资料，以及寻求问题的解答。

（5）AT89C2051 或 AT89S52 相关实验元件，材料费应该在 250 元以内。

我们不建议您一有问题就想找人问，这点是有违 DIY 的精神。想帮助您的人一定希望您的问题是经过“思考”后的好问题，而不是别人的全力帮忙。看程序很辛苦，看别人的程序更辛苦，您认为呢？

下边为一个标准 8051 的程序标准结构，我们强烈建议您所写的任何程序都符合此标准。

```
;程序名称:EXAMPLE.ASM
;完成日期:2006/03/31
;版本:V1.0.0
;后续修正说明
;客户名称
;除错机制说明
;所有的声明与定义
RESET       EQU         0000H
STACK       EQU         60H                 ;堆栈区 60～7FH 共 32 字节
;
            ORG         RESET
            LJMP        START
;中断服务程序进入点
;
START:      MOV         R0,#00H
            DJNZ        R0,$                ;开机延迟一小段时间
            MOV         SP,#STACK           ;设定堆栈
            LCALL       TIMER_INIT          ;Timer 设定
            LCALL       SIO_INIT            ;串行设定
```

```
            LCALL       ISR_INIT            ;中断设定
            SETB        IE.7                ;开始允许中断
MAIN:                                       ;主程序开始
TABLE:                                      ;数据表在这里
;服务例程区
T0_ISR:                                     ;定时中断服务例程
            RETI
SIO_ISR:                                    ;串行中断服务例程
            RETI
ROUTINES:                                   ;例程放在这里
;
;
            END                             ;程序全部结束
```

12.3 这样写会更好

写对程序不难，写一个好的 8051 程序才是重点。我们一直认为 8051 的汇编程序要好好规划，才会减少出错的机会。至于如何好好规划，最好的方法就是观察别人写的程序，参考别人的 8051 程序的写法。以下的程序均摘自本书前面的程序范例，我们顺便把改写的原因也做了说明，这样的写法会更好。

修改前：

【例 12-1】 9-1-1.ASM

```
$MOD51
            ORG         0000H
START:      MOV         TMOD,#01H       ;SET TIMER0 TO MODE1
            ACALL       T0_RELODD       ;CALL RELOAD FUNCTION
;
LOOP:       JNB         TF0,LOOP        ;WAIT FOR OVERFLOW
            ACALL       T0_RELOAD       ;RELOAD TIMER0 SETTINGS
            CPL         P1.0            ;CPL P1.0
            AJMP        LOOP            ;WAIT FOR OVERFLOW AGAIN
;
;SET TIMER0 FOR 10mS(XTAL=12MHz)
T0_RELOAD:
            CLR         TR0             ;STOP TIMER0
            MOV         TH0,#0D8H       ;(65536-10000)/256
            MOV         TL0,#0F0H       ;(65536-10000)%256
            CLR         TF0             ;CLEAR TIMER0 OVERFLOW
            SETB        TR0             ;START TIMER0
            RET
            END
```

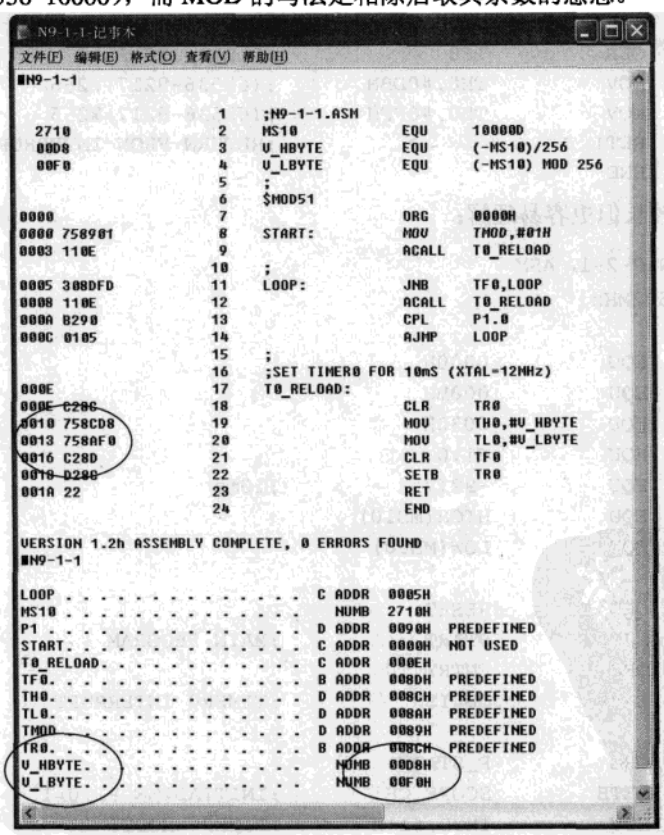
修改后：

【例 12-2】 N9-1-1.ASM

```
MS10    EQU         10000
V_HBYTE EQU         (-MS10)/256
V_LBYTE EQU         (-MS10)MOD 256
;
```

```
$MOD51
        ORG         0000H
START:  MOV         TMOD,#01H       ;SET TIMER0 TO MODE1
        ACALL       T0_RELOAD       ;CALL RELOAD FUNCTION
;
LOOP:   JNB         TF0,LOOP        ;WAIT FOR OVERFLOW
        ACALL       T0_RELOAD       ;RELOAD TIMER0 SETTINGS
        CPL         P1.0            ;CPL P1.0
        AJMP        LOOP            ;WAIT FOR OVERFLOW AGAIN
;
;SET TIMER0 FOR 10mS(XTAL=12MHz)
T0_RELOAD:
        CLR         TR0             ;STOP TIMER0
        MOV         TH0,#V_HBYTE
        MOV         TL0,#V_LBYTE
        CLR         TF0             ;CLEAR TIMER0 OVERFLCOW
        SETB        TR0             ;START TIMER0
        RET
        END
```

修改的部分是把定时器的设定改到声明这边，使程序更具灵活性。MS10 前又加一个负号的写法等于（65536-10000），而 MOD 的写法是相除后取其余数的意思。



```
N9-1-1-记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
■N9-1-1

                        1   ;N9-1-1.ASM
  2710                  2   MS10            EQU     10000D
  00D8                  3   V_HBYTE         EQU     (-MS10)/256
  00F0                  4   V_LBYTE         EQU     (-MS10) MOD 256
                        5   ;
                        6   $MOD51
0000                    7                   ORG     0000H
0000 758901             8   START:          MOV     TMOD,#01H
0003 110E               9                   ACALL   T0_RELOAD
                       10   ;
0005 308DFD            11   LOOP:           JNB     TF0,LOOP
0008 110E              12                   ACALL   T0_RELOAD
000A B290              13                   CPL     P1.0
000C 0105              14                   AJMP    LOOP
                       15   ;
                       16   ;SET TIMER0 FOR 10mS (XTAL=12MHz)
000E                   17   T0_RELOAD:
000E C28C              18                   CLR     TR0
0010 758CD8            19                   MOV     TH0,#V_HBYTE
0013 758AF0            20                   MOV     TL0,#V_LBYTE
0016 C28D              21                   CLR     TF0
0018 D28C              22                   SETB    TR0
001A 22                23                   RET
                       24                   END

VERSION 1.2h ASSEMBLY COMPLETE, 0 ERRORS FOUND
■N9-1-1

LOOP . . . . . C ADDR  0005H
MS10 . . . . . NUMB  2710H
P1 . . . . . D ADDR  0090H  PREDEFINED
START. . . . . C ADDR  0000H  NOT USED
T0_RELOAD. . . . . C ADDR  000EH
TF0. . . . . B ADDR  008DH  PREDEFINED
TH0. . . . . D ADDR  008CH  PREDEFINED
TL0. . . . . D ADDR  008AH  PREDEFINED
TMOD . . . . . D ADDR  0089H  PREDEFINED
TR0. . . . . B ADDR  008CH  PREDEFINED
V_HBYTE. . . . . NUMB  00D8H
V_LBYTE. . . . . NUMB  00F0H
```

图 12-1　TH0 与 TL0 经过 LST 文件检查，确定其设定值与原程序相同

10-2-1 定时中断示范程序修改前的写法：

【例 12-3】10-2-1.ASM

```
$MOD51
            ORG         0000H
            LJMP        START           ;MAIN PROGRAM
            ORG         000BH
            LJMP        T0_ISR          ;TIMER0 INTERRUPT
;
            ORG         0030H
START:      SETB        P1.0            ;INITIALIZE P1.0=1
            MOV         TMOD,#11H       ;TIMER0=MODE1
            MOV         TH0,#0DBH       ;(65536-9217)/256
            MOV         TL0,#0FFH       ;(65536-9217)%256
            CLR         TF0             ;CLEAR TF0 FLAG
            SETB        TR0             ;START TIMER0
            SETB        ET0             ;ENABLE TIMER0 INTERRUPT
            SETB        EA              ;START INTERRUPT
LOOP:       SJMP        LOOP
;
T0_ISR:     CPL         P1.0            ;CPL P1.0
            CLR         TF0             ;CLEAR TF0 FLAG
            MOV         TH0,#0DBH       ;(65536-9217)/256
            MOV         TL0,#0FFH       ;(65536-9217)%256
            RETI                        ;RETURN FRON INTERRUPT
            END
```

修改后程序较长但更容易理解：

【例 12-4】N10-2-1. ASM

```
;XTAL=11.0592MHz
$MOD51
RESET       EQU         0000H
ENTRY_T0    EQU         000BH
P_START     EQU         0030H
SCOPE_CH1   EQU         P1.0
MS10        EQU         -9217           ;10ms
V_H         EQU         HIGH(MS10)
V_L         EQU         LOW(MS10)
;
            ORG         RESET
            LJMP        START           ;MAIN PROGRAM
            ORG         ENTRY_T0
            LJMP        T0_ISR          ;TIMER0 INTERRUPT
;
            ORG         P_START
START:      SETB        SCOPE_CH1       ;INITIALIZE P1.0=1
            MOV         TMOD,#11H       ;TIMER0=MODE1
            MOV         TH0,#V_H        ;(65536-9217)/256
            MOV         TL0,#V_L        ;(65536-9217)%256
```

```
                CLR         TF0             ;CLEAR TF0 FLAG
                SETB        TR0             ;START TIMER0
                SETB        ET0             ;ENABLE TIMER0 INTERRUPT
                ETB         EA              ;START INTERRUPT
LOOP:           SJMP        LOOP
;
T0_ISR:         CPL         SCOPE_CH1       ;CPL P1.0
                CLR         TF0             ;CLEAR TF0 FLAG
                MOV         TH0,#V_H        ;(65536-9217)/256
                MOV         TL0,#V_L        ;(65536-9217)%256
                RETI                        ;RETURN FROM INTERRUPT
                END
```

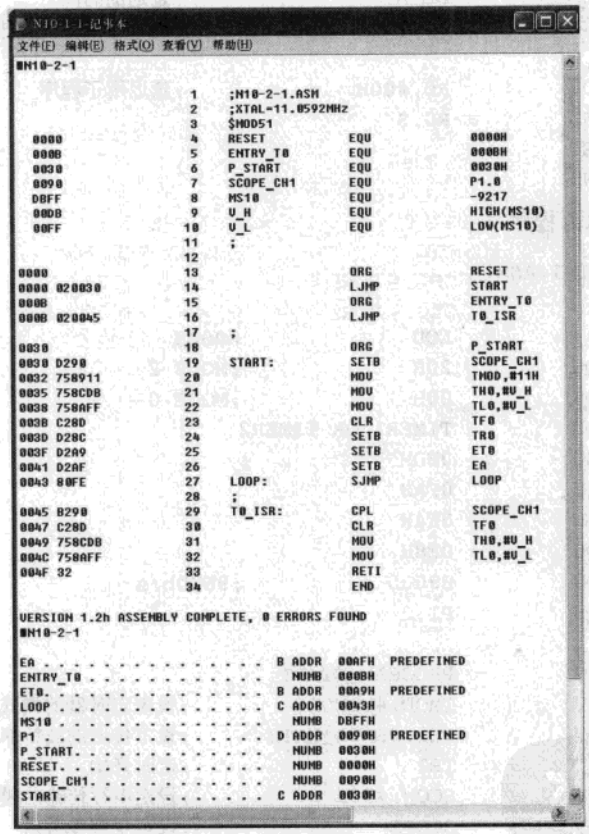


图 12-2　所有的设定值都在程序最前面的声明中用伪指令定义，方便程序修改用 HIGH（MS10）的写法是取该值的高位组（位 15～位 8），LOW（MS10）是取 MS10 的低位组（位 7～位 0）

修改前的串行范例程序：

【例 12-5】 11-1-1.ASM

```
$MOD51
                ORG         0000H
START:          MOV         TMOD,#20H           ;设定定时器的模式
```

```
            MOV         TH1,#0FDH           ;设定传送的波特率 9600b/s
            SETB        TR1                 ;定时开始
            MOV         SCON,#50H           ;设定串行传输的模式
            SETB        P1.0                ;设定观察参考点
;
LOOP:       CLR         TI                  ;清除 TI 标志位
            MOV         A,#31H              ;选择要送出的码
            CPL         P1.0                ;传送起始的参考点
            MOV         SBUF,A              ;从串行专用的寄存器送出
;
WAIT:       JNB         TI,WAIT             ;等待传送完成
            CLP         P1.0                ;传送完成的参考点
            ACALL       DELAY               ;延迟的时间
            SJMP        LOOP                ;继续送同样的码来观察
;
DELAY:      MOV         R0,#00H             ;延迟用子程序
            JNZ         R0,$
            RET
            END
```

修改后的串行示范程序：

【例 12-6】 N11-1-1.ASM

```
$MOD51
PROGRAM_START           EQU             0000H
TIMER1      EQU         20H             ;MODE 2
TIMER2      EQU         00H             ;MODE 0
VTMODE      EQU         TIMER1 OR TIMER2
B9600       EQU         0FDH
B4800       EQU         0FAH
B2400       EQU         0F4H
B1200       EQU         0E8H
BAUD_RATE   EQU         B9600           ;9600b/s
SCOPE       EQU         P1.0
;
            ORG         PROGRAM_START
START:      MOV         TMOD,#VTMODE        ;设定定时器的模式
            MOV         TH1,#BAUD_RATE      ;设定传送的波特率 9600b/s
            SETB        TR1                 ;定时开始
            MOV         SCON,#50H           ;设定串行传输的模式
            SETB        SCOPE               ;设定观察参考点
;
LOOP:       CLR         TI                  ;清除 TI 标志位
            MOV         A,#31H              ;选择要送出的码
            CPL         SCOPE               ;传送起始的参考点
            MOV         SBUF,A              ;从串行专用的寄存器送出
;
WAIT:       JNB         TI,WAIT             ;等待传送完成
            CPL         SCOPE               ;传送完成的参考点
```

```
        ACALL   DELAY           ;延迟的时间
        SJMP    LOOP            ;继续送同样的码来观察
;
DELAY:  MOV     R0,#00H             ;延迟用子程序
        DJNZ    R0,$
        RET
        END
```

12.4 编译不过如何处理

几乎所有的 8051 初学者在刚开始接触汇编程序时，都曾经有一段艰难期，被 ASSEMBLER 整得很惨。只有几行的程序一经过 ASM51 编译竟然一大堆错误，程序写法改了又改还是有错误，经过了大半天还是不行，真的想把电脑砸烂放弃算了。

