目录

导 言

本书的对象主要是希望从事单片机软件开发的人员，当然不是只限于这些人员才可以看，只要你有兴趣，哪怕你想成为一名炒菜的大师傅或者已经是大师傅，我一样欢迎你来阅读本书。

当我还是后生仔的时候，虽然也常做一些某年某月自己能呼风唤雨或腰缠万贯的白日梦，在内心还是想能去传道授业，哪怕是当个大和尚，面对虚心而来请教的人众，该是何等惬意的事情。然而天生不善言辞，更是拙于笔墨，担心误了他人前程而作罢。

可谓江山易改、本性难移，虽然没去当成大和尚，可时不时还想起这个愿望。一天看到台湾侯捷（侯俊杰）写的《深入浅出 MFC》，在我看来这书写得那叫一个好，如果我也能写出这样的书简直是太伟大了。

实际上我是写不出侯捷前辈那种水平书的，那除了要有非常好专业态度和技能外，还要有一双可以偶得之的妙手。反过来一想，不一定非要写出专业水准，只要自己写得开心，权当对自己技术工作的一个总结也没什么不妥，于是开始了本书的写作。

开始写之后才体验到写书的痛苦，一度甚至怀疑洋洋洒洒、下笔如有神这些词汇是不是真的存在。从二〇〇九年六月份开始，基本上是蜗牛一样的速度，写到中间几次都差点放弃，还好有想做大和尚的信念支持，二〇一〇年终于完成初稿，谢天谢地。

为了便于大家理解，书中多采用口语方式讲述单片机的一些相关知识、经验和技巧，所以没什么文采可言。书中内容按先基础后技巧的顺序排列，看完你会发现所写内容你可以用到任意一款单片机上，但又不能直接应用。通过本书我只是想告诉你一些思维方法，并不是让你在某款单片机上照葫芦画瓢。

书中部分 C 语言和汇编的例子，我是用 ADS 针对 ARM 的编译结果作为示例，也有部分采用伪代码进行意思说明。书中内容都是我个人理解，没有进行严谨的考证，难免有一些偏颇之处，欢迎大家批评指正。

文字工作对于我来说难度确实大，何况写作的同时还要完成本职工作，只能是利用工作间隙去写，所以希望大家能体谅我码字的辛苦，如需转载使用千万别忘记标明出处。 ：p

邮箱：daishangju@163.com MSN：sj_dai@hotmail.com

书中少量内容有参考部分网络资料，在此向这些资料作者表示感谢。

第一章　单片机基础

如果你已经有了一定的单片机基础，你只需粗略浏览本章甚至可以直接跳过本章；如果你具备一些基本的电子电路知识，但没有真正接触过单片机，请将本章看完后找一套单片机仿真器体验一下真正的单片机，再回来将本章认真看一遍；如果你之前接触过单片机，但还在似懂非懂的阶段，请认真仔细阅读本章，务必领会我在本章中想表达的意思，点滴都不要放过；如果你已经对单片机很熟悉，请移步到后面你觉得合适的章节，当然也欢迎你仔细阅读本章，但不要吝啬把你发现的错误告诉我。

1.1. 什么是单片机

单片机一词的来历不明，在我看来是单片微控制器缩写的解释可能性最大。在日常生活中我们常听到微电脑控制这样的说法，这里的微电脑控制就是单片机控制。现在用单片机进行控制的电子产品已经深入到我们日常生活的各个角落，手机、电视、冰箱、汽车、飞机等等无一不用单片机进行控制，几乎只要有电的地方就有它的存在。

单片机实际上就是一颗可以让用户设计控制方法和流程的集成电路芯片，在实际生活中，不同产品、不同厂家会有着千奇百怪的控制想法，单片机的优点就是芯片厂家提供了一个通用平台，在这个平台上各个用户只要依照厂家约定规则操作就可以实现自己的控制需求。

单片机种类繁多，功能自然也各不相同。象三星公司的 S3C6410，主频可以高达 800MHz，424 条管脚，功能强大性能几近电脑；而台湾义隆公司的 EM78P152 则只有 8 条管脚，可供用户选择控制的输入输出脚也就 5 条，只能实现一些简单的逻辑控制功能；MICROCHIP 公司的 PIC10F20X 更小，6 条管脚，内置 4M 晶振，上电就可以运行。可见不同型号的单片机差异可以非常之大，价格也是一样，便宜的不用一元人民币就可以买到，贵的则要上百元人民币甚至更多。

几乎所有知名大型电子公司都有生产单片机，如 SAMSUNG/FREESCALE/TI/NXP/RENESAS/TOSHIBA 等，这些公司生产的单片机大都是通用类，没有专门针对某类具体产品设计特定功能，这类单片机特点是性能相对稳定、价格比较高。产品的需求是层出不穷，一些特殊的需求会催生出对应有特殊功能的单片机，比如电视游戏机就需要直接支持音频、视频信号输出，于是一些规模相对要小一些的厂家如 SUNPLUS/WINBOND 等就推出专门针对电视游戏机市场的单片机，就是电脑键盘、鼠标这些非常简单的产品，都有诸如 HOLTEK 这样的厂家来设计专用单片机。

说了这么多那单片机到底是个什么样子？别着急，既然学校大多将 51 系列的单片机如何应用做为单片机的课程内容，这里就展示两张 AT89C51 单片机的图给大家看看。让它工作起来也很简单，写好程序，将程序按规定的方法写到单片机的指定位置，再将这个单片机接上晶振，加上合适的工

作电压，它就会乖乖的工作起来。（如何工作可参见章节 **1.2. 单片机是如何工作的**）

The AT89C51 provides the following standard features: 4 Kbytes of Flash, 128 bytes of RAM, 32 I/O lines, two 16-bit timer/counters, a five vector two-level interrupt architecture, a full duplex serial port, on-chip oscillator and clock circuitry. In addition, the AT89C51 is

(continued)

Pin Configurations

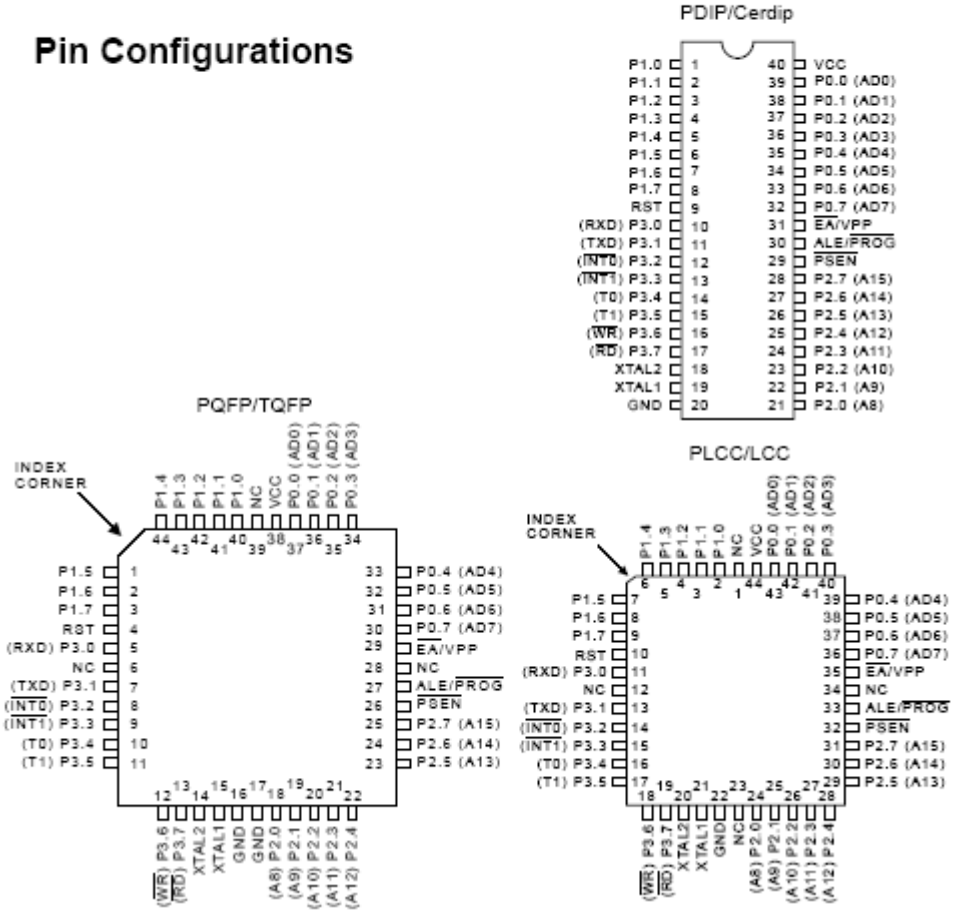


图 1.1.-1 AT89C51 封装图

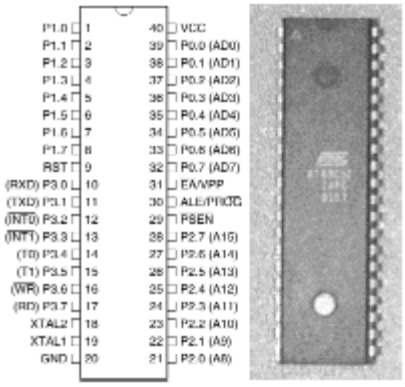


图 1.1.-2 AT89C51 DIP 封装和实物对照图

●注：有的时候为了降低成本可以不要上面的标准封装，要求厂家提供 DICE（也叫 CHIP），拿到 DICE 后自己再将管脚邦定（BONDING）出来

1.2. 单片机是如何工作的

单片机是如何工作的？这还真是一个让人头大的问题，武侠小说中一些权贵豪杰为了保护自己和家人的人身安全，会在自己住的地方让能工巧匠设计许多机关，什么吊网、毒箭、暗坑都统统上场，一环套一环、一招接一招，只要一触发机关，就算来者是神功盖世也十有八九要玩完。这种机关工作方式是触发，上一个机关动作完就会触发下一个机关，直到所有机关都触发完。**触发**这个概念对单片机来说非常重要，单片机是这样的，将需要完成的操作任务预先一步步设定好（这个预先设定好的操作任务也就是程序），然后单片机逐步被触发实现这些操作，那单片机靠什么触发呢，后面我会告诉你。

有没有搞错？单片机居然就是武侠小说中的机关？没错，单片机就和机关差不多，只不过机关是利用力学原理通过一些机械结构来实现，而单片机是靠其内部的电子器件来实现。单片机的核心是一个叫 CPU 的中央处理器，这个东西就好比人的大脑，可以实现一些逻辑运算之类的操作，比如对两个数进行加减、比较大小等，前面提到的程序这里就用得上了，程序实际上是一串数字，这串数字依照单片机制定的规则表达相关特定信息，CPU 则通过某种方式把程序里的这些数字取来然后按数字进行相应的操作。

为了弄清楚单片机是怎么工作的，这里我给大家设计一个简单的单片机，而且这个单片机还与人脑相互兼容，哈，很厉害吧！该单片机外部有两个电压可以选择输出为高或为低的管脚，这里我分别将其命名为左管脚和右管脚，为了演示这个单片机的功能，需要设计一个单片机应用系统来控制两盏灯，这两盏灯会按下面的亮灭次序工作。

a.开始灯一和灯二都是熄灭状态。

b.灯一点亮，灯二熄灭，维持该状态一秒。

c.灯一保持点亮，灯二点亮，维持该状态两秒。

d.灯一熄灭，灯二保持点亮，维持该状态三秒。

e.灯一和灯二都熄灭……

既然是我设计的单片机，那先要定出这个单片机型号，不然就和别人的设计无法区分，简单点就叫 MY_MCU_000，再要制定相关规则，也就是设计 MY_MCU_000 的机器指令。

数字（指令）	MY_MCU_000 实现功能
0	左管脚和右管脚都输出低电压
1	左管脚输出高电压，右管脚输出低电压
2	左管脚输出低电压，右管脚输出高电压
3	左管脚和右管脚都输出高电压
4	保持当前状态，什么都不做

表 1.2.-1 MY_MCU_000 指令表

设计的应用系统，当然这个系统只是简单示意，目的是让大家能明白是如何控制灯的。

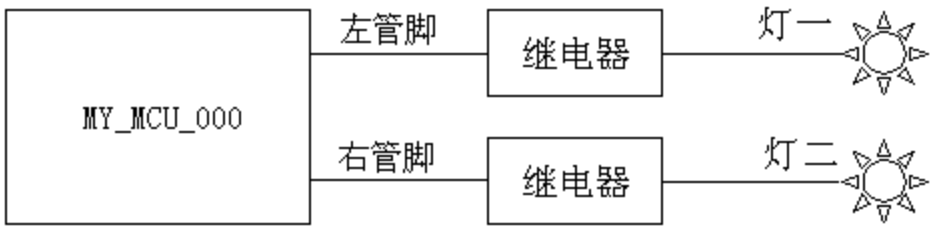


图 1.2.-2　MY_MCU_000 应用系统示意图

先不要进入“难度大”的 MY_MCU_000 编程阶段，前面提到该单片机还能与人脑相互兼容，现在通过一位魔法师给你演示如何实现。魔法师会交给你一张魔法秘籍表，另外还会给你一张写有魔法密语的纸条。

魔法密语	魔法方法
0	左手和右手都松开开关
1	左手按下开关，右手松开开关
2	左手松开开关，右手按下开关
3	左手右手都按下开关

表 1.2.-3　魔法秘籍表

现在你展开魔法密语纸条，魔法师开始指挥你施展魔法。魔法师每挥一下魔法棒，你从纸条上取出一条魔法密语，并按照魔法师先给你的魔法施法秘籍表做出相应动作。如果魔法师每一秒向你挥一次魔法棒，那么你每隔一秒就要从魔法密语纸条上取出一条魔法密语并施展魔法，现在你仔细想想，是不是“神奇”地实现了前面我们所想要实现的亮灯次序？

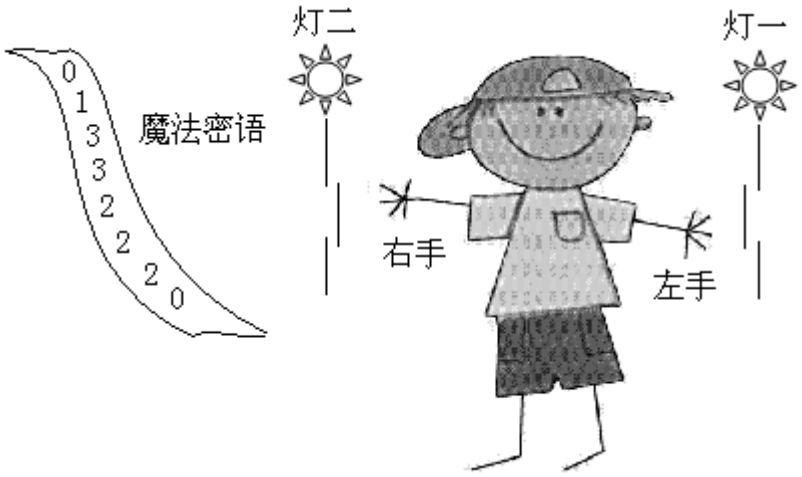


图 1.2.-4　魔法施展图

看到这里不要笑，这么简单也叫什么魔法？这确实不能叫魔法，目的是让你明白一个道理。接下来我们回到**图 1.2.-2 MY_MCU_000 应用系统示意图**，把“魔法密语”交给所设计的这个单片机应用系统，也就是存储在 MY_MCU_000 可以读取的地方，会发生什么事情呢？显然只要给所设计的单片机 MY_MCU_000 每一秒钟发一个触发信号，这个触发信号可以触发 MY_MCU_000 从“魔法密语”里面取出一条密语并执行相应操作，我们所设计的单片机系统同样会按我们期望的次序亮灭两个灯。

现在将魔法和 MY_MCU_000 做个对比：

MY_MCU_000 CPU = 你的大脑

MY_MCU_000 左管脚 = 你的左手

MY_MCU_000 右管脚 = 你的右手

MY_MCU_000 程序存储器 = 魔法纸条

MY_MCU_000 存储器里面的程序= 纸条上的魔法密语

MY_MCU_000 触发信号 = 魔法师的魔法棒

我想到这里你应该明白单片机是怎么工作的了吧？你是用眼睛看到魔法密语，那我们设计的单片机是用什么呢？是控制器里一个叫 PC 程序指针的功能模块，当单片机工作时，控制器由 PC 得到当前单片机指令机器码然后译码执行，同时 PC 会自动指向下一条会被执行的指令机器码。

真正的单片机应用系统由晶振或 RC 振荡器来提供稳定的周期触发信号，魔法密语对应的是单片机程序（机器码）。如果将程序写成魔法密语一样的数字，对程序员来说肯定不是好事情，必须逐个查表才知道相应功能，一不小心就会弄错。人类不擅长查表这种繁琐工作，但电脑擅长，并且快而准。既然这样，那就把这个查表的工作交给电脑去做就是，于是解放程序员的汇编指令和编译器诞生了。

为了便于推广我们设计的 MY_MCU_000，这里我们也设计一套简单的汇编指令和编译器。前面我们已经知道这个单片机有简单的控制左右两个脚输出高低电压的功能，很好，这里我提出几种设计进行对比，看哪种最合适。

方案一	方案二	方案三	数字	MY_MCU_000 可实现功能
左低右低	Left0Right0	L0R0	0	左管脚和右管脚都输出低电压
左高右低	Left1Right0	L1R0	1	左管脚输出高电压,右管脚输出低电压
左低右高	Left0Right1	L0R1	2	左管脚输出低电压,右管脚输出高电压
左高右高	Left1Right1	L1R1	3	左管脚和右管脚都输出高电压
空操作	DoNothing	NOP	4	保持当前状态，什么都不做

表 1.2.-5　MY_MCU_000 汇编指令对比表

方案一用的是中文，对于中国人来说非常直观，可是如果我们设计的这个单片机要占领国际市场，怎么推给外国人呢？另外编程的时候打中文好象也有点复杂，否定。

方案二用的是英语，国际化问题应该不不会再出现，是不是会觉得每一条汇编指令需要输入的字符多了一些？如果有更便捷点的方法自然更好。

方案三，几个字母就能表达出一种功能操作，看上去也不会混淆意思，还不错，那就用它。

实际上这三个方案都可以实现我们想做的事情，只是我们选了相对来说比较优化的方案三，我们设计的 MY_MCU_000 不认识方案三里面的那些指令，这样就需要做一个编译器，编译器将这些指令转换成MY_MCU_000 认识的魔法密语——机器码。

第一代设计		第二代设计	
汇编指令	机器码	汇编指令	机器码
L0R0	0	L0R0	0
L1R0	1	L1R0	1
L1R1	3	L1R1	3
L1R1	3	NOP	4
L0R1	2	L0R1	2
L0R1	2	NOP	4
L0R1	2	NOP	4
L0R0	0	L0R0	0

表 1.2.-6　亮灯程序汇编指令和机器码对比表

到这里我们设计的基本大功告成，可以让市场部门拿出去进行销售，不过接下来我们还是应该对其进行完善，使其更有市场竞争力。对于连续多个 L1R1 这样的操作，客户反馈说看着别扭，能不能改进？没问题，上图右边的第二代设计已经为消除了客户的这个烦恼，另外我们还为了客户方便推出了可以提供调试功能的调试器和可支持 C 语言编程的 C 编译器（这部分工作就留给你来设计吧）。

前面我说真正的单片机系统运行起来很简单，只需接上合适的晶振，再加上合适的电压就可以开始执行单片机的程序。是不是真的这样呢？这里给出一个真实产品的电路图，你会发现在这个单片机的外围元件非常简单，只有一个晶振、三个电容和一个电阻，就是这么简单的电路就能这个单片机跑起来。这里提一个问题，如果不给单片机提供程序，单片机系统能运行起来吗?

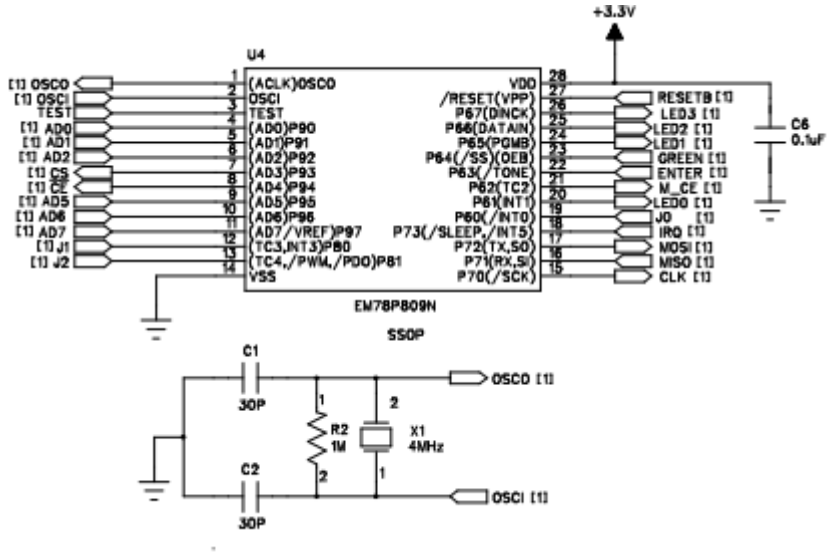


图 1.2.-7　某真实产品电路图

1.3. 单片机与电脑的区别

单片机和电脑有什么区别？从本质上讲，单片机和电脑没有什么区别，都是同一个祖宗，自七十年代开始大约有十年的时间都不分彼此，那时候还没有个人电脑这个概念，怎么分啊？随着技术和市场的发展，个人电脑概念应势而生，应用方向不一样，于是出现两条方向完全不一同的发展道路。单片机是希望所有功能都在自己内部完成，追求的是精简；而电脑则是让 CPU 专门负责数据处理功能，CPU 制定出一系列规则，由外接器件实现具体功能，追求的是标准统一下的高速性。

电脑 CPU 不能单独工作，需要由主板来连接的外部存储器（硬盘、内存等）、输入输出设备（键盘、显示器等）支持才行，单片机只要连上晶振通电就可以运行程序。你可以这样理解：主板和外设支持下的电脑 CPU 系统就是一个功能超强、速度超快、容量超大的单片机，当然价格也超贵。