你一定要先学会写小的 8051 程序才行，先熟悉整个编译的过程，还有 8051 编译程序 ASM51 的约束和限制。而且你一定要自己 KEYIN 程序代码才算，千万不要从光盘上复制一段程序后，就想直接编译成机器码，这时会出现何种错误连 8051 的专家都无法预测。照着书上的写法，写二三十行的程序时，程序不算大，指令打错了或是操作数的写法有问题，ASM51 会指出那一行有错误。这时你一定要花一点时间去找出问题来，真的不行时，对比一下你的程序与书上的范例有哪里不同？甚至一个字一个字地对，只要花点时间绝对可以找出问题点，大不了照抄但是要抄对。

大部分初学者碰到的问题是对指令的不了解，因而用错了指令，即真正的 8051 并没有这种写法，所以刚刚开始时，观察别人的程序范例是很重要的一环，除了研究别人的写法以外，也尽量要理解为什么要这样做?如果不跟着做会有什么后果？最好的学习过程是先跟着一定对的写法做一次，然后再用自己的方法做一次，后者可能会花你八成以上的学习时间，但是如果不这样做，你可能永远无法学到 8051 汇编语言的真正精髓。

以下我们列出几种写 8051 汇编语言程序最常见的错误。

（1）指令写法错误。用了不常用的指令或不熟悉的指令就会犯此错误，解决的方法就是再查一下 8051 的指令。

```
错的  MOV     A，@A+DPTR        ；少了一个 C
对的  MOVC    A，@A+DPTR
错的  CJNE    B，#22H，ADDR     ；8051 没有这个比较指令
对的  CJNE    A，#22H，ADDR     ；最好移到 ACC 来处理
```

（2）数值设定方式错误。将值传给累加器或其他寄存器，十进制与十六进制的表示方法是不同的，这种错误最常见。

```
错的  MOV     R0,#350           ；R0 值超过 255
错的  MOV     A,#1A             ；ASM51 无法确认此值，改成 1AH
错的  MOV     A,#A0H            ；ASM51 误以为 A0H 为标记
对的  MOV     A,#0A0H           ；ASM51 会当成数值 A0H 来处理
不妥  MOV     DPTR,#1235H       ；如果程序有增减，此值可能改变
好的  MOV     DPTR,#TABLE       ；查表用的数据前面加标记
```

（3）跳转错误。在 ASM51 编译时，会自动指出跳转范围错误，短程跳转应该用在例程

或简单的判断上，其他场合应该用 LJMP 或 AJMP。

```
错的   SJMP     SOMEWHERE          ; 短程跳转无法到达
对的   LJMP     SOMEWHERE          ; 改用长程跳转
错的   JB       BIT,ADDR           ; 跳转的地址超过 128 字节
对的   JNB      BIT,ADDR2
       LJMP     ADDR               ; 更改判断式及长程跳转
```

12.5 千错万错都是你的错

ASM51 编译成功完全没有错误，恭喜你。但这只说明程序写法无误，并不表示程序完全是对的。其中最大的问题出在程序的判断式上，一个错误的判断或跳转往往会导致程序完全不工作。

请看下面这个有 BUG 的程序范例：

【例 12-7】PROGRAM NAME:12-5-1.ASM

```
BUFFER      DATA          20H
$MOD51
            ORG           0000H
            MOV           SP,#50H          ;设定堆栈 STACK
            MOV           A,#00H           ;累加器=00H
            MOV           BUFFER,A         ;(20H)=00H
LOOP:       MOV           P1,BUFFER        ;P1=(BUFFER)
            INC           BUFFER           ;将 BUFFER 内容加上 1
            LCALL         DELAY            ;延迟一小段时间
            SJMP          LOOP
;ROUTINE
;单纯时间延迟用
DELAY:      MOV           R0,#00H
DLY :       MOV           R1,#00H
            DJNZ          R1,$
            DJNZ          R0,DELAY
            RET
            END
```

这个程序的用意非常简单，把位于 DATA MEMORY 上 BUFFER 的内容先清为 0，然后将 BUFFER 的值送到 P1 端口上，送出数值后把 BUFFER 的内容加 1 并等待一小段时间，再一直重复上述操作。

如果你仔细的话，会发现第二个 DJNZ 跳错位置了，造成程序一直在 DELAY 例程内做无用功，完全无法离开这个例程。在实际的应用过程中，被调用的例程可能会达数十行之多，我们若要验证程序是否有回到主程序，可以把程序改成下面的样子：

【例 12-8】PROGRAM NAME:12-5-2.ASM

```
$MOD51
BUFFER      DATA          20H
SCOPE_CH1   EQU           P3.7             ;接到示波器 CH1 上观察
```

```
;
            ORG         0000H
            MOV         SP,#50H          ;设定堆栈
            MOV         A,#00H           ;累加器=00H
            MOV         BUFFER,A         ;(20H)=00H
LOOP:       MOV         P1,BUFFER        ;P1=(BUFFER)
            INC         BUFFER           ;将 BUFFER 内容加上 1
            LCALL       DELAY            ;延迟一小段时间
            SJMP        LOOP
;ROUTINE
;单纯时间延迟用
DELAY:      SETB        SCOPE_CH1
            MOV         R0,#00H
DLY:        MOV         R1,#00H
            DJNZ        R1,$
            DJNZ        R0,DELAY
            CLR         SCOPE_CH1
            RET
            END
```

这时只要通过示波器 CH1 观察，就可能发现 P3.7 被设成 1 后就再也没回到 0 的状态，所以可以推论 DELAY 例程一定有问题。用硬件检查的程序技巧也可能改变，安排在主程序 LOOP 上。

【例 12-9】 PROGRAM NAME:12-5-3.ASM

```
$MOD51
BUFFER      DATA        20H
SCOPE_CH1   EQU         P3.7             ;接到示波器 CH1 上观察
;
            ORG         0000H
            MOV         SP,#50H          ;设定堆栈
            MOV         A,#00H           ;累加器=00H
            MOV         BUFFER,A         ;(20H)=00H
LOOP:       MOV         P1,BUFFER        ;P1=(BUFFER)
            INC         BUFFER           ;将 BUFFER 内容加上 1
            SETB        SCOPE_CH1        ;CHECK POINT=1
            LCALL       DELAY            ;延迟一小段时间
            CLR         SCOPE_CH1        ;CHECK POINT=0
            SJMP        LOOP
;ROUTINE
;单纯时间延迟用
DELAY:      MOV         R0,#00H
DLY:        MOV         R1,#00H
            DJNZ        R1,$
            DJNZ        R0,DELAY
            RET
            END
```

上面的程序示范是把硬件的检查点放在主程序上，借用一台数字式示波器就可以看出程

序调用 DELAY 完后就没动静了，当然问题就被锁定在 DELAY 例程上。在找出问题后，只要找到程序中有关 SCOPE_CH1 的行号最前面加个“;”注释记号就行了。接下来我们示范另一种 ASM51 条件式除错的写法。

【例 12-10】PROGRAM NAME:12-5-4.ASM

```
$MOD51
BUFFER      DATA        20H
DEBUG1      EQU         1               ;除错完毕后再把 1 改为 0
SCOPE_CH1 EQU           P3.7            ;接到示波器 CH1 上观察
;
            ORG         0000H
            MOV         SP,#50H         ;设定堆栈
            MOV         A,#00H          ;累加器=00H
            MOV         BUFFER,A        ; (20H)=00H
LOOP:       MOV         P1,BUFFER       ;P1=(BUFFER)
            INC         BUFFER          ;将 BUFFER 内容加上 1
IF(DEBUG1)
            SETB        SCOPE_CH1       ;CHECK POINT=1
ENDIF
            LCALL       DELAY           ;延迟一小段时间
IF(DEBUG1)
            CLR         SCOPE_CH1       ;CHECK POINT=0
ENDIF
            SJMP        LOOP
;ROUTINE
;单纯时间延迟用
DELAY:      MOV         R0,#00H
DLY:        MOV         R1,#00H
            DJNZ        RI,$
            DJNZ        R0,DELAY
            RET
            END
```

程序最前面声明一个 DEBUG1 参数，若 DEBUG 不是 0 的话，就会把除错的信息送到示波器上，等到确定 BUG 被去掉后，再把 DEBUG1 的值改成 0 。有关 DEBUG1 的设定只跟 8051 原始程序有关，当编译成机器码时，完全看不到 DEBUG1 的有关信息。

【例 12-11】PROGRAM NAME:12-5-5.ASM

```
$MOD51
BUFFER      ATDA        20H
            DEBUG EQU   P3.0            ;硬件强制除错位
SCOPE_CH1   EQU         P3.7            ;接到示波器 CH1 上观察
;
            ORG         0000H
            MOV         SP,#50H         ;设定堆栈
            MOV         A,#00H          ;累加器=00H
            MOV         BUFFER,A        ;(20H)=00H
LOOP:       MOV         P1,BUFFER       ;P1=(BUFFER)
            INC         BUFFER          ;将 BUFFER 内容加上 1
;
```

```
                JNB         DEBUG,TEST
                SETB        SCOPE_CH1           ;CHECK POINT=1
TEST:           LCALL       DELAY               ;延迟一小段时间
                JNB         DEBUG,NEXT
                CLR         SCOPE_CH1           ;CHECK POINT=0
NEXT:           SJMP        LOOP
;ROUTINE
;单纯时间延迟用
DELAY:          MOV         R0,#00H
DLY:            MOV         R1,#00H
                DJNZ        R1,$
                DJNZ        R0,DELAY
                RET
                END
```

硬件强制除错的方法经常用在大型设备或仪器上，当产品发生问题时，维修工程师把DEBUG 的按钮按下，接上示波器的测棒，重新开启电源后，机器就进入除错模式，并送出除错信号到指定的位置上，当然也可以找出软硬件的任何错误了。

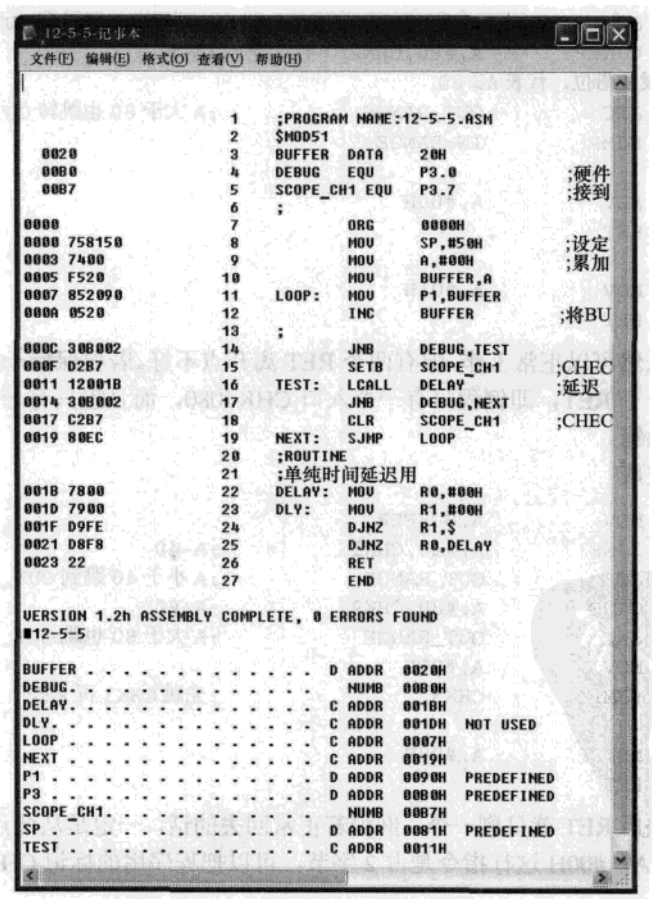


图 12-3　硬件强制除错的方法比起条件式的写法在程序上会多占几行代码，但其方便性是无法估算的

12.6 小心跳转指令

在 8051 汇编语言程序中，最难处理的指令应该就是条件式的跳转了，其操作不难理解，但是难就难在同时有多个条件式跳转出现时，你的逻辑判断能力就要非常清楚，否则一跳错就永远错了。

为了要减少跳转的失误，我们建议程序的写法一定要保持 ONE IN ONE OUT 的方式，也就是尽量保持程序一进一出的编写方式，宁可多占些程序空间，也不要程序精简到自己都看不懂。先看以下的例子：

比较 BUFFER 内的值会由 0～99 变化，试编写一个小例程判断 BUFFER 内的值是否介于 40～80 之间，是的话 ACC=1，反之 ACC=0。

【例 12-12】

```
CHK4080:    MOV         A,BUFFER
            CJNE        A,#40,CHK1          ;A-40
                                            ;A-40 发生借位代表 A<40
CHK1:       JC          OUT_RANGE           ;A 小于 40 跳转到 OUT_RANGE
            CJNE        A,#80,CHK2          ;A-80
;A-80 若没有发生借位，代表 A≥80
CHK2:       JNC         OUT_RANGE           ;A 大于 80 也跳转 OUT_RANGE
            SJMP        IN-RANGE
OUT-RANEG:
            MOV         A,#00H
            RET
IN_RANGE:
            MOV         A,#01H
            RET
```

上面的例程虽然可以正常工作，但有两个 RET 离开点不好。若要保持一进一出的原则时，一定要维持只有一个 RET，即例程只有一个入口 CHK4080，而且也只有一个 RET 出口，这样才方便程序的除错。

所以程序修改成

```
CHK4080:    MOV         A,BUFFER
            CJNE        A,#40,CHK1          ;A-40
CHK1:       JC          OUT_RANGE           ;A 小于 40 跳到 OUT_RANGE
            CJNE        A,#80,CHK2          ;A-80
CHK2:       JNC         OUT_RANGE           ;A 大于 80 也跳 OUT_RANGE
            MOV         A,#01H
            SJMP        CHK3                ;先跳 CHK3 再 RET
OUT_RANGE:
            MOV         A,#00H
CHK3:       RET
```

改成新的写法后 RET 就只剩一个，例程若正常回去的话，一定是从此点离开例程的。如果你查询过 MOV A，#00H 这行指令是占 2 字节，可以把程序用的标记 CHK3 再省下来。

```
CHK4080:    MOV         A,BUFFER
            CJNE        A,#40,CHK1          ;A-40
```

```
CHK1:       JC          OUT_RANGE           ;A 小于 40 跳到 OUT_RANGE
            CJNE        A,#80,CHK2          ;A-80
CHK2:       JNC         OUT_RANGE           ;A 大于 80 也跳 OUT_RANGE
            MOV         A,#01H
            SJMP        OUT_RANGE+2
OUT_RANGE:
            MOV         A,#00H
            RET
```

12.7 入门篇例程整理

例程 1 时间延迟

名称 DELAY,DELAYA

功能：时间延迟或操作延迟约 0.2s。

```
DELAY:  MOV     R6,#00H
        MOV     R7,#00H
        DJNZ    R7,$
        DJNZ    R6,DELAY+2
        RET

另一种写法
BUFDLY1 DATA    1EH                 ;另外声明 DELAY 用变量
BUFDLY2 DATA    1FH
;
DELAYA: MOV     BUFDLY1,#00H        ;占 3 字节
        MOV     BUFDLY2,#00H
        DJNZ    BUFDLY2,$
        DJNZ    BUFDLY1,DELAYA+3
        RET
```

调用方式：

```
        LCALL   DELAY               ;DELAY 0.2S
        ;
        MOV     R6,#80H             ;R6 设成其他值
        LCALL   DELAY+2             ;DELAY 0.1S
```

特别注意事项：

这个例程完全不修改 I/O 的输入/输出状态，注意中断服务程序上不能再运用 R6 和 R7，否则会干扰这两个值。

例程 2 清除存储区

名称 CLR_MEM,FILL_MEM

功能：清除 DATA MEMORY 为 00H 或填入其他值。

```
CLR_MEM:    MOV         A,#00H
FILL_MEM:   MOV         R0,#7FH          ;COUNT=127 字节
CLRM:       MOV         @R0,A
            DJNZ        R7,CLRM
            MOV         @R0,A            ;(00H)=A
            RET
```