个人电脑自概念诞生的那一天起，就决定了电脑 CPU 市场最终会把持到少数的厂家手中。要把电脑 CPU 做好做快需要先前的技术积累，只有性能稳定可靠才能占稳市场，占稳市场才会有资金继续研发保持产品技术领先。事实也证明了这一点，在 386/486 时代还有不少厂家在做电脑 CPU，但现在大都已经放弃这部分业务，只剩下大家熟悉的那么几家。单片机不同，一开始就没有统一标准，相对电脑 CPU 来说，技术门槛低，几个人都有可能设计出一款单片机。加上各种电子产品对单片机的需求各不相同，有需求则有市场，有市场就有存在，高性能单片机有人要，低性能单片机同样也卖得出去，于是单片机体系结构纷繁复杂，形成大公司走高端、小公司创特色的局面。

天下大势，分久必合，合久必分。电脑 CPU 自八十年代与单片机分道扬镳后那是越做越快、越做越小、越做越便宜。而单片机呢又出现两极分化的情况：低端在保证性能同时价格大幅下降，已经出现售价不到一元人民币的单片机，同时还针对特定的市场需求推出专用型号；高端的功能越做越强大，逐步向电脑发展方向并拢，这方面典型的例子就是九十年代出来的 ARM 公司，专门给另外

的公司提供类似电脑 CPU 功能的内核，另外的公司则象电脑公司一样在其内核外围将电脑公司在主板上实现的功能进行扩展，区别是电脑在主板上扩展，高端单片机在芯片内扩展。

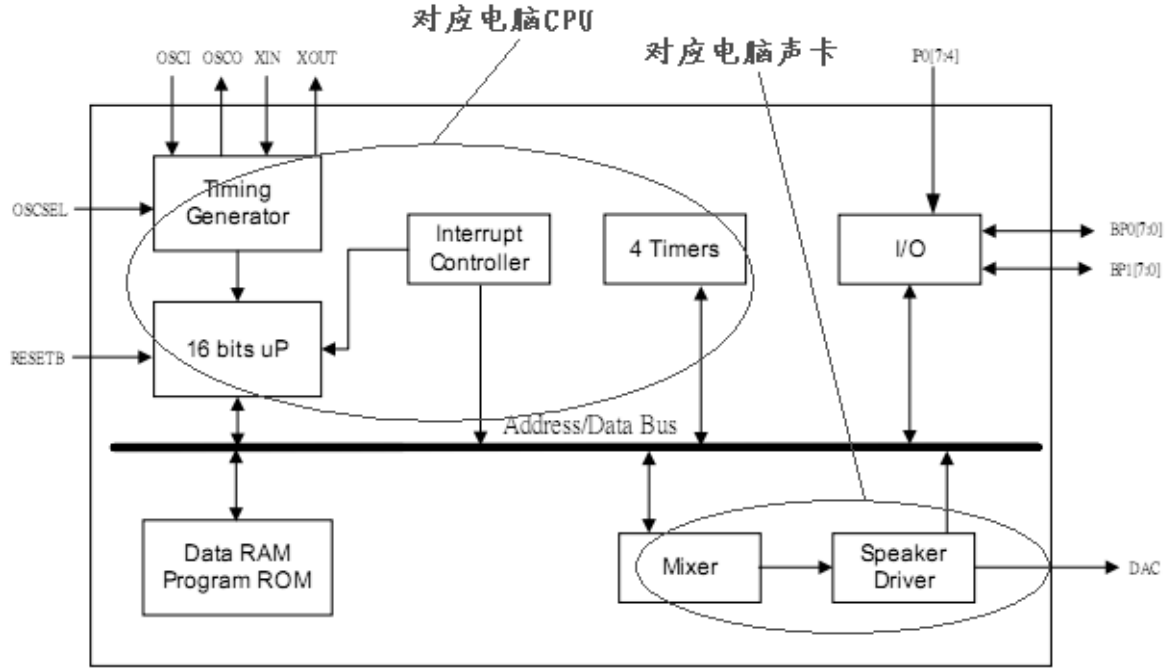


图 1.3.-1　某低端专用单片机构架图

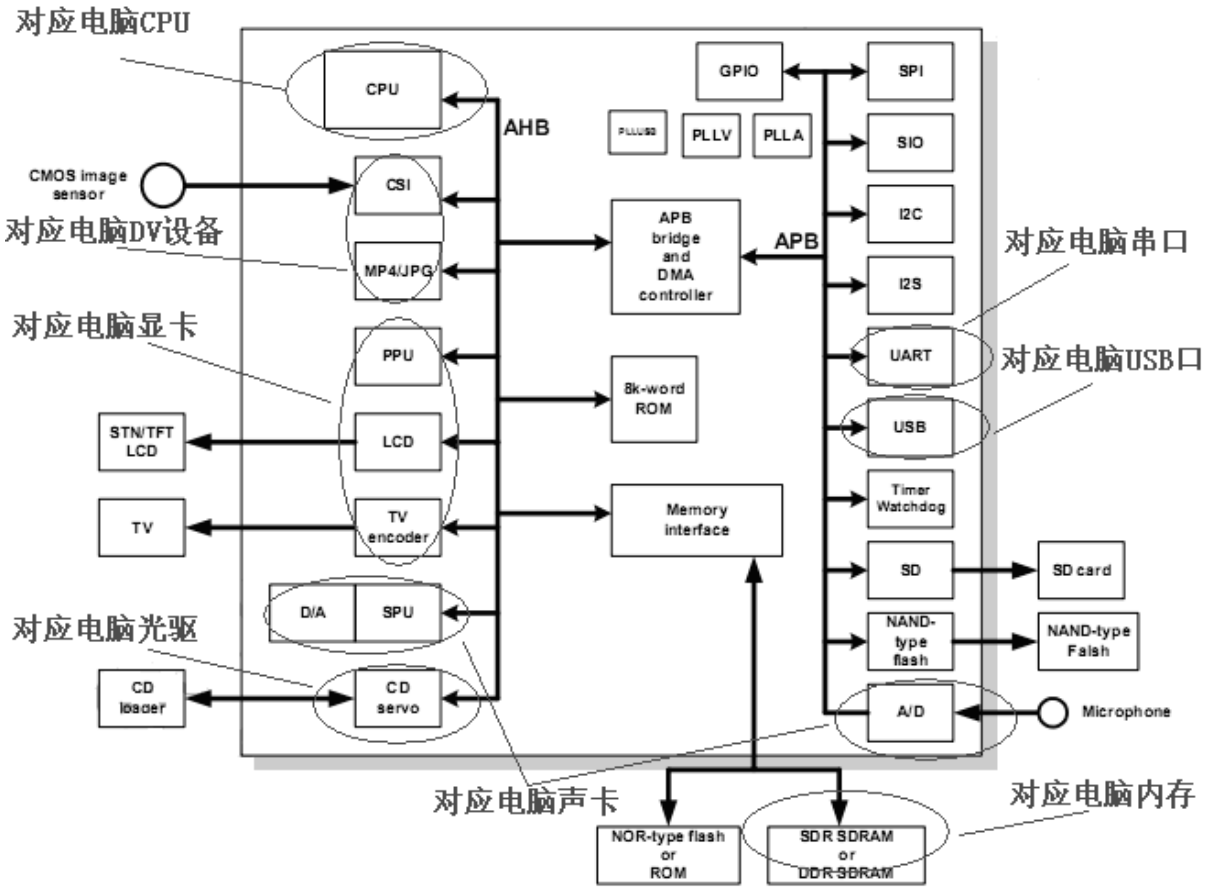


图 1.3.-2　某高端单片机构架图

图 1.3.-1 为一款主要用于发声产品的单片机内部构架图

图 1.3.-2 为一款性能一般的高端单片机（非 ARM 内核）内部构架图

前者除了发声外几乎不能实现其它的电脑日常功能，但后者几乎可以将电脑的日常功能全部实现，只是性能上还存在一定差距。仔细研究后者构架可以看出其构架也已经和电脑构架非常接近，甚至有了主板上非常重要的南北桥技术——AHB/APB（高低速总线）。

主板和外设支持下的电脑 CPU 系统就是一个功能超强、速度超快、容量超大的单片机，这就是两者的区别。电脑 CPU 引领潮流和技术，单片机在特色方面加强体现，两者一度界限明晰，现在又出现相互融合的趋势。会不会有一天单片机又会和电脑 CPU 完全融合成同一种产品呢？我的观点是不会。电脑在人们的眼里，就是需要那么大的个头，哪怕有一天技术已经发展到可以将所有芯片功能融合到一个火柴盒那么大，但人们已经习惯大的显示器和操控顺畅的键盘、鼠标等设备，这个体积难以减小。单片机许多应用场合并不需要这样的显示和输入设备，市场对单片机的要求往往是简单实用，就如同上面**图 1.3.-1** 所示的低端单片机，它只要在发声方面能满足用户的实际需求，就可以稳稳占有带发声功能的简单电子产品市场。

1.4. 晶振

大部分单片机都需要晶振才能工作起来，晶振就象交响乐团的指挥家一样控制单片机的工作节奏，指挥家的指挥棒没起，交响乐团是不会开始演奏的，同样晶振没开始向单片机提供节奏信号，单片机也不会工作起来。在 **1.2. 单片机是如何工作**一章中说**触发**是单片机的一个重要概念，晶振就是单片机所有工作触发时序的信号源，单片机通过它所提供周期稳定的触发信号去触发程序相应操作，或者是去检查外部有没有什么触发信号输入。

不是所有的单片机都用晶振来做触发信号源，还有一些场合可以用 RC 振荡器，甚至是单片机自己内部有 RC 振荡电路，外部接一个电阻来调节振荡频率。既然 RC 振荡器也可以用，为什么还要用晶振呢？原因很简单，RC 振荡器所产生的频率一致性和稳定性都不好，会因为电阻电容值的误差出现比较大的偏差，电压的高低也会产生一定的影响，晶振虽然价格要高一些，但一致性和稳定性比 RC 振荡器要好许多。

实际应用应根据产品特性选择晶振或 RC 振荡器。如果产品对控制性能的时间精度要求并不严格，比如是一个通过按键发光发声的简单儿童玩具，就可以用 RC 振荡器来降低成本；但如果一个产品需要保持显示日期时间，用 RC 振荡器显然不能满足要求，一天下来有可能会差上几分钟，所以必须用晶振，这个例子涉及到**累计误差**的概念，后面有一章节会对**累计误差**专门进行阐述。

既然晶振只是起到触发信号源的作用，那是不是可以用一个周期稳定的信号源来替换晶振或 RC 震荡器呢？适当条件下确实可以，只是这样一个信号源的实现会比用晶振的成本都要高，在我之前的产品开发经历中，就有做过多个单片机只用一个晶振的产品，但不建议这么做，因为晶振电路板

走线都有一定要求的，控制不好容易导致晶振不启振。

是不是一个单片机想跑多快就需要晶振提供同样快的周期触发信号呢？无论是设计单片机芯片还是生产晶振的厂家都不希望这么做。我们知道，频率越快，则越难控制，也越容易被干扰。晶振做为一个外接器件，自然不希望自己被要求提供非常高的频率。频率越高，对产品电路板布线的限制就越多，产品开发、生产部门使用起来也就越麻烦。设计单片机芯片的厂家为我们解决了这个烦扰，他们将一个叫 PLL（锁相环）的技术放到了单片机芯片内，通过这个技术将晶振的频率在单片机内部倍频，这样就可以让单片机得到比晶振高几倍甚至许多倍的工作频率。除此外 PLL 还有一个优点，单片机可以在工作中动态改变工作频率，后面会有章节介绍如何利用这一特性来提升产品性能。

1.5. 系统时钟和周期

说完晶振接下来介绍另外几个和晶振紧密相关的特性：**系统时钟**、**机器周期**、**指令周期**。

●注：后面所有涉及时钟、周期的概念均以晶振为例，RC 震荡器不做重复介绍

这里先把我个人理解的相互关系告诉大家：

指令周期≥机器周期≥系统时钟周期，三者没有绝对的区分准则

系统时钟

单片机内部的所有工作，都是基于由晶振产生的同一个触发信号源，由这个信号来同步协调工作步骤，我们把这个信号称为系统时钟。通过章节 1.4. **晶振**我们知道系统时钟一般由晶振产生，但在单片机内部系统时钟不是一定等于晶振频率，有可能小于晶振频率，也有可能大于晶振频率，具体是多少由单片机内部结构决定，正常情况和晶振频率会存在一个整数倍关系。

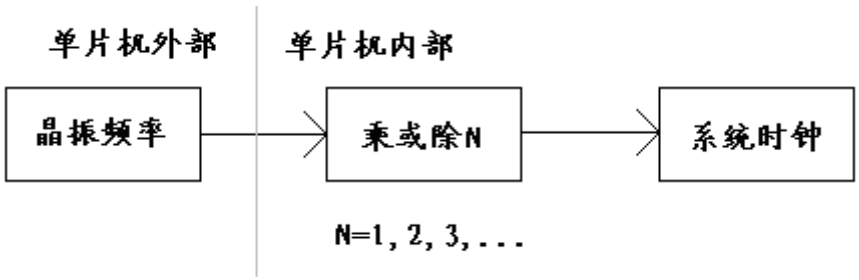


图 1.5.-1　晶振和系统时钟关系图

系统时钟是整个单片机工作节奏的基准，它每振荡一次，单片机就被触发执行一次操作。从**图 1.5.-1 晶振和系统时钟关系图**中可以看出在单片机内部有可能会乘或除某个数，不是所有的单片机都会支持中间的乘或除，此时 N=1。如果是乘，对应的是 PLL（锁相环）功能，这样就可以通过频率较低的晶振得到更快的系统时钟，让单片机跑得更快。如果是除，则是实现方法非常简单的分频

技术，这样可以让单片机在同样频率的晶振下得到不同速度的系统时钟，快的系统时钟用于正常工作状态，慢的系统时钟则用于某些特殊工作模式。

比如现在有一个产品，能自动接收外面设备发送过来的数据，外面的设备发送数据间隔时间比较长，可能一天就发送一两次，我们就可以将这个产品选用支持分频功能的单片机，每当收完一组数据后就将分频系数调大，这样单片机就进入慢速工作状态，功耗会明显降低，当检测到外部设备请求发数据操作后将分频系数调回正常，高效接收处理数据。

机器周期

还得通过章节 **1.2. 单片机是如何工作的**里面举的魔法例子来理解机器周期，当魔法师挥一下魔法棒时，表演者并不能一下就将整个魔法直接完成，需要经历“从纸条上看到魔法数字”→“从魔法秘籍表中查到数字对应动作”→“做出相应动作”这三个阶段，先前是让人一次顺序完成这三个阶段。观众仔细观看了该魔法表演后发现一个问题，表演者控制的灯亮灭与魔法师挥动魔法棒之间有延时，而且不同亮灭延时不一样。

原来每次魔法师挥魔法棒时，表演者得到数字后都要从魔法秘籍表中从头到尾查找对应的动作，因为舞台灯光不好、魔法秘籍表上字迹潦草等原因，表演者找出对应动作需要一定时间，在魔法秘籍表中越靠后的动作查找的时间就越长，这样从魔法师挥动魔法棒到动作效果做出来就会出现长短不一的延时。

为了更好的展现魔术效果，魔法师对原来的规则做了个小小的调整，将原来挥棒一秒一次改为一秒三次，第一次挥表演者去看纸条上的数字，第二次挥表演者再从魔法秘籍表中找出对应动作，第三次挥表演者则做出相应动作，这样观众就应该很难察觉延时不同的存在了吧？

稳妥起见，魔法师决定先内部演练一下。哈！新问题又出现了，现在魔法师一秒会挥三次魔法棒，表演者跟不上这个速度，等他看完纸条或者查完魔法秘籍表再来看魔法师的时候魔法师已经挥了下一次，根本无法表演。

经过测试发现，表演者看纸条最长时间不超过 0.4 秒，查魔法秘籍表则不超过 0.5 秒，做出相应动作很快只要 0.1 秒。这样魔法师再次对规则做了修改，魔法棒的挥棒间隔还是保持一秒不变，只是将原来灯亮灭的时间增加到原来的三倍，表演大获成功。

单片机也就是这个原因而引入机器周期，同样以在 **1.2. 单片机是如何工作的**中我们自己设计的单片机 **MY_MCU_000** 为例来进行说明。为了和魔法师表演兼容，单片机查表采用最简单的方法，从头到尾依次对比，直到找与之到相同的代码。假定每比对一次需要 0.1 秒，这样每次查表的耗时从 0.1 到 0.5 秒不等，会导致单片机也有有延时不同的问题。

代码	MY_MCU_000 可实现功能	查表所需时间
0	左管脚和右管脚都输出低电压	0.1 秒
1	左管脚输出高电压，右管脚输出低电压	0.2 秒
2	左管脚输出低电压，右管脚输出高电压	0.3 秒
3	左管脚和右管脚都输出高电压	0.4 秒

4	保持当前状态，什么都不做	0.5 秒

图 1.5.-2 MY_MCU_000 查表耗时表

我们对单片机运行方式也做同样修改，原来一个时钟触发完成的操作分解成三个时钟触发来完成，第一个时钟信号触发取代码，第二个时钟信号触发查表，第三个时钟信号触发输出。现在再用间隔一秒的时钟信号去触发修改后的单片机，和改进的魔法表演一样延时不同的问题消失。这种工作方式下的**MY_MCU_000** 机器周期等于三个系统时钟周期。

如果你是一个善于提问题的人这里可以提出一个问题来的，上面耗时最长的基本操作是查表要 0.5 秒，现在我们给的系统时钟是 1 秒，那最快可以到多少呢？测试发现，最快可以达到 0.5 秒，不过为了留一定的余量，对外还是只说最快为 1 秒。单片机也是如此，如果你不相信，可以随便拿出一个单片机验证一下，实际允许的最快频率都会比厂家说的要快一点，不过你在开发产品的时候可别这么做，不然出了问题厂家是不会承担责任的。

对单片机有一定基础的朋友可能会有疑问，这个例子的三步骤是不是对应单片机理论中的取指、译码、执行操作？这种理解是对的。

对于单片机芯片的设计，并没有一个固定的规范要求设计人员必须遵守，这样就会导致系统时钟、机器周期和指令周期没有一个严格的准则来进行区分，有些时候会混杂在一起。所以不是系统时钟周期就一定小于机器周期，两者可以相等。如果我们对 **MY_MCU_000** 的查表操作方式进行改良，取到代码数据后不是从头到尾逐个比较，而是直接从表的起始位置直接向后偏移所取数据大小，这样所有代码查表操作耗费的时间都是 0.1 秒，这样即便还是一秒触发就完成三步操作灯的亮灭也同样不会出现延时不同的问题，这个时候 **MY_MCU_000** 机器周期又变回等于系统时钟周期。

指令周期

明白了系统时钟和机器周期，理解指令周期就不再困难，指令周期和机器周期之间的关系类似机器周期和系统时钟周期之间的关系。

这里我不再用魔法师的例子而直接用单片机来解释，单片机所有的操作都一定包含单片机理论中的取指、译码、执行这三步，即便是执行这一步，耗费的时间都会有不同。比如将某个 IO 口设为指定状态，将状态直接送给 IO 就执行完毕，但如果是将内存里面的某个数加一，则需要经过“把这个数读给 ALU（运算器）”→“加一”→“存回原位置”这几个过程，显然耗费的时间要多。

如果这两个操作分别对应设置 IO 和内存变量累加两条指令，这两条指令耗费的机器周期数也就不同。当然设计人员可以设计为不支持内存变量累加这条指令，改成三条分指令：从指定位置读数到 ALU、ALU 加一、存 ALU 中内容到指定位置。这样设计就可以做到指令周期等于机器周期。

经过这些解释相信大家会认可“**指令周期≥机器周期≥系统时钟周期，三者没有绝对的区分准则**”这种观点，单片机的内部架构和设计方法的决定三者间的关系，从上面的例子中可以看出单片机支持哪些指令、一条指令用什么方法实现都会对三者间的关系产生影响。技术不断向前发展，象

51 系列单片机那种一个机器周期等于十二个系统时钟周期的设计会越来越少，现在新设计的单片机大都已经做到“**大部分指令周期=机器周期=系统时钟周期或两倍时钟周期**”。

1.6. 单片机指令和汇编语言

单片机指令系统和编程用的汇编语言是设计一个单片机必不可少的部分，指令系统可以间接告诉用户单片机系统的内部构架方式，用户通过指令系统即可知道单片机效能。单片机指令系统是数字构成的机器码，对用户来说非常不直观，所以在设计指令系统的时候就会同时设计出相应汇编语言，每条机器指令都有对应有一条汇编指令，以便进行理解和记忆。

设计单片机系统的人存在一个通病，想把所有功能都为用户实现。这种做法好处是可以把单片机硬件支持的功能最大限度的直接交给用户使用，不好的地方是指令系统会变得复杂。对于刚开始学习单片机的人简直就是一个噩梦，因为他不知道哪些是重点，只好强迫自己尝试记住全部指令并能理解。例如 51 系列单片机有 111 条指令，严重怀疑这个指令条数吓跑了不少初学者，我不想再次吓到大家，会找一个指令少一点的单片机指令系统来充当例子。