调用方式：

```
        LCALL       CLR_MEM          ;CLEAR MEMORY
        ;
        MOV         A,#22H           ;全部清成 22H
        LCALL       FILL_MEM         ;FILL MEMORY
```

特别注意事项：

AT89C2051 内的 DATA MEMORY 有 128 字节，RET 指令前的 MOV@R0，A 是把 00H 地址内填入累加器的值。

例程 3　检查 8 位是否大于、相等或小于

名称　COMPARE8

功能：比较 ACC 上 8 位的数据与 BUF 上的数据结果可为等于、大于或小于。

```
COMPARE8:
        CLR         BIT_EQU
        CLR         BIT_LARGE
        CLR         BIT_SMALL
        CJNE        A,BUF,NEQU       ;A-(BUF)
EQUAL:  SETB        BIT_EQU          ;A=(BUF)
        SJMP        CMP_END
NEQU:   JC          SMALL            ;CY=1 代表 A<(BUF)
        SETB        BIT_LARGE
        SJMP        CMP_END
SMALL:  SETB        BIT_SMALL
CMP_END:RET
```

调用方式：

```
        LCALL       COMPARE8         ;ACC-(BUF)
        JB          BIT_EQU,GREATER
        JB          BIT_LARGE,GREATER
       ;大于或等于时都到 GREATER
        LCALL       COMPARE8
        JB          BIT_EQU,SMALLER
        JB          BIT_SMALL, SMALLER
        ;小于或等于时都到 SMALLER
```

特别注意事项：

请事先保留存放 BUF（8 位）、BIT_EQU、BIT_LARGE 和 BIT_SMALL 的位置。

例程 4　检查 16 位是否大于、相等或小于

名称　COMPARE16

功能：比较 BUF 上 16 位的数据与 5000（1388H），结果可为等于、大于或小于。

```
 V5000   EQU         5000
COMPARE 16:
        CLR         BIT_EQU
        CLR         BIT_LARGE
        CLR         BIT_SMALL
        MOV         A,BUF1
        CJNE        A,#HIGH(V5000),NEQU     ;A-13H
        MOV         A,BUF2
        CJNE        A,#LOW(V5000),NEQU      ;A-88H
EQUAL:  SETB        BIT_EQU                 ;A=(BUF)
        SJMP        C16_END
NEQU:   JNC         LARGE
SMALL:  SETB        BIT_SMALL
        SJMP        C16_END
LARGE:  SETB        BIT_LARGE
C16_END:RET
```

特别注意事项：

请事先保留存放 BUF1、BUF2、BIT_EQU、BIT_LARGE 和 BIT_SMALL 的位置。

例程 5　二进制 0～F 转成 ASCII 码

名称　BIN2HEX

功能：将 ACC 上位 3～位 0 的 0000～1111B 数据转成 0～F 的 ASCII 码，结果放在 ACC 上。

```
BIN2HEX:    ANL         A,#0FH              ;只取 bit3-bit0
            CJNE        A,#10,BIN2          ;A-10
BIN2:       JC          NUM09               ;A<10
            SUBB        A,#10               ;A=A-10
            ADD         A,#'A'
            SJMP        BIN2_END
NUM09:      ADD         A,#'0'              ;A=A+30H
BIN2_END:
            RET
```

调用方式：

```
        MOV             BUF,#01101011B      ;(BUF)=6BH
        MOV             R0,#20H
        MOV             A,BUF               ;A=(BUF)
        SWAP            A                   ;位 7～位 4 与位 3～位 0 对调
        LCALL           BIN2HEX
        MOV             @R0,A               ;位 7～位 4 转存到(R0)
        INC             R0
        MOV             A,BUF
        LCALL           BIN2HEX
```

```
        MOV             @R0,A                   ;位 3～位 0 再转存到(R0)
        ;执行后(20H)='6'=36H,(21H)='B'=42H
```

特别注意事项：

经过此例程转换后，这些 HEX 值通常会经过串行通信端口传出，送给其他的装置显示或控制用。

例程 6　数据移动

名称　BLOCK_MOVE

功能：将程序区的数据共 4 字节移到 R0 指定的地址上。

```
BLOCK_MOVE:
        MOV         R7,#4               ;移动长度,内定为 4 字节
BLK_MV: MOV         A,#00H
        MOVC        A,@A+DPTR
        MOV         @R0,A
        INC         R0
        INC         DPTR
        DJNZ        R7,BLK_MV
        RET
```

调用方式：

```
        MOV         DPTR,#0234H
        MOV         R0,#30H
        LCALL       BLOCK_MOVE          ;数据共 4 字节移到(33H)～(30H)
        ;
        MOV         DPTR,#0100H         ;数据开始地址
        MOV         R0,#30H             ;存放区开始位置
        MOV         R7,#8
        LCALL       BLOCK_MOVE+2        ;移动共 8 字节
        ;
```

特别注意事项：

一定要事先定义 DPTR、R0 和 R7 的值，再调用 BLOCK_MOVE，由于 R7 只有 8 位宽，所以移动的长度只能到 256 字节。当 R7=10 时是移动 10 字节，R7=FFH 时移 255 字节，而 R7=00H 时是移动 256 字节，而不是 0 字节。

例程 7　数据的比较

名称　BLOCK_COMP

功能：比较数据区与程序区的数据共 4 字节。

```
BLOCK_COMP:
        CLR         BIT_ERR
        MOV         R7,#4               ;比较长度
CMP_NXT:MOV         A,@R0               ;DATA MEMORY
        MOV         B,A
```

```
        MOV         A,#00H
        MOVC        A,@A+DPTR           ;PROGRAM MEMORY
        CJNE        A,B,COMP_ERR        ;FAIL
        INC         R0
        INC         DPTR
        DJNZ        R7,CMP_NXT
        SJMP        COMP_END
COMP_ERR:
        SETB        BIT_ERR
COMP_END:
        RET
```

调用方式：

```
        MOV         DPTR,#STRING
        MOV         R0,#30H
        LCALL       BLOCK_COMP          ;比较数据共 4 字节
        JNB         BIT_ERR,DO_START
;
STRING  DB          'STRT'
STRING2 DB          'STOP'
```

特别注意项：

串行通信时，我们会用到本例程去确定收到的字符串是否正确，完全符合时，才执行特定的操作。

例程 8　16 位数据的加法

名称　ADD16

功能：两组 16 位数据的加法。

```
ADD16:
        MOV         A,@R0
        CLR         C
        ADD         A,@R1
        MOV         @R0,A
        INC         R0
        INC         R1
        MOV         A,@R0
        ADDC        A,@R1               ;ADD WITH CARRY
        MOV         @R0,A
        RET
```

调用方式：

```
        MOV         R0,#VALUE1          ;VALUE1(16 位)
        MOV         R1,#VALUE2          ;VALUE2(16 位)
        LCALL       ADD16               ;VALUE=VALUE1+VALUE2
```

特别注意事项：

这是最常用的 16 位加法例程的写法，若加完后有进位时，其进位标志位会改变为 1。

例程 9　两组 16 位数据的减法

名称　SUB 16

功能：两组 16 位数据的减法。

```
SUB16:  MOV     A,@R0
        CLR     C
        SUBB    A,@R1           ;SUBSTRACT WITH CY
        MOV     @R0,A
        INC     R0
        INC     R1
        MOV     A,@R0
        SUBB    A,@R1           ;SUBBTRACT WITH CARRY
        MOV     @R0,A
        RET
```

调用方式：

```
        MOV     R0,#VALUEL      ;VALUE1 (16 位)
        MOV     R1,#VALUE2      ;VALUE2(16 位)
        LCALL   SUB16           ;VALUE=VALUE1-VALUE2
```

特别注意事项：

这是也最常用的 16 位减法例程的写法，若减完后有借位时，其 CY 标志位会改变为 1。

例程 10　16 位数据的乘法

名称　MUL16

功能：两组 16 位数据的乘法。

```
;(23)(22) (21)(20)= (09)(08)×(0B)(0A)
;(27)(26) (25)(24)MUL 后暂时存放区
;
MUL16:  MOV     20H,#00H        ;清除最后结果区，共 4 字节
        MOV     21H,#00H
        MOV     22H,#00H
        MOV     23H,#00H
;MUL1
        LCALL   CLEAR_BUF       ;清除暂时数据存放区
        MOV     A,08H
        MOV     B,0AH
        MUL     AB              ;16 位 RESULT IN B,A
        MOV     24H,A
        MOV     25H,B
        LCALL   ADD_4BYTE
;MUL2
        LCALL   CLEAR_BUF
        MOV     A,09H
        MOV     B,0AH
```

```
        MUL     AB              ;RESULT IN BA
        MOV     25H,A
        MOV     26H,B
        LCALL   ADD_4BYTE
;MUL3
        LCALL   CLEAR_BUF
        MOV     A,08H
        MOV     B,0BH
        MUL     AB              ;RESULT IN BA
        MOV     25H,A
        MOV     26H,B
        LCALL   ADD_4BYTE
;MUL4
        LCALL   CLEAR_BUF
        MOV     A,09H
        MOV     B,0BH
        MUL     AB              ;RESULT IN BA
        MOV     26H,A
        MOV     27H,B
        LCALL   ADD_4BYTE       ;结果在(23)(22)(21)(20)共32位
        RET
;
CLEAR_BUF:
        MOV     24H,#00H
        MOV     25H,#00H
        MOV     26H,#00H
        MOV     27H,#00H
        RET
;
ADD_4BYTE:
        MOV     R0,#20H
        MOV     R1,#24H
        MOV     R7,#4
        CLR     C
ADD_4N: MOV     A,@R0
        ADDC    A,@R1
        MOV     @R0,A
        INC     R0
        INC     R1
        DJNZ    R7,ADD_4N
        RET
```

特别注意事项：

这是用 8051 的 MUL 指令做 16 位的乘法写法，最后的结果在 23H～20H 内，23H 为最高位。16 位的乘法在某些场合是派用得上的，可是用 8051 处理起来就有点吃力了，由此看来 8051 比较在行的还是控制，执行复杂的运算会花上较多的执行时间，而且程序已经不算单纯了。

您可以从下列公司的网站获得更进一步的信息：

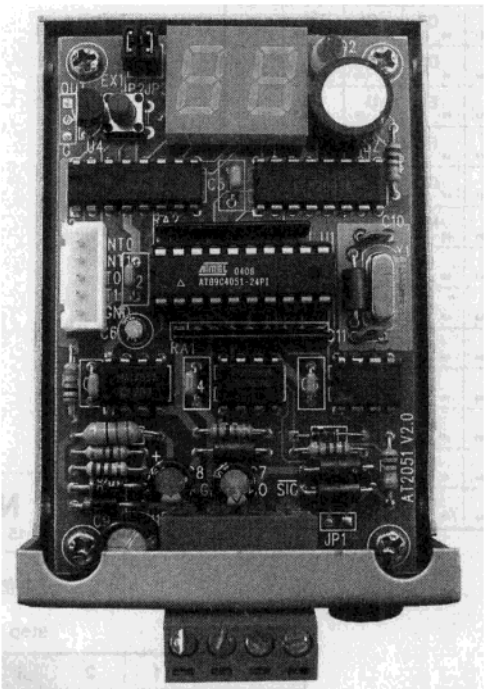
（1）www.chipware.com.tw：查询单片机相关应用的参考资料。

（2）www.flag.com.tw：查询“8051 单片机彻底研究”系列丛书的相关介绍。

（3）www.atmel.com：制造 8051 的厂商，有很多 MCU 的相关参考资料。

（4）www.intel.com：查询 8051（MCS51）的相关介绍。

A 总附录



本书的附录多达 40 多页，这些都在 8051 单片机学习之余，你经常要参考和查阅的主题，这些资料经过我们仔细地筛选和编排，内容绝对是宝贵的，我们希望本附录能成为你编写程序时最佳的参考资料之一。

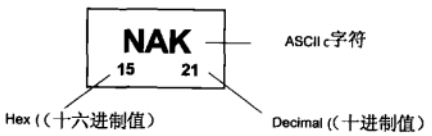
附录 A　ASCII 表

	0	1	2	3	4	5	6	7
0	NUL 00 0	DEL 10 16	SP 20 32	0 30 48	@ 40 64	P 50 80	` 60 96	p 70 112
1	SOH 01 1	DC1 11 17	! 21 33	1 31 49	A 41 65	Q 51 81	a 61 97	q 71 113
2	STX 02 2	DC2 12 18	" 22 34	2 32 50	B 42 66	R 52 82	b 62 98	r 72 114
3	ETX 03 3	DC3 13 19	# 23 35	3 33 51	C 43 67	S 53 83	c 63 99	s 73 115
4	EOT 04 4	DC4 14 20	$ 24 36	4 34 52	D 44 68	T 54 84	d 64 100	t 74 116
5	ENQ 05 5	NAK 15 21	% 25 37	5 35 53	E 45 69	U 55 85	e 65 101	u 75 117
6	ACK 06 6	SYN 16 22	& 26 38	6 36 54	F 46 70	V 56 86	f 66 102	v 76 118
7	BEL 07 7	ETB 17 23	' 27 39	7 37 55	G 47 71	W 57 87	g 67 103	w 77 119
8	BS 08 8	CAN 18 24	(28 40	8 38 56	H 48 72	X 58 88	h 68 104	x 78 120
9	HT 09 9	EM 19 25	) 29 41	9 39 57	I 49 73	Y 59 89	i 69 105	y 79 121
A	LF 0A 10	SUB 1A 26	* 2A 42	: 3A 58	J 4A 74	Z 5A 90	j 6A 106	z 80 122
B	VT 0B 11	ESC 1B 27	+ 2B 43	; 3B 59	K 4B 75	[5B 91	k 6B 107	{ 7A 123
C	FF 0C 12	FS 1C 28	, 2C 44	< 3C 60	L 4C 76	\ 5C 92	l 6C 108	| 7B 124
D	CR 0D 13	GS 1D 29	- 2D 45	= 3D 61	M 4D 77	] 5D 93	m 6D 109	} 7C 125
E	SO 0E 14	RS 1E 30	. 2E 46	> 3E 62	N 4E 78	^ 5E 94	n 6E 110	~ 7D 126
F	SI 0F 15	US 1F 31	/ 2F 47	? 3F 63	O 4F 79	_ 5F 95	o 6F 111	DEL 7F 127

高位元

低位元

Example: NAK (HEX 15, DECIMAL 21)