PIC12F629/675 指令集

助记符，操作数		说明	周期	14 位操作数 MSb LSb	影响的状态位	注释
针对字节的数据寄存器操作指令						
ADDWF	f, d	W 加 f	1	00 0111 dfff ffff	C,DC,Z	1,2
ANDWF	f, d	W 和 f 与运算	1	00 0101 dfff ffff	Z	1,2
CLRF	f	f 清零	1	00 0001 1fff ffff	Z	2
CLRW	-	W 清零	1	00 0001 0xxx xxxx	Z	
COMF	f, d	求 f 的补码	1	00 1001 dfff ffff	Z	1,2
DECF	f, d	f 减 1	1	00 0011 dfff ffff	Z	1,2
DECFSZ	f, d	f 减 1，为 0 则跳过	1(2)	00 1011 dfff ffff		1,2,3
INCF	f, d	f 加 1	1	00 1010 dfff ffff	Z	1,2
INCFSZ	f, d	f 加 1，为 0 则跳过	1(2)	00 1111 dfff ffff		1,2,3
IORWF	f, d	W 和 f 同或运算	1	00 0100 dfff ffff	Z	1,2
MOVF	f, d	移动 f	1	00 1000 dfff ffff	Z	1,2
MOVWF	f	将 W 的内容移动至 f	1	00 0000 1fff ffff		
NOP	-	空操作	1	00 0000 0xx0 0000		
RLF	f, d	f 带进位左循环	1	00 1101 dfff ffff	C	1,2
RRF	f, d	f 带进位右循环	1	00 1100 dfff ffff	C	1,2
SUBWF	f, d	f 减去 W	1	00 0010 dfff ffff	C,DC,Z	1,2
SWAPF	f, d	f 半字节交换	1	00 1110 dfff ffff		1,2
XORWF	f, d	W 和 f 异或运算	1	00 0110 dfff ffff	Z	1,2
针对位的数据寄存器操作						
BCF	f, b	f 位清零	1	01 00bb bfff ffff		1,2
BSF	f, b	f 位置 1	1	01 01bb bfff ffff		1,2
BTFSC	f, b	检测 f 的位，为 0 则跳过	1 (2)	01 10bb bfff ffff		3
BTFSS	f, b	检测 f 的位，为 1 则跳过	1 (2)	01 11bb bfff ffff		3
立即数和控制操作						
ADDLW	k	立即数加 W	1	11 111x kkkk kkkk	C,DC,Z	
ANDLW	k	立即数和 W 与运算	1	11 1001 kkkk kkkk	Z	
CALL	k	调用子程序	2	10 0kkk kkkk kkkk		
CLRWDT	-	看门狗定时器清零	1	00 0000 0110 0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	跳转	2	10 1kkk kkkk kkkk		
IORLW	k	立即数和 W 同或运算	1	11 1000 kkkk kkkk	Z	
MOVLW	k	将立即数移动到 W 寄存器	1	11 00xx kkkk kkkk		
RETFIE	-	从中断返回	2	00 0000 0000 1001		
RETLW	k	返回时将立即数存入 W	2	11 01xx kkkk kkkk		
RETURN	-	从子程序返回	2	00 0000 0000 1000		
休眠	-	进入待机模式	1	00 0000 0110 0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	立即数减去 W	1	11 110x kkkk kkkk	C,DC,Z	
XORLW	k	立即数和 W 异或运算	1	11 1010 kkkk kkkk	Z	

注 1: 当 I/O 寄存器作为自身的函数被修改时（例如，MOVF GPIO, 1），使用的值将是该引脚上的当前值。例如，如果某引脚配置为输入，其数据锁存器中的值为“1”，被外部器件拉为低电平时，则写回的数据锁存值将是“0”。
2: 当该指令的执行使用 TMR0 寄存器（以及 d=1）时，如果将预分频器分配给 Timer0 模块，则将其清零。
3: 如果程序计数器（PC）被修改或条件测试为真，则该指令需要执行两个周期。第二个周期执行一条 NOP 指令。

图 1.6. -1 PIC12F6XX 指令表

MICROCHIP 公司的 PIC12F6XX 系列单片机指令系统相对简单，只有三十多条，这套指令系统既然没 51 系列单片机指令那么复杂，那也就别指望给开发人员提供到有 51 系列单片机指令那么多功能，不要说进行乘除这样的操作，就是要实现一个查表功能都有点麻烦。出一道题：PIC 的单片机如何实现查表功能？本书中不给出答案，读者可以自己通过网络查找。

接下来告诉大家如何通过这个指令表来了解单片机的内部构架和效能，要注意只是可以间接的了解一部分，要想得到这方面详细而且准确的信息，还得去查看器件手册。

先看看指令周期，多数为一个周期，少数为两个周期，这说明该单片机指令效率不错。

●注：指令表字符说明：k 表示是数字，f 为寄存器（RAM 地址），W 是工作寄存器（ALU 用其进行运算处理），d 只能为 1 或 0（用来选择运算结果存放到工作寄存器还是 RAM 里面）。

GOTO k（跳转指令），对应指令机器码为 10 1kkk kkkk kkkk，没有进行条件限制的附加说明，

也就是这条指令可以跳到这个单片机能支持的存储空间内的任意位置，我们只要知道可以跳转的范围，也就知道最大存储空间有多大。指令机器码告诉我们可以跳转的范围是 k 所示一个 11 位二进制数（0~2047）。

从单片机的规格书中可以看到程序存储空间确实为 1024 字，也就是 2k 字节。

器件	程序存储器	数据存储器		I/O	10 位 A/D 转换器（通道）	比较器	8/16 位定时器
	闪存（字）	SRAM（字节）	EEPROM（字节）				
PIC12F629	1024	64	128	6	-	1	1/1
PIC12F675	1024	64	128	6	4	1	1/1

图 1.6. -2 PIC12F6XX 存储空间表

在 **1.2. 单片机是如何工作**一章中我曾提出一个问题“如果不给单片机提供程序，单片机能运行吗？”这里我们会通过对 PIC12F6XX 指令分析找到答案，首先我们要知道如果我们不烧写程序到该款单片机内部那用来存放程序的区域到底是什么内容，通过查询芯片手册可以知道该款单片机内部采用的是闪存，闪存在未使用的初始状态位是全为 1。

现在我们继续观察 PIC12F6XX 的指令表，会看到下面指令：

ADDLW k（工作寄存器加上立即数 k 后结果存回工作寄存器）

对应指令机器码为 11 111x kkkk kkkk （x 表示可以为 1 也可以为 0，k 为汇编代码中的立即数）。

现在闪存是未使用的初始状态，里面所有的内容都为 11 1111 1111 1111。

哈！初始状态的闪存里面的内容居然全是 ADDLW 255 指令，也就是说存在这样的可能，一个全新从未使用过的 PIC12F6XX 单片机，拿回来装好外围电路，会在里面执行工作寄存器加 255 的操作。

当然我的这个假设在 PIC12F6XX 上并不一定能真正发生，有可能设计这个单片机的人针对这种情况已经在内部做了某种保护，只是我不知道而已。

继续我们的假设，如果闪存在未使用的初始状态是全为 0，又是什么样的分析结果呢？

查看指令 NOP （空操作）

对应指令机器码为 00 0000 0xx0 0000（x 表示可以为 1 也可以为 0）。

现在闪存是未使用的初始状态，里面所有的内容都为 00 0000 0000 0000。

唉！这种假设的结果居然是闪存里面全为 NOP 指令，上电后单片机会在里面做空操作。

开发人员一定要学会多想问题，我们的假设是未使用的 PIC12F6XX 单片机理论上如果不加相应保护会把空闪存内容也当成代码执行，这一点能不能给我们设计指令系统给出一些启示呢？

当然会，单片机的 ROM 空间大都是 kBytes 的整数倍，实际写好的程序刚好用完所有 ROM 的几率非常小，这样将程序烧写到单片机里面在后面就有一段空余空间，这段空间的内容是通常是全 0

或全1。

单片机程序有时候会因为外部干扰让程序跑飞，如果刚好飞到这段空余空间里面，无外乎就是这两种结果：

结果一：全0或全1有对应的指令机器码，单片机错误的执行这些指令，跑到ROM的最后位置后单片机再怎么处理未知（PIC12F6XX的处理是跳到0地址循环运行）。

结果二：全0或全1没有对应的指令机器码，单片机只能是不知道该怎么做或者不停的按异常指令处理。

如果指令系统在设计的时候考虑全面一点，将全0和全1设计成复位指令，这样一旦程序因为干扰飞到空余空间就会复位重新运行。所以即便是同样的指令系统，只是具体的指令机器码定义不同都会在实际应用中产生区别，看来设计出一个稳定可靠的单片机还真是不简单。

写程序的开发人员也可以在这方面做出改进，在程序后面的空余空间填充上复位指令机器码，如果没有复位指令可以利用看门狗等单片机资源实现复位，或者直接跳到0地址，这些做法对产品抗干扰都有积极作用。

单片机的汇编语言实际上就是将指令系统的机器代码用简单的字母缩写对应，字母缩写要求能尽可能的让用户一看就明白是什么意思，便于理解和记忆，在 **1.2. 单片机是如何工作**一章中我们就定义过一个超级简单的汇编语言样例，用L1R0表示左输出高右输出低指令。

再看看真正的单片机汇编语言，还是看PIC12F6XX的指令表。

CLRF f 将功能寄存器清零，clear function register, register number is f

CLRW 将工作寄存器清零， clear work register

INCF f,d 将寄存器内容加一，increase function register,……

如果你非常讨厌英语，就是想实现中文编程的理想，你自己可以在PIC提供的编译器的技术上实现你的愿望，自己用中文定出与英文汇编指令对应的中文汇编指令，比如“CLRF”对应“功能寄存器清零”，再把英文的指令表和指令说明中所有英文汇编指令都改成你定义的中文汇编指令做成指令文档，写一个可以进行语法错误检查和中文汇编指令转换成指令机器码功能的电脑程序，用这个程序编译得到的代码烧进PIC单片机同样也可以运行。

当然单片机的汇编指令不是单纯地局限于与机器指令一一对应，写编译器的人会通过**伪指令**的方法来扩充汇编指令，以便让汇编编程更加简单灵活。**伪指令**意思就是伪造的指令，单片机指令系统里面并没有和这种汇编指令相对应的机器指令。

在实际产品的单片机程序编写中，会出现许多和单片机硬件指令无关的特殊需求，如果只是依靠和单片机指令一一对应的汇编指令，很难满足这些特殊需求。比如某些时候需要将某条代码放在指定的地址，如果单靠单片机指令来实现这个功能会非常复杂，只能是靠增减前面的代码数将这条代码地址前后移动来找到正确的位置，这样需要编程的人一条条的数代码并计算出地址，实际应用根本无法接受。另外有时候单片机程序需要存放一些不用改写的数据在ROM代码区，这些数据不是代码，程序也不会跳到这个地方当成代码来执行，要实现这样的功能单靠单片机指令也是很麻烦。

伪指令能很好的解决这些问题，通过定义与单片机指令没有对应关系的另外汇编指令，对汇编程序进行结构上的控制，或者方便汇编语言实现一些特殊功能。这里我们看看伪指令如何来解决前面函数放在指定地址和程序代码区放数据所遇到的问题。

定义出一条**伪指令** ORG，用法是按照 ORG xxxx 的格式进行，xxxx 是一个用来表示地址的数字，这条伪指令的作用是让编译器在编译的时候将其后面相临的代码放在 xxxx 地址。现在假定要在 100 这个地址固定去 CALL func，来看这条指令给汇编语言编程带来的不同。

无伪指令 ORG 汇编代码写法：

```
NOP
NOP
…靠编程者手工连续填 100 个…
NOP
CALL func
……
```

有伪指令 ORG 汇编代码写法：

```
ORG 0
…可以写代码，也可以空着，如果写代码该部分不能超过地址 100…
ORG 100  ;编译器通过 ORG 知道下一条指令放在地址 100 的位置
CALL func
……
```

	无伪指令		有伪指令	
地址				
0		NOP		NOP
1		NOP		NOP
2	人工输	NOP	自动填	NOP
…	入100个	…	充100个	…
98		NOP		NOP
99		NOP		NOP
100		CALL func		ORG 100 CALL func

图 1.6. -3　伪指令 ORG 效果示意图

很显然，用到 ORG 伪指令的汇编代码要精简许多，再看看如何利用伪指令在代码区放数据。定义出一条**伪指令** DB，用法是按照 DB xx 的格式进行，表示在当前位置厨房 xx 这个数据。假定在程序中某个位置要放 0 到 9 十个数。