如何利用本表格

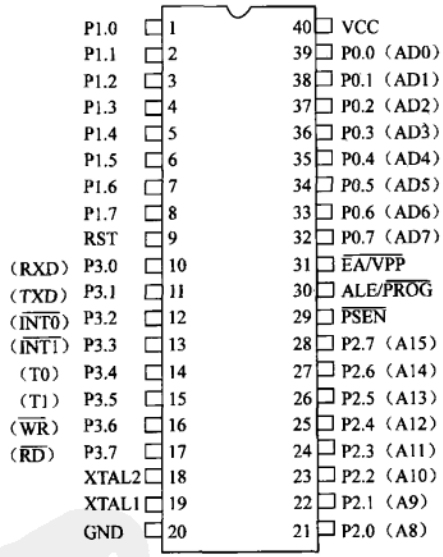
如果我们收到了“30H”这个十六进制数时，代表什么意义呢？请先查竖行的 3，再查横行的 0，两条线交插刚好是“0”（数字 0），这表示我们收到的是一个数字。如下表所示。许多可与电脑连接的仪器设备由于只传递数据，所以送回的值都介于 30H 和 39H 之间，查 ASCII 表的结果，刚好就是数字 0～9。另外如果我们送一个 45H 给 PC 时，PC 的屏幕上就会显示出“E”的字样。

step 1

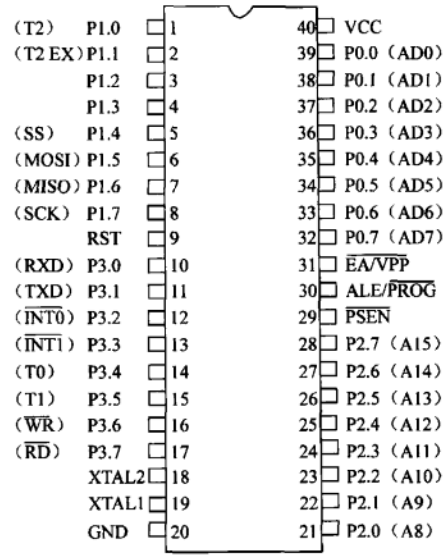
step 2

	0	1	2	3	4	5	6	7
0	NUL 00 0	DEL 10 16	SP 20 32	0 30 48	@ 40 64	P 50 80	` 60 96	p 70 112
1	SOH 01 1	DC1 11 17	! 21 33	1 31 49	A 41 65	Q 51 81	a 61 97	q 71 113
2	STX 02 2	DC2 12 18	" 22 34	2 32 50	B 42 66	R 52 82	b 62 98	r 72 114
3	ETX 03 3	DC3 13 19	# 23 35	3 33 51	C 43 67	S 53 83	c 63 99	s 73 115
4	EOT 04 4	DC4 14 20	$ 24 36	4 34 52	D 44 68	T 54 84	d 64 100	t 74 116
5	ENQ 05 5	NAK 15 21	% 25 37	5 35 53	E 45 69	U 55 85	e 65 101	u 75 117
6	ACK 06 6	SYN 16 22	& 26 38	6 36 54	F 46 70	V 56 86	f 66 102	v 76 118
7	BEL 07 7	ETB 17 23	' 27 39	7 37 55	G 47 71	W 57 87	g 67 103	w 77 119
8	BS 08 8	CAN 18 24	(28 40	8 38 56	H 48 72	X 58 88	h 68 104	x 78 120
9	HT 09 9	EM 19 25	) 29 41	9 39 57	I 49 73	Y 59 89	i 69 105	y 79 121
A	LF 0A 10	SUB 1A 26	* 2A 42	: 3A 58	J 4A 74	Z 5A 90	j 6A 106	z 80 122

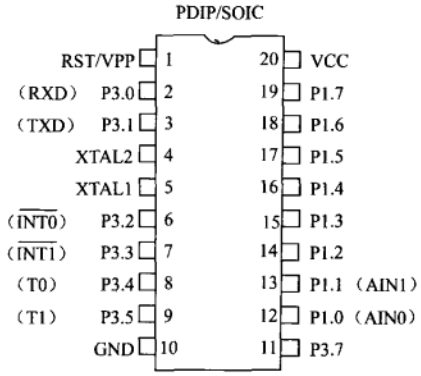
附录 B　8051 相关 IC 的引脚图



8051/8031
MICROCONTROLLER

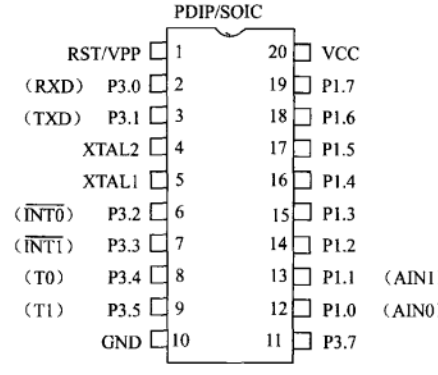


8K FLASH+2K BYTES EEPROM
AT89S8252
MICROCONTROLLER



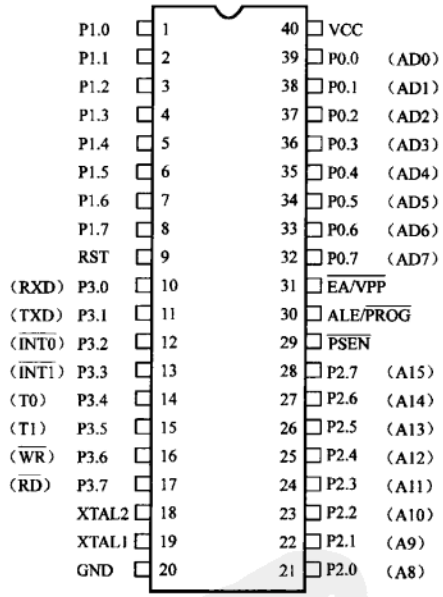
2K/4K Flash
AT89C2051/AT89C4051

AIN0 AS POSITIVE INPUT
AIN1 AS NEGATIVE INPUT
COMPARATOR OUTPUT AT P3.6

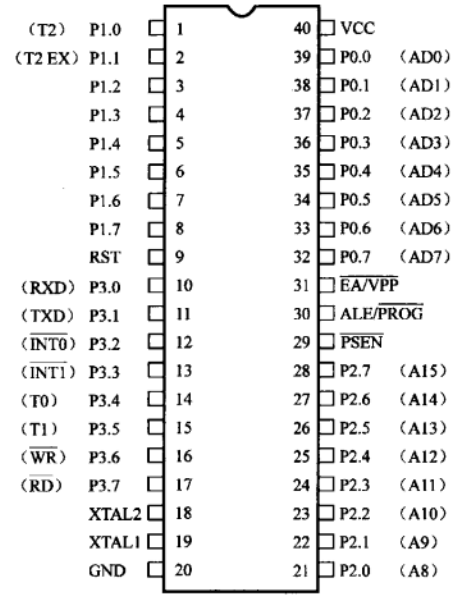


1K Flash
AT89C1051U

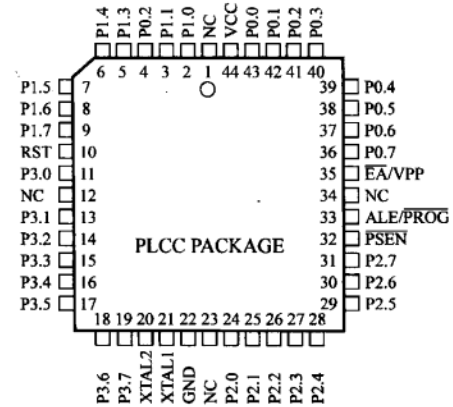
AIN0 AS POSITIVE INPUT
AIN1 AS NEGATIVE INPUT
COMPARATOR OUTPUT AT P3.6

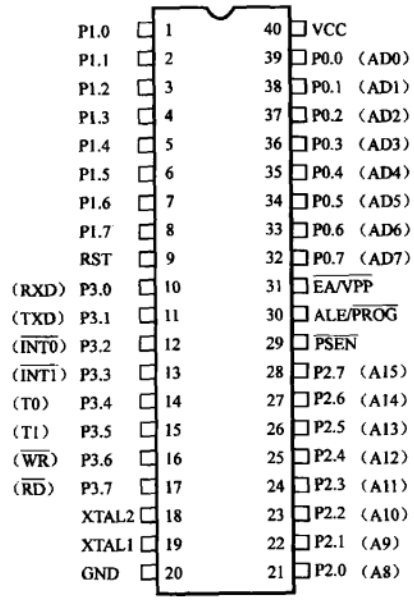


4K FLASH
AT89C51
MICROCONTROLLER

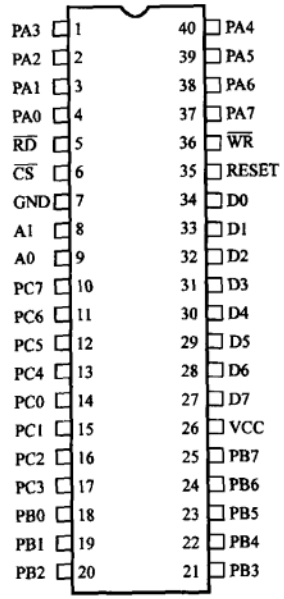


8K FLASH
AT89C52
MICROCONTROLLER



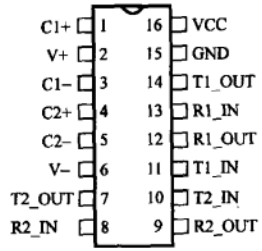


8051/8031
MIICROCONTROLLER

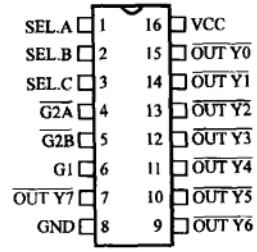


8255PPI

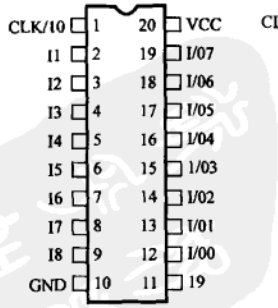
K817/TLP521
OPTO ISOLATOR



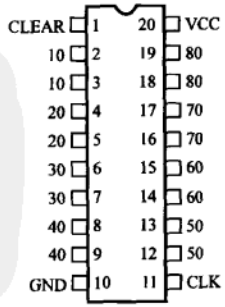
MAX232
RS-232 TRANSCEVIER



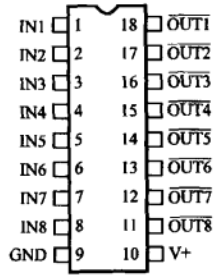
74LS138
3 TO 8 DECODER



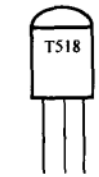
PEEL18CV8



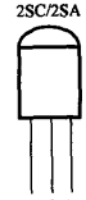
74LS273
OCTAL D-FF WITH
CLEAR INPUT



ULM2803
DARLINGTON TRANSISTER ARRAY
(OUTPUT INVERTED)



RESET IC



SMALL SIGNAL TRANSISTER

附录 C　8051 指令集总整理

影响标志位的指令整理

指　令	标　志		
	进　位	溢　出	辅助借位
ADD	V	V	V
ADDC	V	V	V
SUBB	V	V	V
MUL	0	V	
DIV	0	V	
DAA	V		
RRC	V		
RLC	V		
SETB C	1		
CLR C	0		
CPL C	V		
ANL C，bit	V		
ANL C，/bit	V		
ORL C，bit	V		
ORL C，/bit	V		
MOV C，bit	V		
CJNE	V		

Note：1. Operation on SFR byte address 208 or bit addresses 209-215
(that is，the PSW or bits in the PSW)also affect flag settings.

The Instruction Set and Addressing Modes

Rn	Register R7-R0 of the currently selected Register Bank.
direct	8-bit internal data location's address. This could be an Internal Data RAM location (0-127) or a SFR[i.e., *VO* port, control register, status register, etc. (128-255)].
@Ri	8-bit internal data RAM location (0-255) addressed indirectly through register R1 or R0.
#data	8-bit constant included in instruction.
#data 16	16-bit constant included in instruction.
addr 16	16-bit destination address. Used by LCALL and LJMP. A branch can be anywherre within the 64K byte Program Memory address space.
addr 11	11-bit destination address. Used by ACALL and AJMP. THe branch will be within the same 2K byte page of program memory as the first byte of the following instruction.
Srel	Signed (two's complement) 8-bit offset byte. Used by SJMP and all conditional jumps. Range is -128 to +127 bytes relative to first byte of the following instruction.
bit	Direct Addressed bit in Internal Data RAM or Special Function Register.

8051 指令设置目录

	0	1	2	3	4	5	6	7
0	NOP	JBC bit, tel [3B, 2C]	JB bit, rel [3B, 2C]	JNB bit, rel [3B, 2C]	JC rel [2B, 2C]	JNC rel [2B, 2C]	JZ rel [2B, 2C]	JNZ rel [2B, 2C]
1	AJMP (P0) [2B, 2C]	ACALL (P0) [2B, 2C]	AJMP (P1) [2B, 2C]	ACALL (P1) [2B, 2C]	AJMP (P2) [2B, 2C]	ACALL (P2) [2B, 2C]	AJMP (P3) [2B, 2C]	ACALL (P3) [2B, 2C]
2	LJMP addr16 [3B, 2C]	LCALL addr16 [3B, 2C]	RET [2C]	RETI [2C]	ORL dir, A [2B]	ANL dir, A [2B]	XRL dir, A [2B]	ORL C, bit [2B, 2C]
3	RR A	RRC A	RL A	RLC A	ORL dir, #data [3B, 2C]	ANL dir, #data [3B, 2C]	XRL dir, #data [3B, 2C]	JMP @A +DPTR [2C]
4	INC A	DEC A	ADD A, #data [2B]	ADDC A, #data [2B]	ORL A, #data [2B]	ANL A, #data [2B]	XRL A, #data [2B]	MOV A, #data [2B]
5	INC dir [2B]	DEC dir [2B]	ADD A, dir [2B]	ADDC A, dir [2B]	ORL A, dir [2B]	ANL A, dir [2B]	XRL A, dir [2B]	MOV dir, #data [3B, 2C]
6	INC @R0	DEC @R0	ADD A, @R0	ADDC A, @R0	ORL A, @R0	ANL A, @R0	XRL A, @R0	MOV @R0, @data [2B]
7	INC @R1	DEC @R1	ADD A, @R1	ADDC A, @R1	ORL A, @R1	ANL A, @R1	XRL A, 0R1	MOV @ R1, #data [2B]
8	INC R0	DEC R0	ADD A, R0	ADDC A, R0	ORL A, R0	ANL A, R0	XRL A, R0	MOV R0, #data [2B]
9	INC R1	DEC R1	ADD A, R1	ADDC A, R1	ORL A, R1	ANL A, R1	XRL A, R1	MOV R1, #data [2B]
A	INC R2	DEC R2	ADD A, R2	ADDC A, R2	ORL A, R2	ANL A, R2	XRL A, R2	MOV R2, #data [2B]
B	INC R3	DEC R3	ADD A, R3	ADDC A, R3	ORL A, R3	ANL A, R3	XRL A, R3	MOV R3, #data [2B]
C	INC R4	DEC R4	ADD A, R4	ADDC A, R4	ORL A, R4	ANL A, R4	XRL A, R4	MOV R4, #data [2B]
D	INC R5	DEC R5	ADD A, R5	ADDC A, R5	ORL A, R5	ANL A, R5	XRL A, R5	MOV R5, #data [2B]
E	INC R6	DEC R6	ADD A, R6	ADDC A, R6	ORL A, R6	ANL A, R6	XRL A, R6	MOV R6, #data [2B]
F	INC R7	DEC R7	ADD A, R7	ADDC A, R7	ORL A, R7	ANL A, R7	XRL A, R7	MOV R7, #data [2B]

Note: [2B]=2 Byte, [3B]=3Byte, [2C]=2Cycle, [4C]=4 Cycle, Blank=1 Byte/1 Cycle

8051 指令设置目录（续表）

	8	9	A	B	C	D	E	F
0	SJMP REL [2B, 2C]	MOV DPTR, #data 16 [3B, 2C]	ORL C,/bit [2B, 2C]	ANL C,/bit [2B, 2C]	PUSH dir [2B, 2C]	POP dir [2B, 2C]	MOVX A, @DPTR [2C]	MOVX @DPTR, A [2C]
1	AJMP (P4) [2B, 2C]	ACALL (P4) [2B, 2C]	AJMP (P5) [2B, 2C]	ACALL (P5) [2B, 2C]	AJMP (P6) [2B, 2C]	ACALL (P6) [2B, 2C]	AJMP (P7) [2B, 2C]	ACALL (P7) [2B, 2C]
2	ANL C, bit [2B, 2C]	MOV bit, C [2B, 2C]	MOV C, bit [2B]	CPL bit [2B]	CLR bit [2B]	SETB bit [2B]	MOVX A, @R0 [2C]	MOVX @ R0, A [2C]
3	MOVC A, @A + PC [2C]	MOVC A, @A + DPTR [2C]	INC DPTR [2C]	CPL C	CLR C	SETB C	MOVX A, @RI [2C]	MOVX @ RI, A [2C]
4	DIV AB [2B, 4C]	SUBB A, #data [2B]	MUL AB [4C]	CJNE A, #data, rel [3B, 2C]	SWAP A	DA A	CLR A	CPL A
5	MOV dir, dir [3B, 2C]	SUBB A, dir [2B]		CJNE A, dir, rel [3B, 2C]	XCH A, dir [2B]	DJNZ dir, rel [3B, 2C]	MOV A, dir [2B]	MOV dir, A [2B]
6	MOV dir, @R0 [2B, 2C]	SUBB A, @R0	MOV @R0, dir [2B, 2C]	CJNE @R0, #data, rel [3B, 2C]	XCH A, @R0	XCHD A, @R0	MOV A, @R0	MOV @R0, A
7	MOV dir, @R1 [2B, 2C]	SUBB A, @R1	MOV @R1, dir [2B, 2C]	CJNE @Rl,#data, rel [3B,2C]	XCH A, @R1	XCHD A, @R1	MOV A, @R1	MOV OR1, A
8	MOV dir, R0 [2B, 2C]	SUBB A, R0	MOV R0, dir [2B, 2C]	CJNE R0, #data, rel [3B, 2C]	XCH A, R0	DJNZ R0, rel [2B, 2C]	MOV A, R0	MOV R0, A
9	MOV dir, R1 [2B, 2C]	SUBB A, R1	MOV R1, dir [2B, 2C]	CJNE R1, #data, rel [3B, 2C]	XCH A, R1	DJNZ R1, rel [2B, 2C]	MOV A, R1	MOV R1,A
A	MOV dir, R2 [2B, 2C]	SUBB A, R2	MOV R2, dir [2B, 2C]	CJNE R2,#data, rel [3B, 2C]	XCH A, R2	DJNZ R2, rel [2B, 2C]	MOV A, R2	MOV R2, A
B	MOV dir, R3 [2B, 2C]	SUBB A, R3	MOV R3, dir [2B, 2C]	CJNE R3, #data, rel [3B, 2C]	XCH A, R3	DJNZ R3, rel [2B, 2C]	MOV A, R3	MOV R3, A
C	MOV dir, R4 [2B, 2C]	SUBB A, R4	MOV R4, dir [2B, 2C]	CJNE R4,#data, rel [3B, 2C]	XCH A, R4	DJNZ R4, rel [2B, 2C]	MOV A, R4	MOV R4, A
D	MOV dir, R5 [2B, 2C]	SUBB A, R5	MOV R5, dir [2B, 2C]	CJNE R5, #data, rel [3B, 2C]	XCH A, R5	DJNZ R5, rel [2B, 2C]	MOV A, R5	MOV R5, A
E	MOV dir, R6 [2B, 2C]	SUBB A, R6	MOV R6, dir [2B, 2C]	CJNE R6,#data, rel [3B, 2C]	XCH A, R6	DJNZ R6, rel [2B, 2C]	MOV A, R6	MOV R6, A
F	MOV dilr, R7 [2B, 2C]	SUBB A, R7	MOV R7, dir [2B, 2C]	CJNE R7,#data, rel [3B, 2C]	XCH A, R7	DJNZ R7, rel [2B, 2C]	MOV A, R7	MOV R7, A

Note: [2B]=2 Byte, [3B]=3Byte, [2C]=2Cycle, [4C]=4 Cycle, Blank=1 Byte/1 Cycle

8051 指令集——单字节指令整理 1

	0	1	2	3	4	5	6	7
0	NOP							
1								
2			RET [2C]	RET [2C]				
3	RR A	RRC A	RL A	RLC A				JMP @A+DPTR [2C]
4	INC A	DEC A						
5								
6	INC @R0	DEC @R0	ADD A, @R0	ADDC A, @R0	ORL A, @R0	ANL A, @R0	XRL A, @R0	
7	INC @R1	DEC @R1	ADD A, @R1	ADDC A, @R1	ORL A, @R1	ANL A, @R1	XRL A, @R1	
8	INC R0	DEC R0	ADD A, R0	ADDC A, R0	ORL A, R0	ANL A, R0	XRL A, R0	
9	INC R1	DEC R1	ADD A, R1	ADDC A, R1	ORL A, R1	ANL A, R1	XRL A, R1	
A	INC R2	DEC R2	ADD A, R2	ADDC A, R2	ORL A, R2	ANL A, R2	XRL A, R2	
B	INC R3	DEC R3	ADD A, R3	ADDC A, R3	ORL A, R3	ANL A, R3	XRL A, R3	
C	INC R4	DEC R4	ADD A, R4	ADDC A, R4	ORL A, R4	ANL A, R4	XRL A, R4	
D	INC R5	DEC R5	ADD A, R5	ADDC A, R5	ORL A, R5	ANL A, R5	XRL A, R5	
E	INC R6	DEC R6	ADD A, R6	ADDC A, R6	ORL A, R6	ANL A, R6	XRL A, R6	
F	INC R7	DEC R7	ADD A, R7	ADDC A, R7	ORL A, R7	ANL A, R7	XRL A, R7	

Note：[2B]=2 Byte,[3B]=3Byte,[2C]=2Cycle,[4C]=4Cycle,Blank=1Byte/1Cycle

8051 指令集—单字节指令整理 2

	8	9	A	B	C	D	E	F
0							MOVX A, @DPTR [2C]	MOVX @ DPTR, A [2C]
1								
2							MOVX A, @R0 [2C]	MOVX @ R0, A [2C]
3	MOVC A, @A+ PC [2C]	MOVC A, @ A + DPTR [2C]	INC DPTR [2C]	CPL C	CLR C	SETB C	MOVX A, @RI [2C]	MOVX @RI, A [2C]
4			MUL AB [4C]		SWAP A	DA A	CLR A	CPL A
5								
6		SUBB A,@R0			XCH A, @R0	XCHD A, @R0	MOV A, @R0	MOV @R0,A
7		SUBB A, @R1			XCH A, @R1	XCHD A,@R1	MOV A, @R1	MOV @R1,A
8		SUBB A, R0			XCH A, R0		MOV A, R0	MOV R0, A
9		SUBB A, R1			XCH A, R1		MOV A, R1	MOV R1,A
A		SUBB A, R2			XCH A, R2		MOV A, R2	MOV R2, A
B		SUBB A, R3			XCH A, R3		MOV A, R3	MOV R3, A
C		SUBB A, R4			XCH A, R4		MOV A, R4	MOV R4, A
D		SUBB A, R5			XCH A, R5		MOV A, R5	MOV R5, A
E		SUBB A, R6			XCH A, R6		MOV A, R6	MOV R6, A
F		SUBB A, R7			XCH A, R7		MOV A, R7	MOV R7, A

Note:[2B]=2Byte，[3B]=3Byte，[2C]=2Cycle，[4C]=4Cycle，Blank=1Byte/1Cycle

8051 指令集——双字节指令整理 1

	0	1	2	3	4	5	6	7
0					JC rel [2B, 2C]	JNC rel [2B, 2C]	JZ rel [2B, 2C]	JNZ rel [2B, 2C]
1	AJMP (P0) [2B, 2C]	ACALL (P0) [2B, 2C]	AJMP (P1) [2B, 2C]	ACALL (P1) [2B, 2C]	AJMP (P2) [2B, 2C]	ACALL (P2) [2B, 2C]	AJMP (P3) [2B, 2C]	ACALL (P3) [2B, 2C]
2					ORL dir, A [2B]	ANL dir, A [2B]	XRL dir, A [2B]	ORL C, bit [2B, 2C]
3								
4			ADD A, #data [2B]	ADDC A, #data [2B]	ORL A, #data [2B]	ANL A, #data [2B]	XRL A, #data [2B]	MOV A, #data [2B]
5	INC dir [2B]	DEC dir [2B]	ADD A, dir [2B]	ADDC A, dir [2B]	ORL A, dir [2B]	ANL A, dir [2B]	XRL A, dir [2B]	
6								MOV @R0, @data [2B]
7								MOV @ R1, #data [2B]
8								MOV R0, #data [2B]
9								MOV R1,#data [2B]
A								MOV R2, #data [2B]
B								MOV R3,#data [2B]
C								MOV R4,#data [2B]
D								MOV R5,#data [2B]
E								MOV R6,#data [2B]
F								MOV R7, #data [2B]

Note: [2B]=2 Byte, [3B]=3Byte, [2C]=2Cycle, [4C]=4 Cycle, Blank=1Byte/1Cycle

8051 指令集——双字节指令整理 2

	8	9	A	B	C	D	E	F
0	SJMP REL [2B, 2C]		ORL C,/bit [2B, 2C]	ANL C,/bit [2B, 2C]	PUSH dir [2B, 2C]	POP dir [2B, 2C]		
1	AJMP (P4) [2B, 2C]	ACALL (P4) [2B, 2C]	AJMP (P5) [2B, 2C]	ACALL (P5) [2B, 2C]	AJMP (P6) [2B, 2C]	ACALL (P6) [2B, 2C]	AJMP (P7) [2B, 2C]	ACALL (P7) [2B, 2C]
2	ANL C, bit [2B, 2C]	MOV bit, C [2B, 2C]	MOV C, bit [2B]	CPL bit [2B]	CLR bit [2B]	SETB bit [2B]		
3								
4	DIV AB [2B, 4C]	SUBB A, #data [2B]						
5		SUBB A, dir [2B]			XCH A, dir [2B]		MOV A, dir [2B]	MOV dir, A [2B]
6	MOV dir, @R0 [2B, 2C]		MOV @R0, dir [2B, 2C]					
7	MOV dir, @R1 [2B, 2C]		MOV @R1, dir [2B, 2C]					
8	MOV dir, R0 [2B, 2C]		MOV R0, dir [2B, 2C]			DJNZ R0, rel [2B, 2C]		
9	MOV dir, R1 [2B, 2C]		MOV R1, dir [2B, 2C]			DJNZ Rl, rel [2B, 2C]		
A	MOV dir, R2 [2B, 2C]		MOV R2, dir [2B, 2C]			DJNZ R2, rel [2B, 2C]		
B	MOV dir, R3 [2B, 2C]		MOV R3, dir [2B, 2C]			DJNZ R3, rel [2B, 2C]		
C	MOV dir, R4 [2B, 2C]		MOV R4, dir [2B, 2C]			DJNZ R4, rel [2B, 2C]		
D	MOV dir, R5 [2B, 2C]		MOV R5, dir [2B, 2C]			DJNZ R5, rel [2B, 2C]		
E	MOV dir, R6 [2B, 2C]		MOV R6, dir [2B, 2C]			DJNZ R6, rel [2B, 2C]		
F	MOV dir, R7 [2B, 2C]		MOV R7, dir [2B, 2C]			DJNZ R7, rel [2B, 2C]		

Note: [2B]=2Byte, [3B]=3Byte, [2C]=2Cycle, [4C]=4Cycle, Blank=1Byte/1Cycle

8051 指令集——3 字节指令整理 1

	0	1	2	3	4	5	6	7
0		JBC bit, rel [3B, 2C]	JB bit, rel [3B, 2C]	JNB bit, rel [3B, 2C]				
1								
2	LJMP addr16 [3B, 2C]	LCALL addr16 [3B, 2C]						
3					ORL dir, #data [3B, 2C]	ANL dir, #data [3B, 2C]	XRL dir, #data [3B, 2C]	
4								
5								MOV dir, #data [3B, 2C]
6								
7								
8								
9								
A								
B								
C								
D								
E								
F								

Note: [2B]=2Byte, [3B]=3Byte, [2C]=2Cycle, [4C]=4Cycle, Blank=1Byte/1Cycle

8051 指令集——3 字节指令整理 2

	8	9	A	B	C	D	E	F
0		MOV DPTR, #data 16 [3B, 2C]						
1								
2								
3								
4				CJNE A, #data, rel [3B, 2C]				
5	MOV dir, dir [3B, 2C]			CJNE A, dir, rel [3B, 2C]		DJNZ dir, rel [3B, 2C]		
6				CJNE @R0, #data, rel [3B, 2C]				
7				CJNE @R1, #data, rel [3B, 2C]				
8				CJNE R0, #data, rel [3B, 2C]				
9				CJNE R1, #data, rel [3B, 2C]				
A				CJNE R2, #data, rel [3B, 2C]				
B				CJNE R3, #data, rel [3B, 2C]				
C				CJNE R4, #data, rel [3B, 2C]				
D				CJNE R5,#data, rel [3B, 2C]				
E				CJNE R6,#data, rel [3B, 2C]				
F				CJNE R7, #data, rel [3B, 2C]				

Note: [2B]=2Byte，[3B]=3Byte,[2C]=2Cyle，[4C]=4Cyle，Blank=1Byte/1Cycle

附录 D 8051 指令整理（按功能区分）

助记符		描 述	字 节	振荡周期
算术运算				
ADD	A,Rn	Add register to Accumulator	1	12
ADD	A,direct	Add direct byte to Accumulator	2	12
ADD	A,@Rj	Add indirect RAM to Accumulator	1	12
ADD	A,#data	Add immediate data to Accumulator	2	12
ADDC	A,Rn	Add register to Accumulator with Carry	1	12
ADDC	A,direct	Add direct byte to Accumulator with Carry	2	12
ADDC	A,@Ri	Add indirect RAM to Accumulator with Carry	1	12
ADDC	A,#data	Add immediate data to Acc with Carry	2	12
SUBB	A,Rn	Subtract Register from Acc with borrow	1	12
SUBB	A,direct	Subtract direct byte from Acc with borrow	2	12
SUBB	A,@Ri	Subtract indirect RAM from ACC with borrow	1	12
SUBB	A,#data	Subtract immediate data from Acc with borrow	2	12
INC	A	Increment Accumulator	1	12
INC	Rn	Increment register	1	12
INC	direct	Increment direct byte	2	12
INC	@Ri	Increment direct RAM	1	12
DEC	A	Decrement Accumulator	1	12
DEC	Rn	Decrement register	1	12
DEC	direct	Derement direct byte	2	12
DEC	@Ri	Decrement indirect RAM	1	12
INC	DPTR	Increment Data Pointer	1	24
MUL	AB	Multiply A&B	1	48
DIV	AB	Divide A by B	1	48
DA	A	Decimal Adjust Accumulator	1	12

Note: 1.All mnemonics copyrighted @Intel Corp., 1980.