无伪指令 DB 汇编代码写法：

```
……
```

从指令表中查出 0x00 对应的指令放在这里，如果没查到我也不知道怎么做

从指令表中查出 0x01 对应的指令放在这里，如果没查到我也不知道怎么做

……

从指令表中查出 0x09 对应的指令放在这里，如果没查到我也不知道怎么做

……

有伪指令 DB 汇编代码写法：

……

```
DB  0   ;编译器通过 DB 知道这是数据直接存放 0x00
DB  1   ;编译器通过 DB 知道这是数据直接存放 0x01
……
DB  9   ;编译器通过 DB 知道这是数据直接存放 0x09
……
```

如果将 DB xx 增强到支持 DB xx, xx, xx 这样的格式就采用下面更简洁的用法。

```
DB  0,1,2,3,4,5,6,7,8,9
```

单片机的宏定义也是一种伪指令，和其它的伪指令宏的最大特点是被用做替换，比如程序中有一段代码经常重复出现，这段代码又不适合用函数来实现，这个时候宏就被排上用场，可以用一个简单明了的名称将这段代码定义成一个宏，在需要调用这段代码的地方用所定义的宏来表示。

不同的单片机汇编语法各不相同，这里用一段 C 为蓝本的伪代码来就宏的用处进行说明。

```
#define  USE_MACRO  定义使用宏
#ifdef  USE_MACRO  通过这个宏可以进行条件编译，编译只选择一种代码编译
#define SET_IOA0_OUTPUT_HIGH  读回 IOA 输入输出方向设置指令\
                                设定 IOA0 为选择输出指令\
                                读回 IOA 输出状态指令\
                                设定 IOA0 输出为高指令
代码…
SET_IOA0_OUTPUT_HIGH
代码…
SET_IOA0_OUTPUT_HIGH
代码…
SET_IOA0_OUTPUT_HIGH
代码…
#else
代码…
读回 IOA 输入输出方向设置指令
```

```
设定 IOA0 为选择输出指令
读回 IOA 输出状态指令
设定 IOA0 输出为高指令
代码…
读回 IOA 输入输出方向设置指令
设定 IOA0 为选择输出指令
读回 IOA 输出状态指令
设定 IOA0 输出为高指令
代码…
读回 IOA 输入输出方向设置指令
设定 IOA0 为选择输出指令
读回 IOA 输出状态指令
设定 IOA0 输出为高指令
代码…
#endif
```

采用宏和不采用宏用编译器编译出来的最终机器代码是一样的，只是在程序中显示方式繁简会不一样，对比两种方式所写的程序代码，有用宏的程序代码要简洁直观不少。

1.7. RAM/ROM 的作用

RAM 和 ROM 都是单片机的存储器，先通过两个图来解释存储器的实现，实际的存储器实现方法和这里的图示会不一致，但基本原理相同。

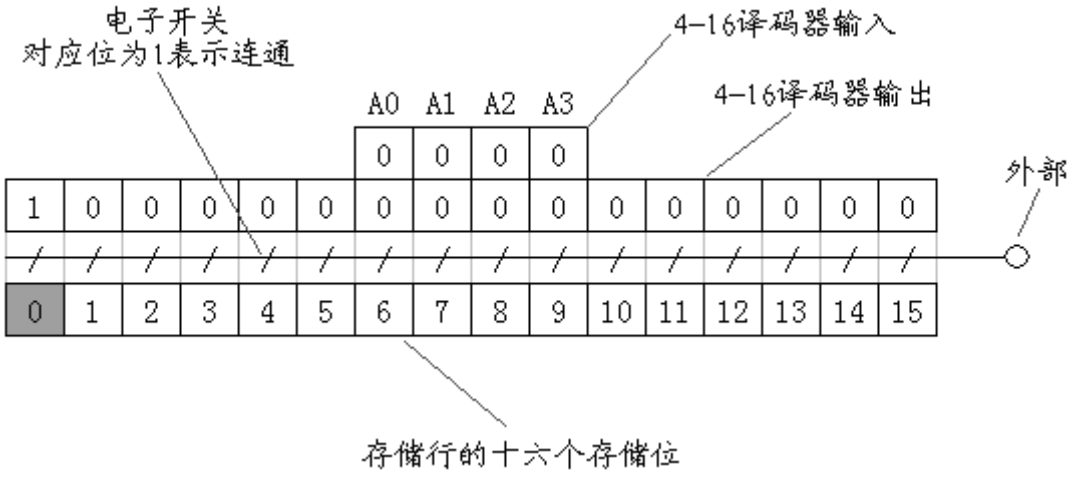


图 1.7.-1　十六位宽存储行选通示意图

上图是一个有十六个存储位的存储行，A[3:0]四位地址做为一个 4-16 译码器输入端会让该译

码器的十六个输出端输出一个 1，其余全为 0。4-16 译码器的十六个输出端分别控制十六个电子开关，为 1 导通。示例中的 A[3:0]=0b0000，4-16 译码器的第一个输出端为 1，外部数据线被连接到存储行的 bit0 上。

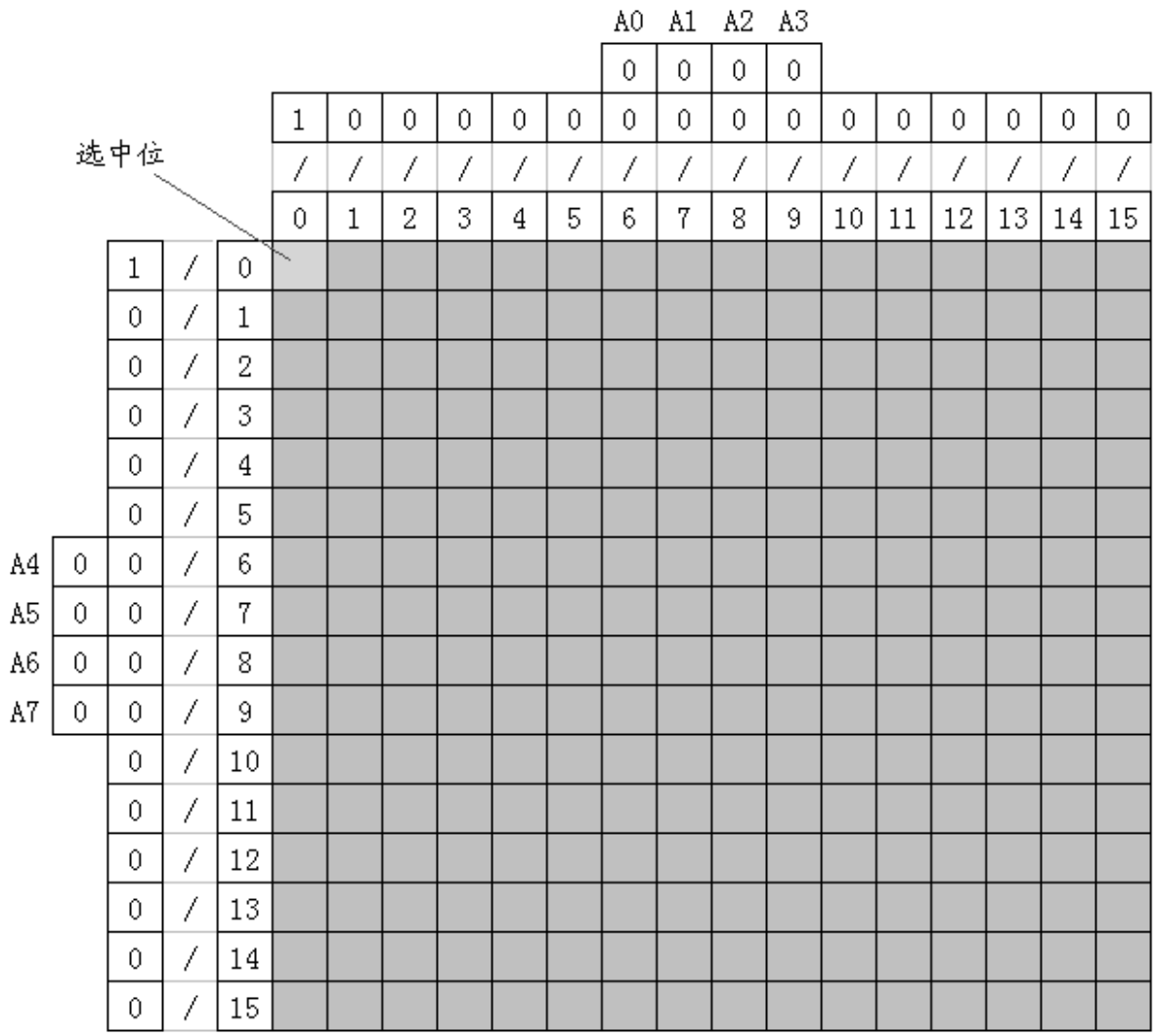


图 1.7.-2　16*16 存储阵列选通示意图

将这个原理实现的存储行进列扩展，只有对应行列 4-16 译码器输出端都为 1 时才连通对应存储位到外部数据线上，上图表示选中了行和列都是第一的存储位。如果 A[3:0]和 A[7:4]并行控制和上图一样的八个存储阵列，就可以实现八位宽的数据读写，同样也可以实现十六位和三十二位宽的数据读写。

这样就可以将存储器理解成一个存储阵列，由地址线决定数据线和里面哪个位置相连通，设计的时候一般都尽量设为行和列数目接近的阵列，这样内部硬件电路实现起来要简单一些。上面 16*16 存储阵列的例子，只需要两个 4-16 译码器就可以实现行列地址译码，如果设计成 256*1 这样的一个单行序列，则需要十六个 4-16 译码器才能实现地址译码。

ROM 里面存储的内容只能读，不能被修改，断电后内容不会丢失，RAM 里面存储的内容既可以

读也可以被修改，断电后里面的内容丢失。RAM 实现的电路要比 ROM 复杂，这样同样大小的 RAM 和 ROM 相比，RAM 需要更多的电路才能实现，所以简单的单片机系统为了降低成本，大都只提供少量的 RAM 空间供程序使用。

通常来说单片机程序都是存放在 ROM 里面，RAM 则是用做程序运行所需的变量。但不要有这样的误解：ROM 里面全是代码，RAM 里面全是数据。对于单片机程序而言，程序包含代码和数据，我们说的是大多数时候程序都存放在 ROM 里面，这样就是代码和数据大都存放在 ROM 里，实际上 RAM 是不会用来存放任何东西的，RAM 的特点断电里面的内容就会丢失，如果存放到 RAM 里面下次上电只能得到一片空白。RAM 被用来存放程序运行的中间变量，这些中间变量只在程序运行当中起作用，在断电时并不需要被保存。RAM 虽然不能存放代码，但可以用来临时装载代码并执行，嵌入式系统代码都是从 ROM 装到 RAM 中后才被执行的，后面会讲一个利用 RAM 来动态装载不同程序代码运行的例子。

一些简单的单片机并不提供真正的 RAM，而是用通用功能寄存器来当 RAM 用，这种情况我们可以不用管两者之间的差异，直接把通用功能寄存器等同 RAM 来进行理解。用通用功能寄存器来当 RAM 用的单片机为了处理上的方便，在逻辑上会将 ROM 和 RAM 地址都从 0 地址开始，两者并行重叠，读 ROM 和 RAM 用不同指令来进行区分。