助记符		描　述	字　节	振荡周期
逻辑运算				
ANL	A,Rn	AND register to Accumulator	1	12
ANL	A,direct	AND direct byte to Accumulator	2	12
ANL	A,@Ri	AND indirect RAM to Accumulator	1	12
ANL	A,#data	AND immediate data to Accumulator	2	12
ANL	direct,A	AND Accumulator to direct byte	2	12
ANL	direct,#data	AND immediate data to direct byte	3	24
ORL	A,Rn	OR register to Accumulator	1	12
ORL	A,direct	OR direct byte to Accumulator	2	12
ORL	A,@Ri	OR indirect RAM to Accumulator	1	12
ORL	A,#data	OR immediate data to Accumulator	2	12
ORL	direct,A	OR Accumulator to direct byte	2	12
ORL	direct,#data	ORimmediate data to direct byte	3	24
XRL	A,Rn	Exclusive-OR register to Accumulator	1	12
XRL	A,direct	Exclusive-OR direct byte to Accumulator	2	12
XRL	A,@Ri	Exclusive-OR indirect RAM to Accumulator	1	12
XRL	A,#data	Exclusive-OR immediate data to Accumulator	2	12
XRL	direct,A	Exclusive-OR Accumulator to direct byte	2	12
XRL	direct,#data	Exclusive-OR immediate data to direct byte	3	24
CLR	A	Clear Accumulator	1	12
CPL	A	Complement Accumulator	1	12
RL	A	Rotate Accumulator Left	1	12
RLC	A	Rotate Accumulator Left through the Carry	1	12
RR	A	Rotate Accumulator Right	1	12
RRC	A	Rotate Accumulator Right through the Carry	1	12
SWAP	A	Swap nibbles within the Accumulator	1	12

助记符		描　述	字　节	振荡周期
数据传送				
MOV	A,Rn	Move register to Accumulator	1	12
MOV	A,direct	Move direct byte to Accumulator	2	12
MOV	A,@Ri	Move indirect RAM to Accumulator	1	12
MOV	A,#data	Move immediate data to Accumulator	2	12
MOV	Rn,A	Move Accumulator to register	1	12
MOV	Rn,direct	Move direct byte to register	2	24
MOV	Rn,#data	Move immediate data to register	2	12
MOV	direct,A	Move Accumulator to direct byte	2	12
MOV	direct,Rn	Move register to direct byte	2	24
MOV	direct,direct	Move direct byte to direct	3	24
MOV	direct,@Ri	Move indirect RAM to direct byte	2	24
MOV	direct,#data	Move immediate data to direct byte	3	24
MOV	@ Ri,A	Move Accumulator to indirect RAM	1	12
MOV	@Ri,direct	Move direct byte to indirect RAM	2	24
MOV	@ Ri,#data	Move immediate data to indirect RAM	2	12
MOV	DPTR,#data 16	Load Data Pointer with a 16-bit constant	3	24
MOVC	A,@A+DPTR	Move Code byte relative to DPTR to Acc	1	24
MOVC	A,@A+PC	Move Code byte relative to DPTR to Acc	1	24
MOVX	A,@Ri	Move External RAM (8-bit addr) to Acc	1	24
MOVX	A,@DPTR	Move Exemal RAM(16-bit addr) to Acc	1	24
MOVX	@ Ri,A	Move Acc to External RAM (8-bit ddr)	1	24
MOVX	@DPTR,A	Move Acc to External RAM (16-bit addr)	1	24
PUSH	direct	Push direct byte onto stack	2	24
POP	direct	Pop direct byte from stack	2	24
XCH	A,Rn	Exchange register with Accumulator	1	12
XCH	A,direct	Exchange direct byte with Accumulator	2	12
XCH	A,@Ri	Exchange indirect RAM with Accumulator	1	12
XCHD	A,@Ri	Exchange low-order Digit indirect RAM with Acc	1	12

助记符		描　述	字　节	振荡周期
布尔运算				
CLR	C	Clear Carry	1	12
CLR	bit	Clear diret bit	2	12
SETB	C	Set Carry	1	12
SETB	bit	Set direct bit	2	12
CPL	C	Complement Carry	1	12
CPL	bit	Complement direct bit	2	12
ANL	C,bit	AND direct bit to CARRY	2	24
ANL	C,/bit	AND complement of direct bit to Carry	2	24
ORL	C,bit	OR direct bit to Carry	2	24
ORL	C,/bit	OR complement of direct bit to Carry	2	24
MOV	C,bit	Move direct bit to Carry	2	12
MOV	bit,C	Move Carry to direct bit	2	24
JC	REL	Jump if Carry is set	2	24
JNC	REL	Jump if Carry not set	2	24
JB	bit,rel	Jump if direct Bit is set	3	24
JNB	bit,rel	Jump if direct Bit is Not set	3	24
JBC	bit,rel	Jump if direct Bit is set & clear bit	3	24

助记符		描　述	字　节	振荡周期
跳转指令				
ACALL	addr11	Absolute Subroutine Call	2	24
LCALL	addr16	Long Subroutine Call	3	24
RET		Return from Subroutine	1	24
RETI		Return from interrupt	1	24
AJMP	addr11	Absolute Jump	2	24
LJMP	addr16	Long Jump	3	24
SJMP	rel	Short Jump (relative addr)	2	24
JMP	@A+DPTR	Jump indirect relative to the DPTR	1	24
JZ	rel	Jump if Accumulator is Zero	2	24
JNZ	rel	Jump if Accumulator is Not Zero	2	24
CJNE	A,direct,rel	Compare direct byte to Acc and Jump if Not Equal	3	24
CJNE	A,#data,rel	Compare immediate to Acc and Jump if Not Equal	3	24
CJNE	Rn,#data,rel	Compare immediate to register and Jump if Not Equal	3	24
CJNE	@ Ri,#data,rel	Compare immediate to indirect and Jump if Not Equal	3	24
DJNZ	Rn,rel	Decrement register and Jump if Not Zero	2	24
DJNZ	direct,rel	Decrement direct byte and Jump if Not Zero	3	24
NOP		No Operation	1	12

附录 E　8051 指令整理（按十六进制排列）

十六进制编码	字节数	助记符	操作数
00	1	NOP	
01	2	AJMP	code addr
02	3	LJMP	code addr
03	1	RR	A
04	1	INC	A
05	2	INC	data addr
06	1	INC	@R0
07	1	INC	@R1
08	1	INC	R0
09	1	INC	R1
0A	1	INC	R2
0B	1	INC	R3
0C	1	INC	R4
0D	1	INC	R5
0E	1	INC	R6
0F	1	INC	R7
10	3	JBC	bit addr,code addr
11	2	ACALL	code addr
12	3	LCALL	code addr
13	1	RRC	A
14	1	DEC	A
15	2	DEC	data addr
16	1	DEC	@ R0
17	1	DEC	@R1
18	1	DEC	R0
19	1	DEC	R1
1A	1	DEC	R2
1B	1	DEC	R3
1C	1	DEC	R4
1D	1	DEC	R5
1E	1	DEC	R6
1F	1	DEC	R7

续表

十六进制编码	字节数	助记符	操作数
20	3	JB	bit addr, code addr
21	2	AJMP	code addr
22	1	RET	
23	1	RL	A
24	2	ADD	A,#data
25	2	ADD	A,data addr
26	1	ADD	A,@R0
27	1	ADD	A,@R1
28	1	ADD	A,R0
29	1	ADD	A,R1
2A	1	ADD	A,R2
2B	1	ADD	A,R3
2C	1	ADD	A,R4
2D	1	ADD	A,R5
2E	1	ADD	A,R6
2F	1	ADD	A,R7
30	3	JNB	bit addr,code addr
31	2	ACALL	code addr
32	1	RETI	
33	1	RLC	A
34	2	ADDC	A,#data
35	2	ADDC	A,data addr
36	1	ADDC	A,@R0
37	1	ADDC	A,@R1
38	1	ADDC	A,R0
39	1	ADDC	A,R1
3A	1	ADDC	A,R2
3B	1	ADDC	A,R3
3C	1	ADDC	A,R4
3D	1	ADDC	A,R5
3E	1	ADDC	A,R6
3F	1	ADDC	A,R7
40	2	JC	code addr
41	2	AJMP	code addr

续表

十六进制编码	字节数	助记符	操作数
42	2	ORL	data add,A
43	3	ORL	data addr,#data
44	2	ORL	A,#data
45	2	ORL	A,data addr
46	1	ORL	A,@R0
47	1	ORL	A,@R1
48	1	ORL	A,R0
49	1	ORL	A,R1
4A	1	ORL	A,R2
4B	1	ORL	A,R3
4C	1	ORL	A,R4
4D	1	ORL	A,R5
4E	1	ORL	A,R6
4F	1	ORL	A,R7
50	2	JNC	code addr
51	2	ACALL	code addr
52	2	ANL	data addr,A
53	3	ANL	data addr,#data
54	2	ANL	A,#data
55	2	ANL	A,data addr
56	1	ANL	A,@R0
57	1	ANL	A,@R1
58	1	ANL	A,R0
59	1	ANL	A,R1
5A	1	ANL	A,R2
5B	1	ANL	A,R3
5C	1	ANL	A,R4
5D	1	ANL	A,R5
5E	1	ANL	A,R6
5F	1	ANL	A,R7
60	2	JZ	code addr
61	2	AJMP	code addr
62	2	XRL	data addr,A
63	3	XRL	data addr,#data
64	2	XRL	A,#data
65	2	XRL	A,data addr
66	1	XRL	A,@R0
67	1	XRL	A,@R1
68	1	XRL	A,R0
69	1	XRL	A,R1
6A	1	XRL	A,R2

续表

十六进制编码	字节数	助记符	操作数
6B	1	XRL	A,R3
6C	1	XRL	A,R4
6D	1	XRL	A,R5
6E	1	XRL	A,R6
6F	1	XRL	A,R7
70	2	JNZ	code addr
71	2	ACALL	code addr
72	2	ORL	C,bit addr
73	1	JMP	@A+DPTR
74	2	MOV	A,#data
75	3	MOV	data addr,#data
76	2	MOV	@ R0,#data
77	2	MOV	@ R1,#data
78	2	MOV	R0,#data
79	2	MOV	R1,#data
7A	2	MOV	R2,#data
7B	2	MOV	R3,#data
7C	2	MOV	R4,#data
7D	2	MOV	R5,#data
7E	2	MOV	R6,#data
7F	2	MOV	R7,#data
80	2	SJMP	code addr
81	2	AJMP	code addr
82	2	ANL	C,bit addr
83	1	MOVC	A,@A+PC
84	1	DIV	AB
85	3	MOV	data addr, data addr
86	2	MOV	data addr,@ R0
87	2	MOV	data addr,@ R1
88	2	MOV	data addr, R0
89	2	MOV	data addr,R1
8A	2	MOV	data addr, R2
8B	2	MOV	data addr, R3
8C	2	MOV	data addr,R4
8D	2	MOV	data addr, R5
8E	2	MOV	data addr,R6
8F	2	MOV	data addr, R7
90	3	MOV	DPTR,#data
91	2	ACALL	code addr
92	2	MOV	bit addr,C

续表

十六进制编码	字节数	助记符	操作数
93	1	MOVC	A,@A+DPTR
94	2	SUBB	A,#data
95	2	SUBB	A,data addr
96	1	SUBB	A,@R0
97	1	SUBB	A,@R1
98	1	SUBB	A,R0
99	1	SUBB	A,R1
9A	1	SUBB	A,R2
9B	1	SUBB	A,R3
9C	1	SUBB	A,R4
9D	1	SUBB	A,R5
9E	1	SUBB	A,R6
9F	1	SUBB	A,R7
A0	2	ORL	C,/bit addr
A1	2	AJMP	code addr
A2	2	MOV	C,bit addr
A3	1	INC	DPTR
A4	1	MUL	AB
A5		reserved	
A6	2	MOV	@ R0,data addr
A7	2	MOV	@ R1 ,data addr
A8	2	MOV	R0,data addr
A9	2	MOV	R1 ,data addr
AA	2	MOV	R2,data addr
AB	2	MOV	R3,data addr
AC	2	MOV	R4,data addr
AD	2	MOV	R5,data addr
AE	2	MOV	R6,data addr
AF	2	MOV	R7,data addr
B0	3	ANL	C,/bit addr
B1	2	ACALL	code addr
B2	2	CPL	bit addr
B3	1	CPL	C
B4	3	CJNE	A,#data,code addr
B5	3	CJNE	A,data addr,code addr
B6	3	CJNE	@ R0,#data,code addr
B7	3	CJNE	@ R1 ,#data,code addr
B8	3	CJNE	R0,#data,code addr
B9	3	CJNE	R1 ,#data,code addr
BA	3	CJNE	R2,#data,code addr

续表

十六进制编码	字节数	助记符	操作数
BB	3	CJNE	R3,#data,code addr
BC	3	CJNE	R4,#data,code addr
BD	3	CJNE	R5,#data,code addr
BE	3	CJNE	R6,#data,code addr
BF	3	CJNE	R7,#data,code addr
C0	2	PUSH	data addr
C1	2	AJMP	code addr
C2	2	CLR	bit addr
C3	1	CLR	C
C4	1	SWAP	A
C5	2	XCH	A,data addr
C6	1	XCH	A,@R0
C7	1	XCH	A,@R1
C8	1	XCH	A,R0
C9	1	XCH	A,R1
1CA	1	XCH	A,R2
CB	1	XCH	A,R3
CC	1	XCH	A,R4
CD	1	XCH	A,R5
CE	1	XCH	A,R6
CF	1	XCH	A,R7
D0	2	POP	data addr
D1	2	ACALL	code addr
D2	2	SETB	bit addr
D3	1	SETB	C
D4	1	DA	A
D5	3	DJNZ	A,data addr, code addr
D6	1	XCHD	A,@R0
D7	1	XCHD	A,@R1
D8	2	DJNZ	R0,code addr
D9	2	DJNZ	R1,code addr
DA	2	DJNZ	R2,code addr
DB	2	DJNZ	R3,code addr
DC	2	DJNZ	R4,code addr
DD	2	DJNZ	R5,code addr
DE	2	DJNZ	R6,code addr
DF	2	DJNZ	R7,code addr
E0	1	MOVX	A, @ DPTR
E1	2	AJMP	code addr
E2	1	MOVX	A,@R0

续表

十六进制编码	字节数	助记符	操作数
E3	1	MOVX	A,@R1
E4	1	CLR	A
E5	2	MOV	A,data addr
E6	1	MOV	A,@R0
E7	1	MOV	A,@R1
E8	1	MOV	A,R0
E9	1	MOV	A,R1
EA	1	MOV	A,R2
EB	1	MOV	A,R3
EC	1	MOV	A,R4
ED	1	MOV	A,R5
EE	1	MOV	A,R6
EF	1	MOV	A,R7
F0	1	MOVX	@ DPTR, A
F1	2	ACALL	code addr
F2	1	MOVX	@ R0,A
F3	1	MOVX	@ R1 ,A
F4	1	CPL	A
F5	2	MOV	data addr A
F6	1	MOV	@ R0,A
F7	1	MOV	@R1,A
F8	1	MOV	R0,A
F9	1	MOV	R1,A
FA	1	MOV	R2,A
FB	1	MOV	R3,A
FC	1	MOV	R4,A
FD	1	MOV	R5,A
FE	1	MOV	R6,A
FF	1	MOV	R7,A

附录 F　8051 SFR 表与 RESET 后的初始值

F8H									FFH
F0H	B 00000000								F7H
E8H									EFH
E0H	ACC 00000000								E7H
D8H									DFH
D0H	PSW 00000000								D7H
C8H	T2CON 00000000	T2MOD XXXXXX00	RCAP2L 00000000	RCAP2H 00000000	TL2 00000000	TH2 00000000			CFH
C0H									C7H
B8H	IP XX000000								BFH
B0H	P3 11111111								B7H
A8H	IE 0X000000								AFH
A0H	P2 11111111								A7H
98H	SCON 00000000	SBUF XXXXXXXX							9FH
90H	P1 11111111								97H
88H	TCON 00000000	TMOD 00000000	TL0 00000000	TL1 00000000	TH0 00000000	TH1 00000000	AUXR XXX00XX0		8FH
80H	P0 11111111	SP 00000111	DP0L 00000000	DP0H 00000000	DP1L 00000000	DP1H 00000000		PCON 0XXX0000	87H