1.8. 单片机接口

单片机的接口就相当于人的眼耳口鼻舌手脚这些器官，单片机通过接口来与外部交换信息，交换信息的过程叫通讯。接口可以分为输入和输出，单片机通过输入接口可以知晓外部电路或器件的状态信息，通过输出接口则可以将它的程序思想告诉外部电路或器件。

在单片机技术发展的过程中，人们发现如果对接口制定一些规则会更利于技术的发展，设计单片机的公司可以潜心在单片机构架等方面发展单片机技术，另外一些专业公司则将自己独有的技术设计成带某种接口的模块，各自发挥自己的优势技术同步发展，于是大量的接口标准被制定出来。

需要支持的接口越多，单片机就越复杂，外部管脚也会越多，所以单片机会根据自己应用的方向决定支持哪些接口。象 IO、UART、SPI、I2C、I2S、USB、TFT、CSTN、SD、MMC、CF、CSI 等都是单片机的常用接口。

各个公司在设计一款单片机的时候，会根据这款单片机的主要应用方向除了 IO 外还会选用可能需要使用到的接口，比如现在有一款除了 IO 外还能支持 UART、SPI、I2C 这三种接口，在 IO 之外再将这三种接口独立的加进去，就会使外部引脚增多，单片机的封装增大。对于大多数应用，不会将这三种接口同时用上，于是采用一种 IO 复用的方法来保证单片机引脚不增多，就是内部有这些模块，但可以通过内部的电子开关选择外部引脚来选是普通 IO 还是特殊接口，由产品开发工程师在程序中进行选择配置。

图示为 IO[1:0]和 UART 复用，IO[5:2]和 SPI 复用，IO[7:6]和 I2C 复用，如果现在产品要使用

UART 功能，但不需要 SPI 和 I2C 功能，就可以将选 UART 的电子开关设置成选 UART，选 SPI 和 I2C 的电子开关设置为选 IO，这时单片机的管脚 1~2 是 UART 的 TX 和 RX，管脚 3~8 是普通 IO。

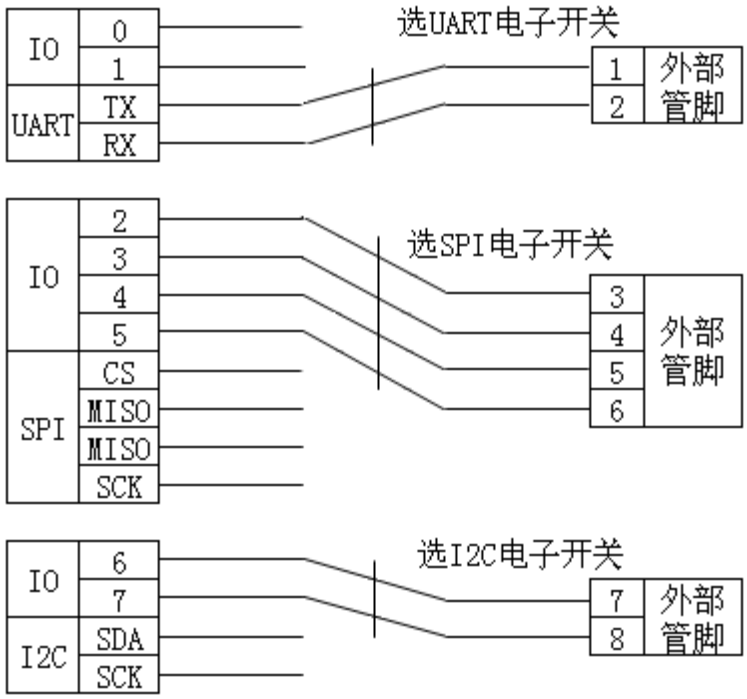


图 1.8.-1　IO 复用示意图

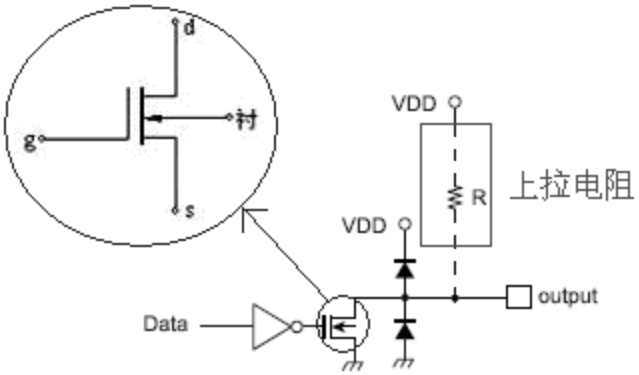
IO

对于单片机来说，最基本的接口无异是 IO（现在开始流行叫 GPIO），也就是通用输入输出口。对于 IO 口用做输出时只能输出高低电压两种状态，用做输入时同样只能判断出外接的电压是高还是低，功能虽然简单，但数字处理器也只能处理 1 和 0 两种状态，所以 IO 口对外的逻辑控制上非常实用，可以说只要是单片机，就一定会提供 IO 口给开发者使用。

只支持高低两种状态就自然有一定局限性，在有些时候对于一个单片机应用系统会希望控制更精细一些，想知道外接电压的大小或者控制输出电压的大小。这样在普通 IO 的基础功能上进行扩展，出现带 ADC/DAC 功能的 IO 口，带 ADC 功能 IO 口可以测量出外接电压的大小，带 DAC 功能的 IO 口可以输出一个自己可以控制大小的电压。

支持 ADC/DAC 功能的 IO 口内部的电路结构要复杂许多，而且一个电子产品并不是每个 IO 都需要支持 ADC/DAC 功能，所以有的单片机可能没有 IO 可支持 ADC/DAC，或者是只有少数的 IO 支持。如果万一出现需要非常多路 ADC/DAC 的实际应用，可以选择专门的 ADC/DAC 芯片来实现。

使用 IO 的时候会遇到 Tri-State、Open Drain 等不同的驱动方式，到底有什么不同《数字电子技术基础》一书中有详细描述，在应用当中记住如果是 Open Drain 需要接上拉电阻，否则不能正常输出高电平。



上拉电阻	Data	Vgs	MOS管	output	说明
无	1	0	截止	浮动	d脚因为MOS管截止处于悬浮状态
	0	VDD	导通	低	d脚短接到地
有	1	0	截止	高	d脚被上拉电阻R拉高
	0	VDD	导通	低	d脚短接到地

图 1.8.-2 Open Drain 输出示意图

UART

UART（串口）是用来进行异步串行通讯的接口，了解这种接口之前需要知道什么是异步串行通讯。通讯是两方或者多方进行信息沟通，最基本数字通讯只传递 1 和 0 两种状态，通讯的时候将所传递的数字转换成二进制，再依次进行传送。比如我们现在想传送 0x12，用二进制表示是 0010 0001(低位)，如果是先低位后高位就要按这样的次序传送：1→0→0→0→1→0→0，这就是串行方式。

对于单片机的 UART 接口，比电脑的串口要简单，只需要三条线就可以进行通讯，负责发送的 TX、负责接收的 RX 和地线 GND。

图 1.8.-3 UART 通讯连接图

单片机 UART 采用的是 TTL 电平信号，TX 为高表示传送 1，为低表示传送 0，同样 RX 为高表示当前接收 1，为低表示接收 0。不要与电脑的 RS232 所采用的负逻辑混淆在一起，RS232 用-5V 到-15V 表示 1，5V 到 15V 表示 0，-5V 到-15V 表示无效噪声（也有说是±3V 到±15V），为什么用这么一个奇怪的反向电压来表示 1 和 0 的原因不明（解释是为了抗干扰）。因为存在这样的差异，所以单片机串口和电脑的串口进行通讯时需要用进行电平转换，比如 MAX232 就是实现两者电平转换的专用芯片。

回过头来接着解释异步，通讯是一方把信息传给另外一方，虽然我们规定是按次序一位一位的

传送，但还不能让发送方发送的信息被接收方准确解读。如果发送是每一位持续时间为一秒，接收是每一秒去读一次接收状态，这样是双方是可以正确传送数据；但如果发送还是维持一秒，接收改为每两秒去读一次接收状态呢？显然双方此时不能正确传送数据。这样就要求规定双方在收发数据位时，要采用同样的时间间隔，由于收发双方各自采用自己的时钟基准，就称为异步。

由于收发双方各自采用自己的时钟基准，所以两方的时钟会不相同，计算出的时间间隔自然会有差异，如果连续发送数据，这个差异会被累加下去，最终导致通讯出错。比如发送方用自己的时钟算出 1 秒的间隔等于绝对时间的 0.9 秒，而接收方刚好是 1 秒，现在发送方以每秒一个位的次序连续发送数据，当发完 10 个数据位的时候，绝对时间才过去 9 秒，接收方因为是以绝对时间 1 秒为间隔去读接收状态，就只读到 9 个数据位，显然已经出错。另外只定义高为 1 低为 0，接收方读接收状态要不就为 1 要不就为 0，这样接收方无法知道发送方是不是在发送数据。

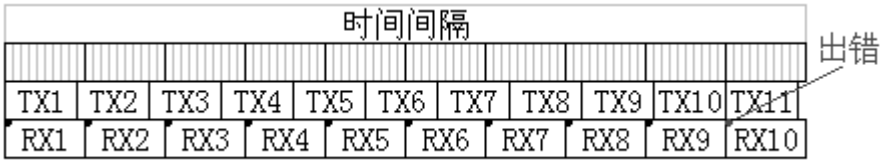


图 1.8.-4　时间间隔累计误差示意图

为了解决这两个问题，对异步串行通讯的协议除了增加时间约定外还有控制信息约定，这就是我们常见用串口通讯时设定的“9600/n/8/1”此类参数。规定平时空闲状态 TX 保持高电平，当有数据发送的时候先发送一位 0，接收方检测到这个 0 就知道发送方开始发送数据。双方约定好每个位的时间宽度，然后从低位开始以这个宽度发送数据位，数据位发送完毕依照约定决定是否发一个校验位，最后发送一个约定宽度的停止信号。这种做法每次发送的总位数最多也就十多个，两边时间间隔不同误差所造成的累计误差完全可以控制在允许范围内，从而保证不出错。

“9600/n/8/1”表示数据位宽是 1/9600 秒、无校验位、8 位数据位、1 位停止位。

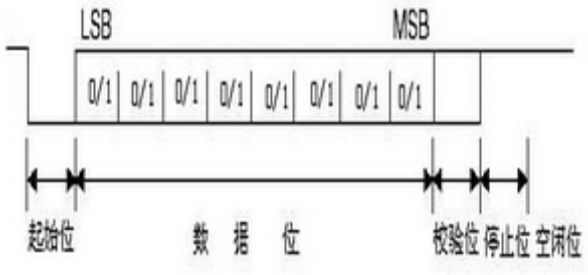


图 1.8.-5　UART 通讯时序图

异步串行接口除了 UART 外还有 IRDA、USB 等，要注意的是 USB 可支持非常高的数据速率，比如 USB2.0 最大速率可以到 480Mbps，这在传统的串行通讯上是不可思议的速率，其传输信号与传统