附录 G　SFR 特殊功能寄存器整理表

寄存器名称	地址	可位寻址	8051	8052
ACC 累加器	EOH	*	*	*
B 通用寄存器	FOH	*	*	*
PSW 程序状态字	DOH	*	*	*
SP 堆栈指针	81H		*	*
DPH 数据指针（高位）	83H		*	*
DPL 数据指针（低位）	82H		*	*
P0 端口 0	80H	*	*	*
P1 端口 1	90H	*	*	*
P2 端口 2	A0H	*	*	*
P3 端口 3	B0H	*	*	*
IP 中断优先级控制器	B8H	*	*	*
IE 中断使能控制	A8H	*	*	*
TMOD 定时/计数模式控制	89H		*	*
TCON 定时/计数控制	88H	*	*	*
T2CON 第 2 定时/计数控制	C8H	*		*
TH0 TIMER0 高位设定	8CH		*	*
TL0 TIMER0 低位设定	8AH		*	*
TH1 TIMER1 高位设定	8DH		*	*
TL1 TIMER1 低位设定	8BH		*	*
TH2 TIMER2 高位设定	CDH			*
TL2 TIMER2 低位设定	CCH			*
RCAP2H 捕获寄存器（高位）	CBH			*
RCAP2L 捕获寄存器（低位）	CAH			*
SCON 串行通信控制器	98H	*	*	*
SBUF 串行数据缓冲器	99H		*	*
PCON 电源控制寄存器	87H		*	*

PSW 各位定义

（D0H）

CY	AC	F0	RS1	RS0	0V	—	P
D7	D6	D5	D4	D3	D2	D1	D0

CY(PSW.7)：　运算进位标志位
AC(PSW.6)：　辅助运算标志位
F0(PSW.5)：　用户自行定义用标志位
RS1(PSW.4)：　R0~R7寄存器组选用位
RS0(PSW.3)：　R0～R7寄存器组选用位
0V(PSW.2)：　运算溢位标志位
-(PSW.1)：　Intel保留待未来开发用
P(PSW.0)：　ACC内容校验标志位

注：

（1）RS1和RS0共可组成4个寄存器组(BANK)，以切换数据存储区内的R0~R7值。

RS1，RS0=00，选用BANK 0(00H～07H)

RS1，RS0=01，选用BANK 1(08H～0FH)

RS1，RS0=10，选用BANK 2(10H～17H)

RS1，RS0=11，选用 BANK 3(18H～1FH)

（2）PSW.0 即 Parity 校验位，通常在串行通信中，做数据检查用。

T2CON 各位定义

TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T2	CP/RL2
D7	D6	D5	D4	D3	D2	D1	D0

TF2 (T2CON.7)：TIMER2的溢位标志位，可由软件清除

EXF2 (T2CON.6)：TIMER2的外部标志位

RCLK (T2CON.5)：串行接收时基标志位

TCLK (T2CON.4)：串行传送时基标志位

EXEN2(T2CON.3)：TIMER2外部使能标志位

TR2 (T2CON.2)：TIMER2开始/停止控制位

C/T2 (T2CON.1)：TIMER2选定计数或定时的功能

CP/RL2 (T2CON.0)：TIMER2的Capture/Reload标志位

注：8052的TIMER 2的操作较复杂，详细操作请参考《8051单片机彻底研究——实习篇》第6章8051与8052的差异中的介绍和说明。

SCON 串行控制端口位说明

SM0	SM1	SM2	REN	TB8	RB8	TI	RI
D7	D6	D5	D4	D3	D2	D1	D0

SM0(SCON.7)：串行通信模式设置位

SM1(SCON.6)：串行通信模式设置位

SM2(SCON.5)：多重处理器通信连接的功能控制

REN(SCON.4)：串行通信接收使能，该位可用软件控制

TB8(SCON.3)：串行通信第2和第3模式下欲送出的第9个位，可用软件设置或清除

RB8(SCON.2)：串行通信第2和第3模式下欲送出的第9个位，可用软件读入再做判断

TI(SCON.1)：串行传送中断标志位

RI(SCON.0)：串行接收中断标志位

PCON 电源控制寄存器各位说明

SMOD	—	—	—	GF1	GF0	PD	IDL
D7	D6	D5	D4	D3	D2	D1	D0

SMOD(PCON.7)：双倍波特率设定

- (PCON.6)：原厂保留

- (PCON.5)：　原厂保留
- (PCON.4)：　原厂保留
GF1(PCON.3)：　一般用途标志位（由用户自行定义）
GF0(PCON.2)：　一般用途标志位（由用户自行定义）
PD(PCON.1)：　降低功率控制位，等于1时会使CPU的功率下降
IDL(PCON.0)：　系统闲置（Idle）控制位

注：
（1）若PD和IDL同时为1，则仅有PD(Power Down)有效。
（2）系统RESET之后，PCON的值为0XXX000B。

IP 中断优先级控制寄存器各位定义

—	—	PT2	PS	PT1	PX1	PT0	PX0
D7	D6	D5	D4	D3	D2	D1	D0

- (IP.7)：　原厂保留
- (IP.6)：　原厂保留
PT2(IP.5)：　定义TIMER 2中断的优先次序
PS(IP.4)：　定义串行通信中断的优先次序
PT1(IP.3)：　定义TIMER 1中断的优先次序
PX1(IP.2)：　定义INT 1外部中断的优先次序
PT0(IP.1)：　定义TIMER 0中断的优先次序
PX0(IP.0)：　定义INT 0外部中断的优先次序

注：
（1）优先级仅分为2种，设成1时代表有较高优先级，设成0时则属低优先级。
（2）若数个中断信号同时发生时，8051将先处理高优先级的中断，然后才是处理低优先级的中断。

IE 中断使能寄存器各位定义

EA	—	ET2	ES	ET1	EX1	ET0	EX0
D7	D6	D5	D4	D3	D2	D1	D0

EA(IE.7)：　CPU是否接受中断完全由此位控制，若EA=0，则CPU不接受任何中断；若EA=1，则由以下各位来决定是否接受该中断请求：
- (IE.6)：　原厂保留
ET2(IE.5)：　TIMER 2的中断控制
ES(IE.4)：　串行通信的中断控制
ET1(IE.3)：　TIMER 1的中断控制
EX1(IE.2)：　外部硬件中断（INT 1）的控制
ET0(IE.1)：　TIMER0的中断控制
EX0(IE.0)：　外部硬件中断（INT 0）的控制

注：若bit=1代表允许该中断，bit=0则为禁止该中断，即使真有中断信号，但CPU仍不会处理。

附录 H　如何上网买电子元件

初学者在学习 8051 时最常遇到的问题，除了对程序熟悉度不足外，还有不知道该怎么买元件的问题!如果住家附近就有电子产品商店的话，走一趟电子产品商店是最方便不过了；如果电子产品商店里没有我们想要购买的元件，或是住家附近根本就没有电子产品商店，那我们应该怎么办？

一般来说，电子产品商店会有代购电子元件的服务，只要能提供详细的元件数据，如型号、规格、用途、制造厂家等基本数据，电子产品商店就可以帮我们订购需要的电子元件。不过利用这种方式采购的元件，除了价格会贵一些，更麻烦的是订购时有最小量订购的限制，也就是一次就要订购 100 个或 1000 个以上，可是我们的需求只有几个，这么多的数量绝对是没办法接受的。而上网购买元件，就变成另一个不错的购买渠道。接下来我们所要介绍的是上网购买元件的一些技巧。

提供元件采购的相关网站：

1. 拍卖网站

其实在不少拍卖网站上，有电子产品商店或个人工作室上网拍卖电子元件，只要进入拍卖网站搜索想要的电子元件名称，可以找到一些信息，如果价格上可以接受，就可以直接跟卖家联络。不过这样的采购方式，风险比较大，且购买的元件往往比较没有保障，除非我们对电子元件比较熟悉，懂得怎样去判断好坏，不然就很容易买到二手元件，或是制造日期已经是很久很久以前的电子元件。

2. 专业电子元件网

除了拍卖网站以外，还有几个专卖电子元件的公司网站，在这里提供给大家参考：一个是 RS 网站（WWW.RSTAIWAN.COM），另一个是 DIGIKEY 网站（TW.DIGIKEY.COM）。在这些网站里，我们可以直接将所需要的电子元件名称键入搜索栏，让网站上的搜索引擎直接帮我们找该元件有没有库存；也可以点选该组件的类型，顺便看看一些相关的电子元件。我们以 RS 的网站为例，实际来介绍一下要如何上网购买电子元件。

图 H-1　RS 网站的首页，在左上角有搜索栏，我们可以键入我们所要寻找的电子元件名称，以本图为例，我们要找的电子元件是 AT89C2051

上网购买元件的方法：

1. 寻找需要的电子元件

进入 RS 网站后，先在左上方的搜索栏里填入我们想要购买的元件名称，并按下搜索键，RS 网站就会帮我们找到跟该元件名称相关的信息，最重要的就是元件的包装格式、库存数量、与产品价格。

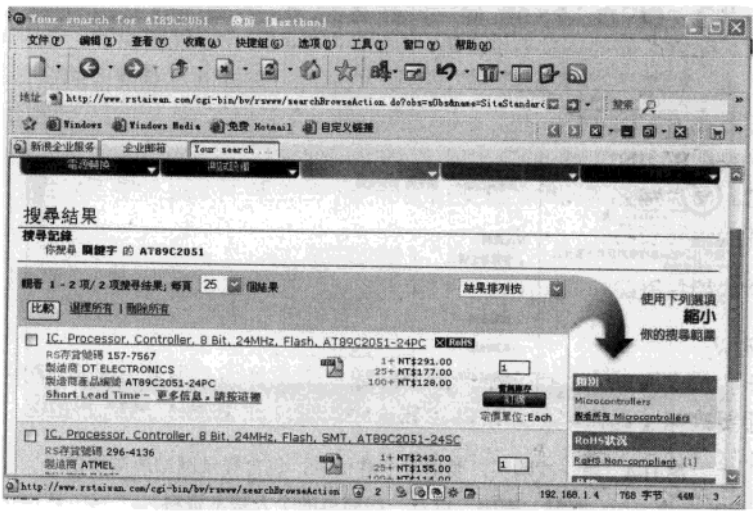
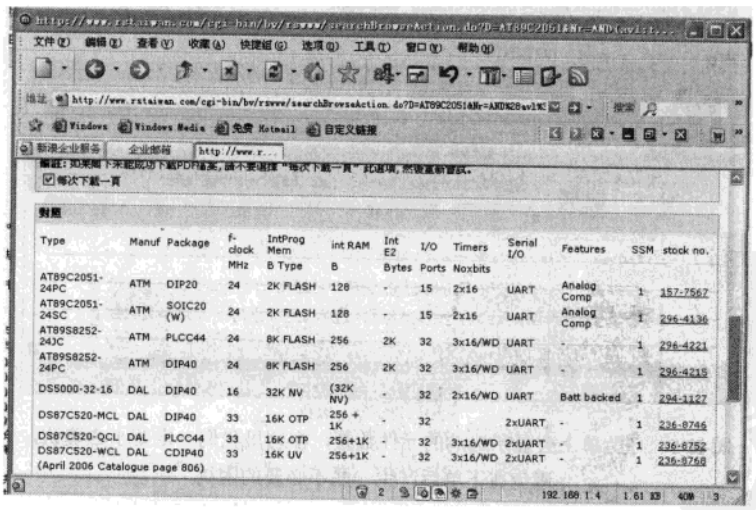


图 H-2　开始搜索后，网站里会将与 AT89C2051 有关的信息都列出来，我们可以一一点进去看，确认是不是我们所要购买的元件



Type	Manuf	Package	f-clock MHz	IntProg Mem B Type	int RAM B	Int E2 Bytes	I/O Ports	Timers Noxbits	Serial I/O	Features	SSM	stock no.
AT89C2051-24PC	ATM	DIP20	24	2K FLASH	128	-	15	2x16	UART	Analog Comp	1	157-7567
AT89C2051-24SC	ATM	SOIC20 (W)	24	2K FLASH	128	-	15	2x16	UART	Analog Comp	1	296-4136
AT89S8252-24JC	ATM	PLCC44	24	8K FLASH	256	2K	32	3x16/WD	UART	-	1	296-4221
AT89S8252-24PC	ATM	DIP40	24	8K FLASH	256	2K	32	3x16/WD	UART	-	1	296-4215
DS5000-32-16	DAL	DIP40	16	32K NV	(32K NV)	-	32	2x16/WD	UART	Batt backed	1	294-1127
DS87C520-MCL	DAL	DIP40	33	16K OTP	256 + 1K	-	32		2xUART	-	1	236-8746
DS87C520-QCL	DAL	PLCC44	33	16K OTP	256+1K	-	32	3x16/WD	2xUART	-	1	236-8752
DS87C520-WCL	DAL	CDIP40	33	16K UV	256+1K	-	32	3x16/WD	2xUART	-	1	236-8768

(April 2006 Catalogue page 806)

图 H-3　这是库存元件的资料，里面包含了该元件的名称、制造商、包装与规格、价格等主要信息

2. 确认相关的采购信息

找到我们想要的元件后，如果是第一次购买，网站会要求我们填写一些必要的个人信息，待数据填完后，就会进入采购单部分；如果我们已经填写过个人数据，只要直接输入账号密码即可。完成上面的步骤后，要确认我们所采购的元件数量与价格是否吻合，并指定付款条件及收件人的名称、地址、联系电话。

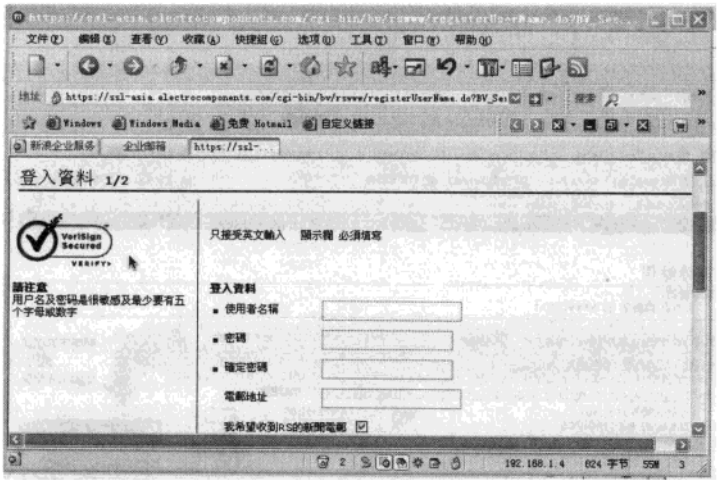


图 H-4 第一次购买，要填写一些信息，包含登录网站所需要的账号、密码，以及一些简单的个人基本资料

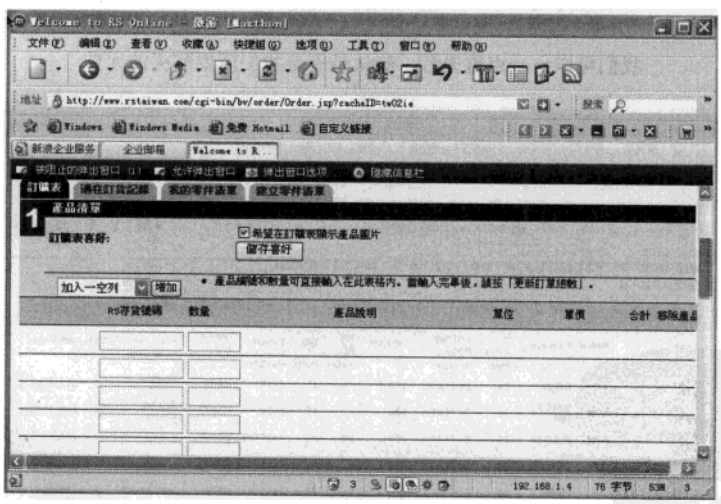


图 H-5 订购单上会显示选购的元件数量，在这里我们一定要小心确认，避免在下单后产生一些不必要的困扰

3. 货比三家不吃亏

如果您所选购的电子元件是独家代理的，那可能就没有比价的空间，但大多数常用的电子元件并不属于此类，因此，多比较几家不同公司的元件价格，您可以购买到物美价廉的电

子元件。

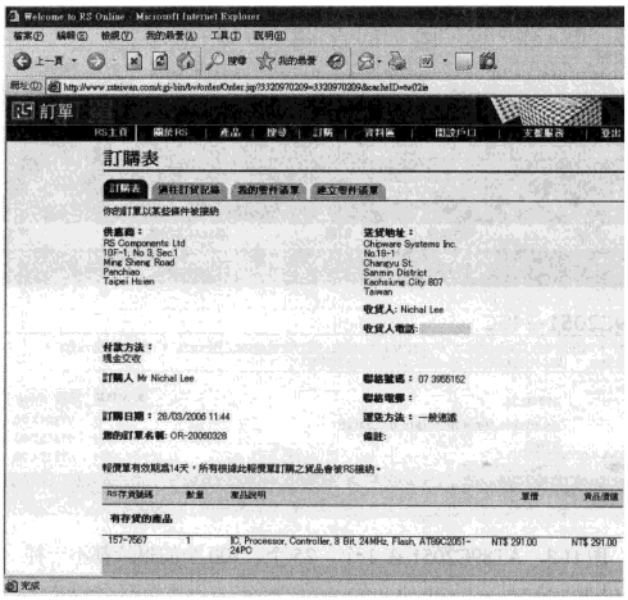


图 H-6　所有采购信息都确认后，网站的系统还会出现一张整理好的报价（订购）单，订单的有效日期是 14 天，14 天内如果你都没有将订单寄出，该订单就会被取消了

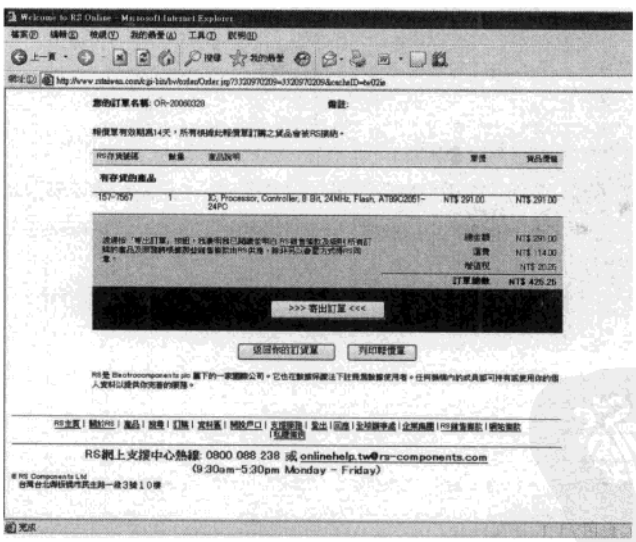


图 H-7　当你再次确认所有的数据都正确时，点选寄出订单，你的订单就被 RS 接受了

4. 数量决定售价

相信你一定会发现，如果购买的数量较多，报价也会跟着有所不同，而各个元件分担的运费也会比较低，尽量将想要购买的元件一次买齐，或和朋友一起合购，这样会比较划算。

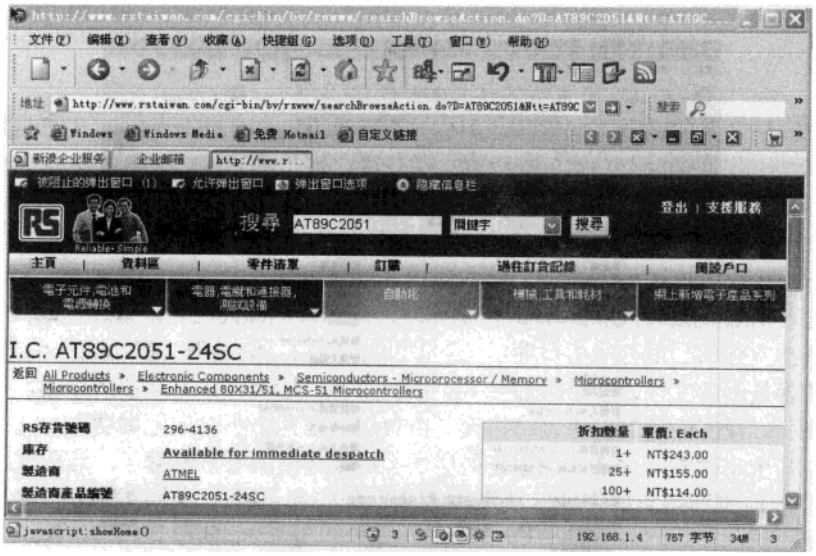


图 H-8　AT89C2051 在 1 个，25 个，100 个的报价都不一样

经过一连串的介绍，您是否对购买元件的方式有更进一步的了解了呢?

附录 I　一张照片一个故事

在 8051 的开发领域中，一定有许多小故事可以讲的。有些是苦的，有些是甜的，不过也有些故事是甜与苦交错存在的。曾经有一次在客户端修改程序，连续改了三个小时仍旧无法让机器工作，改到最后，把程序改到开机后只有一个 LED 灯点亮，还是没办法工作。这时问题已经很明显了，不是待烧录的 IC 损坏，要不然就是烧录器坏了。

又经过半个小时问题仍旧存在，IC 换新的且烧录器经过验证也没有问题。所有在场人的脸色都铁青，最后突然有人看到 NB 在编译时，好像有一些问题？建议打“DIR”看一下编译后的二进制文件的长度与生成日期，一看就知道问题了，烧录的二进制文件根本没有生成，所以烧录的都是旧的文件，顺着屏幕往下看，不得了！显示 C 盘是 0 字节可用，整个硬盘的空间已经满了，没有注意到编译时的错误，白白做了好几个小时的无用功。

经过了半个小时将程序还原到原先的样子，再除错后机器果然动了，这种椎心的感受只有写程序的固件工程师能够理解。接下来有六个我们亲身经历的小故事和照片，这些都是 8051 单片机的应用，但它们都有一个共同点：充分利用 8051 的串行通信功能。下面就是故事的开始，请您分享。

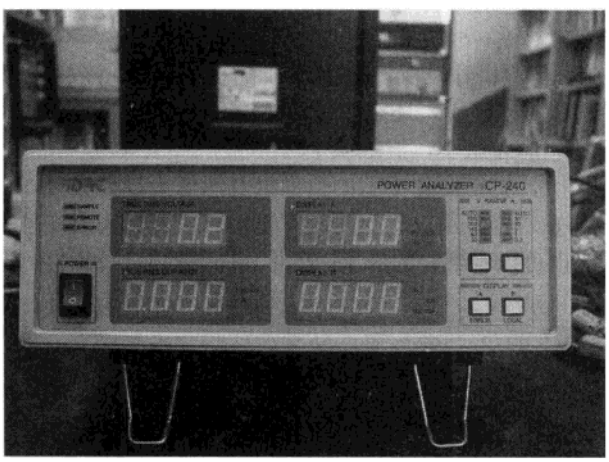


图 I-1　AC 功率计

AC 功率计

CP-240 是一台 AC 功率分析仪，它是由擎宏电子（IDRC）所研发生产的精密测量设备。当我们在进行电器功耗的效率分析时，CP-240 可以帮我们很大的忙，因为它可以观察输入电压（ACV）、输入电流（ACA）消耗功率（Power）及功率因数（Power Factor），有了这样的设备，就可以清楚地知道电子产品的用电效率，在设计时作为功率因数修正的参考。CP240 是 8051 单片机的深度应用，其背后有个通信端口，没错就是串行端口的应用实例。

由于近年来对能源使用效率的要求，工程师在设计电子产品时，已经不能再忽略用电效率这个课题了！

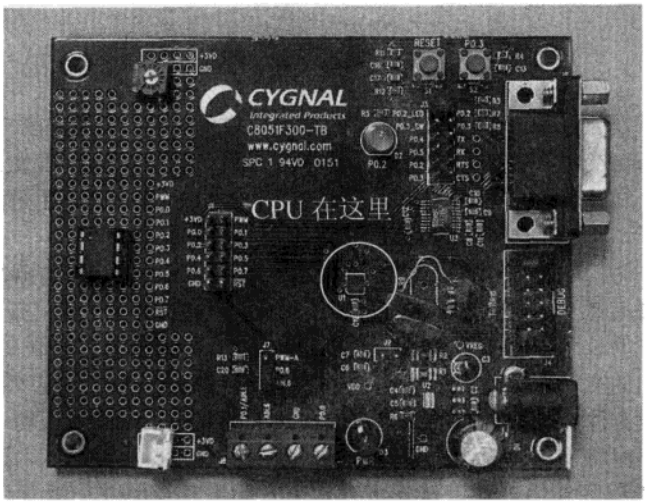


图 I-2　微型化 8051 开发板

微型化 8051 开发板

这是微型化的 8051 单片机开发板，使用 8051 的程序代码，该芯片的大小仅 3mm×3mm，内置振荡器，有八个 I/O 可以使用，还有 8 位的 ADC、8 位或 16 位的 PWM 输出、I2C 的串行传送接口等等，可谓“麻雀虽小，五脏俱全”。

在实际的医疗应用上，它搭配超小型的摄像机及影像无线传输技术，制成透明胶囊的形状，可作为肠胃道观测发送器，用以取代传统胃镜，有效降低病人在使用胃镜时的不适应；在小型风扇或灯控里，也可以看到它的影子，然而其控制的原理仍是从 8051 的基本结构延伸出来的。

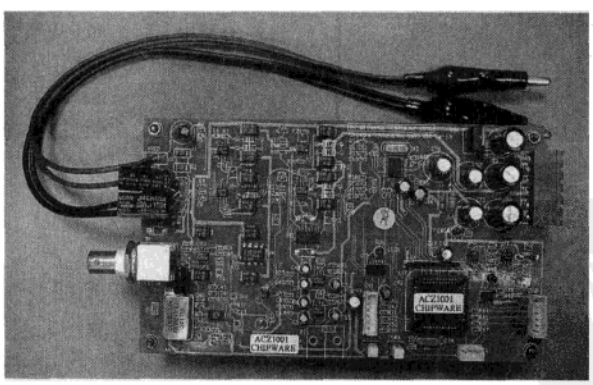


图 I-3　电池内阻测量设备

电池内阻测量设备

图 I-3 是一台旗威科技公司开发的精密电池内阻测量设备，它的核心也是 8051，通过外

加的模拟电路，可以测量到很小很小的电池内阻值，测量的范围是 0～3.0000Ω。程序中有强大的运算公式及自我校正功能，在生产时可以快速校正到容许的误差范围内，这是研发工程师在设计产品时最大的挑战。

它也有串行的通信端口，可以很容易与 PC 或其他设备连接，进行数据的交换或设定值的存入与更新。

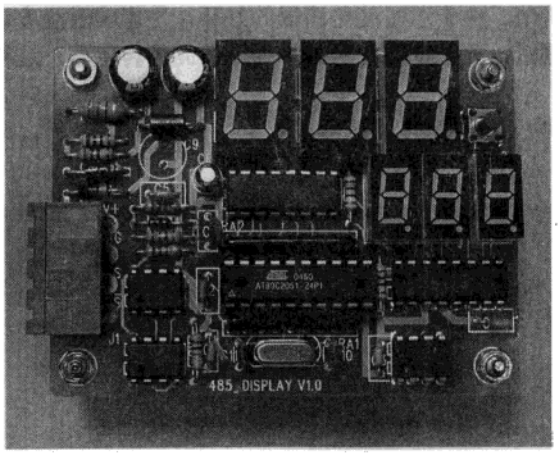


图 I-4　温度显示与设定器

温度显示与设定器

七段显示模块在大多数的控制场合里是相当实用的。它除了提供有效的数值信息，还有报警提示与 RS485 串行通信的功能。

通过 RS485 的连接，可以将分散各地的所有信息全部收集到对应的显示模块上，以便集中管理。而这个模块的核心，仅需一个 AT89C2051 便可完成。程序是用汇编语言写成，不但精简且执行效率高，这都有赖于 8051 例程的灵活应用。串行通信的服务例程如果改用 C 语言来写，除了程序空间会加大外，执行效率也会比汇编语言差一些。

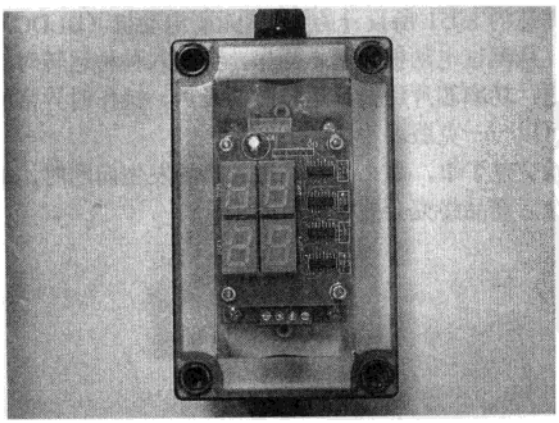


图 I-5　温湿度控制器

温湿度控制器

温湿度控制器是很普遍的，它是在环境控制系统上的必要工具之一，因为温湿度的参数是环境控制系统里最基本的记录参数。照片中的 TH2040 是旗威科技公司开发的精密数字式温湿度计，除了准确的测量值以外，高性能的校正方式与 RS485 的通信结构，让温湿度的记录更简单也更具参考价值。由于温湿度传感器的好坏，是影响温湿度误差的最主要因素，因此在选用温湿度传感器时，要选择特性稳定且重现性高的传感器，这样才能有效地降低误差值。

对于一台不到 250 元的温湿度显示器，其值只能“仅供参考”，无法记录也无法把值传到其他装置上。

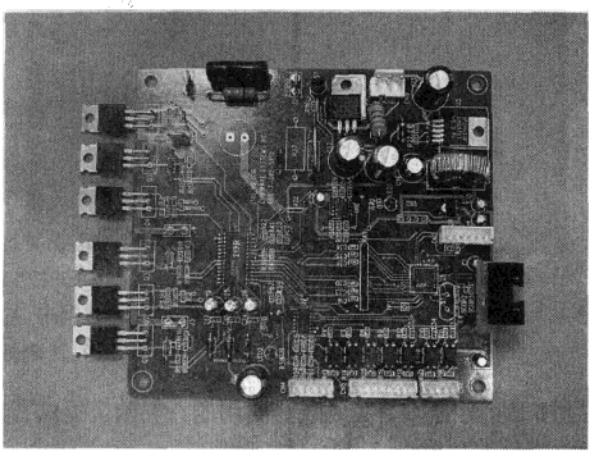


图 I-6　无刷电动机控制器

无刷电动机控制器

电动机控制一直是很吸引人的领域，它将电子和机械两大领域紧密地结合在一起。图 I-6 是旗威科技公司利用高速的 8051 所设计的 1kW 无刷电动机（BLDC）控制器，在这方面旗威科技公司投入很深，从测试电动机的基本操作，到加入种种运转时的保护措施。当然其中经历很多的挫折和难题：功率芯片烧毁、运转时的噪声、操作时异常的电磁干扰等，只要理清问题的根本原因就可以一一克服。

若只想可以工作就交差了事，那么当产品进入市场发生问题时，很可能会完全没有除错的方向和处理头绪，最后商品终究会被市场所淘汰。

[General Information]
书名=8051单片机彻底研究　入门篇
作者=林伸茂编著
页数=274
出版社=中国电力出版社
出版日期=2007
SS号=12660528
DX号=000006205446
URL=http://book.szdnet.org.cn/bookDetail.jsp?dxNumber=000006205446&d=2417E2C8F2E77CE667774969DFF22105