串行传输的方式不一样，是采用两条信号线差分驱动和其它一些特殊技术来实现高速率传输。

同步接口

异步串行通讯口 UART 需要在使用上另外增加一些约定，这些约定会影响传送效率，而且发送方和接收方的时钟基准要相近，即便如此，还是会有机会出错。

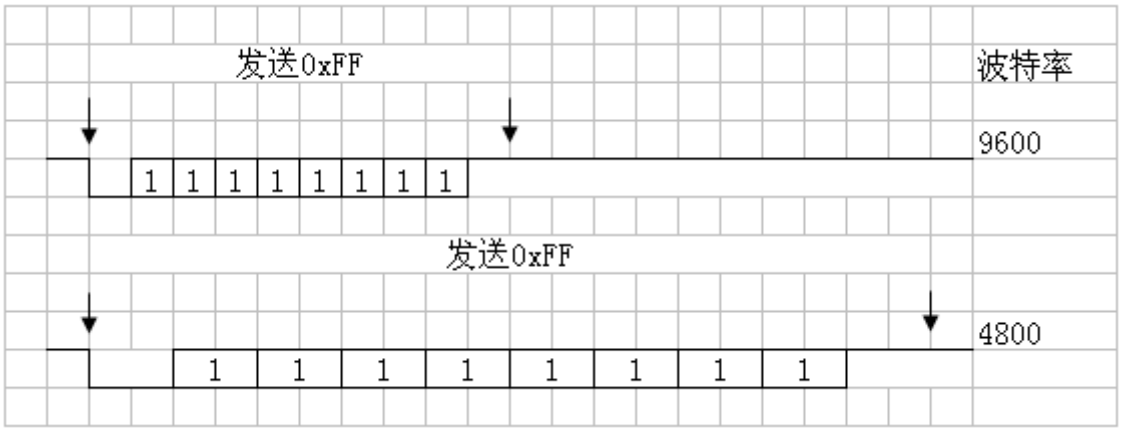


图 1.8.-6　UART 发送 0xFF 波形图

对比上图中的两个波形，如果 TX/RX 定义为 8 位数据位、无校验、1 位停止位，发送方以 9600 的波特率发送 0xFF，如果接收方将波特率定义为 9600，显然能正确接收到 0xFF。但如果将波特率定义为 4800，一样可以收到数据，而且数据也是 0xFF。这样看来异步串行通讯出错的风险还是比较高，要减少这种风险，就要在时间基准上想办法。解决方法就是增加同步控制信号，让发送方和接收方都使用同一个时间信号来做时间基准，SPI 就是这样的同步接口。

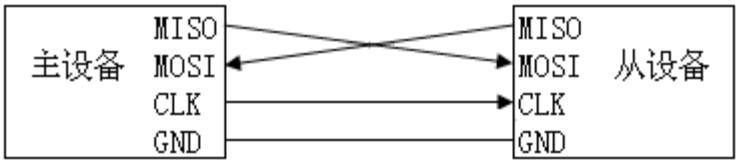


图 1.8.-7　SPI 通讯连接图

和 UART 相比，SPI 增加了一条线 CLK，这条线是时钟线，由主设备给出，时钟线 CLK 每高低变化一次，MISO 就输出一个数据位，同时 MOSI 读入一个数据位，这样只要 CLK 信号不超过两侧设备的最高限制速度，无论传多少数据，都不会出现类似 UART 时间误差被累计的情况。

通过一个 SPI 的时序图来看看 SPI 的信号是如何传输控制的，忽略片选信号 CSN 的作用，这个时序图显示该 SPI 接口在 CLK 的上升沿读 MOSI 状态得到当前输入数据位，在 CLK 下降沿向 MISO 输出当前数据位。

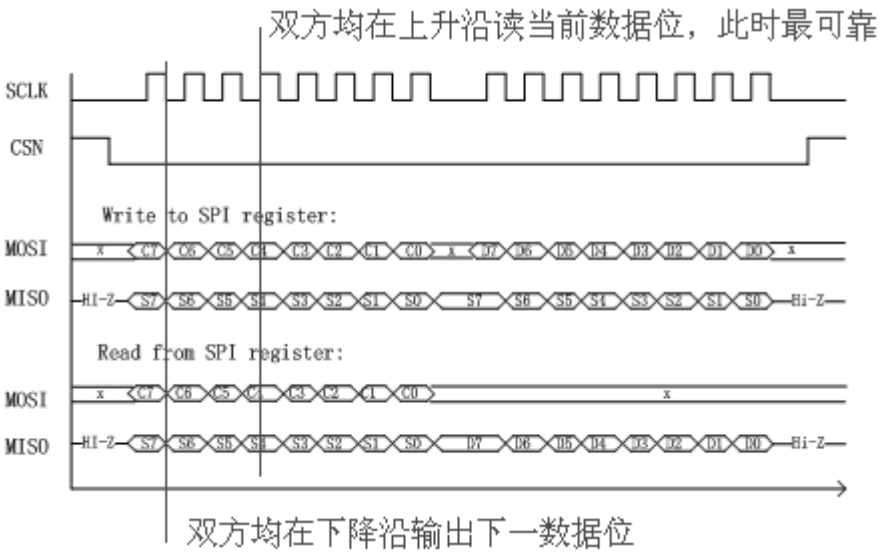


图 1.8.-8 SPI 通讯时序图

SPI 接口发送方和接收方都通过这个 CLK 的上下跳变来触发操作，也就是收发一个一个数据位，都会由 CLK 信号进行同步，这样就即使数据无穷无尽的收发下去，不会出现前面异步传行通讯所遇到的误差累计问题。

I2C 是也是一种常见的同步串行接口，和 SPI 的区别是 SPI 收发是两条独立的信号线，而 I2C 是共用一条，SPI 可以收发同时进行，但 I2C 同一时刻只能从收发操作中选一种。对于 I2C 接口最常见的应用就是读写 EEPROM，SPI 常用于 SPI FLASH 的读写。另外象 SIO、I2S 等也都是同步串行接口。

用时钟信号进行同步只是一种最简单的同步方式，有些时候单独用时钟信号来进行同步是不够的，比如 CSI 这类数字视频接口，除了时钟信号同步外还有场同步、行同步信号。

并行接口

与串行通讯对应的是并行通讯，串行一次只能传送一个数据位，并行则可以传送多个数据位，理论上并行传送的数据位宽是多少，速率就是对应串行通讯的多少倍。当需要进行高速数据传输时，并行接口就被排上用场，最常见的就是打印口，不过单片机一般不支持打印口，一些功能强点的单片机会有 IDE、CSI、TFT、SD、CF 等并行接口。

并行接口先天存在一个不足，因为用途就是高速数据传输，象 UART 为了避免时间误差的累计而增加起始位这样的做法会严重影响其高速性，所以异步方式对其来说完全不适用，只要是并行接口，就一定是同步接口。同步接口使得原本就不少的信号线进一步增多，高速性则限制了信号线的长度，这样凡是同步接口的设备，最大通讯距离都不长。

AV 接口

前面介绍的都是数字接口，日常生活中还有一种特殊的接口应用非常广泛，就是视听产品所带

的 AV 接口，AV 接口是模拟接口，本质就是音频、视频的 DAC 输出，音频 DAC 除了增强功率输出外和 DAC 并无太大区别，但视频 DAC 则不同，内部采用的 DAC 是依照电视信号标准专门设计的，输出信号有着自己独有的格式特点，具体细节请自行查阅电视信号相关资料。

1.9. 接口驱动能力

每种接口都有自己的驱动能力，这个驱动能力又分成电气性能的模拟驱动能力和扩展性能的数字驱动能力。接口驱动能力常常被人忽视，有不少这样的单片机应用工程师，在完成产品开发后问他所用接口的驱动能力，他会一脸茫然，内心会想产品都已经做了出来那接口驱动能力肯定没什么问题，用不着关心。

接口驱动能力是一项非常重要的参数，在规划设计可对系统进行功能扩展的产品时尤显重要。如果工程师在设计产品的初期阶段能习惯性地去考虑接口驱动能力，那说明他已经具备负责整体项目开发的基本素质。为了体现接口驱动能力的重要性，特意将其做为一个专门的章节进行解释。

IO 接口模拟驱动能力

IO 口有驱动能力，先忽略 IO 是什么电路实现的这些细节，做个小实验来看看 IO 的驱动能力，找一个 IO 口可以直接输出高低电压的单片机，将其中一条 IO 输出高，并将这个 IO 口接一个电阻到地，改变这个电阻的阻值，然后去测量这个 IO 口的输出电压，看看有什么情况发生。

……（详见完整版）

UART 接口模拟驱动能力

来看看数字接口电气性能方面的模拟驱动能力，UART 因为是靠 TX/RX 与 GND 之间的电压高低来表示 1 和 0，先建立一个 UART 通讯的电气特性示意图。

……（详见完整版）

I2C 接口数字驱动能力

不是每种数字接口都会有体现扩展性能的数字驱动能力，这里以 I2C 接口为例进行讲解。常用的 EEPROM 大都是采用 I2C 接口，它支持多片 EEPROM 进行并联，我们选用 ATMEL 公司的 AT24CXX 的资料来做分析。

……（详见完整版）

1.10. 方便实用的中断

每个周末丁丁小朋友的父母会要求他独立完成一些家务，来培养他的劳动习惯，家务是固定的

三件事情：烧两壶开水、炖一锅排骨、将家里的地板拖一遍。如果单独完成这些事情，烧一壶开水大概需要十分钟，炖排骨大概二十五分钟，拖地板大概需要三十分钟，烧开水只要等水开了倒进保温瓶里，排骨炖好后关掉火就行。

第一周，丁丁小朋友先开始烧水和炖排骨，然后去拖地板，为了看水有没有烧开和排骨有没有炖好，拖一会地板就要停下来跑到厨房去看一看，这样看一次需要一分钟，总共看了十次，四十分钟后三样家务全部做完。

虽然四十分钟把家务全部做完，但丁丁小朋友是隔几分钟才去看一下水有没有烧开，于是水被烧开了一会丁丁小朋友才发现，水烧开后从壶里溢出流到煤气灶上，有点危险，显然从家务完成的质量来看不是很理想。

第二周，丁丁小朋友吸取了上周的经验，烧水换用水烧开后可以自动鸣笛的壶，排骨有上周的经验知道炖二十五分钟火候差不多，于是炖的时候用一个闹钟定时二十五分钟，接下来专心开始拖地板。大约十分钟后，第一壶水烧开鸣笛，丁丁小朋友停下拖地板去把水倒进保温瓶接着烧第二壶，继续拖地板；又过了大约十分钟，第二壶水烧开，丁丁小朋友同样处理；二十五分钟时间到，闹钟响起，丁丁小朋友过去看排骨，已经炖好于是关火，接着拖地板；三十三分钟，地板拖完，家务全部完成。

和第一周对比，时间少用了七分钟，而且水一开就去倒掉，消除了潜在危险，完成的质量自然要好一些，看来日常生活中的一些事情，不同的处理方法做出来的效果也会有明显差异。

丁丁小朋友做家务的例子对应单片机同时需要处理几个工作任务的两种基本方法：轮流查询和中断响应。开水烧开了不马上处理就会有危险，拖地板被打断有延时不会发生什么意外，但烧开水只要把水倒进壶里烧就行，烧的过程中并不需要做其它事情，拖地板则需要一直拖到全部地板拖完。如果说第一周的方法是轮流查询那二周的方法就是中断响应，水烧开鸣笛和闹铃为中断发生信号，从丁丁小朋友两周完成的结果可以看出中断响应效果要好过轮流查询。

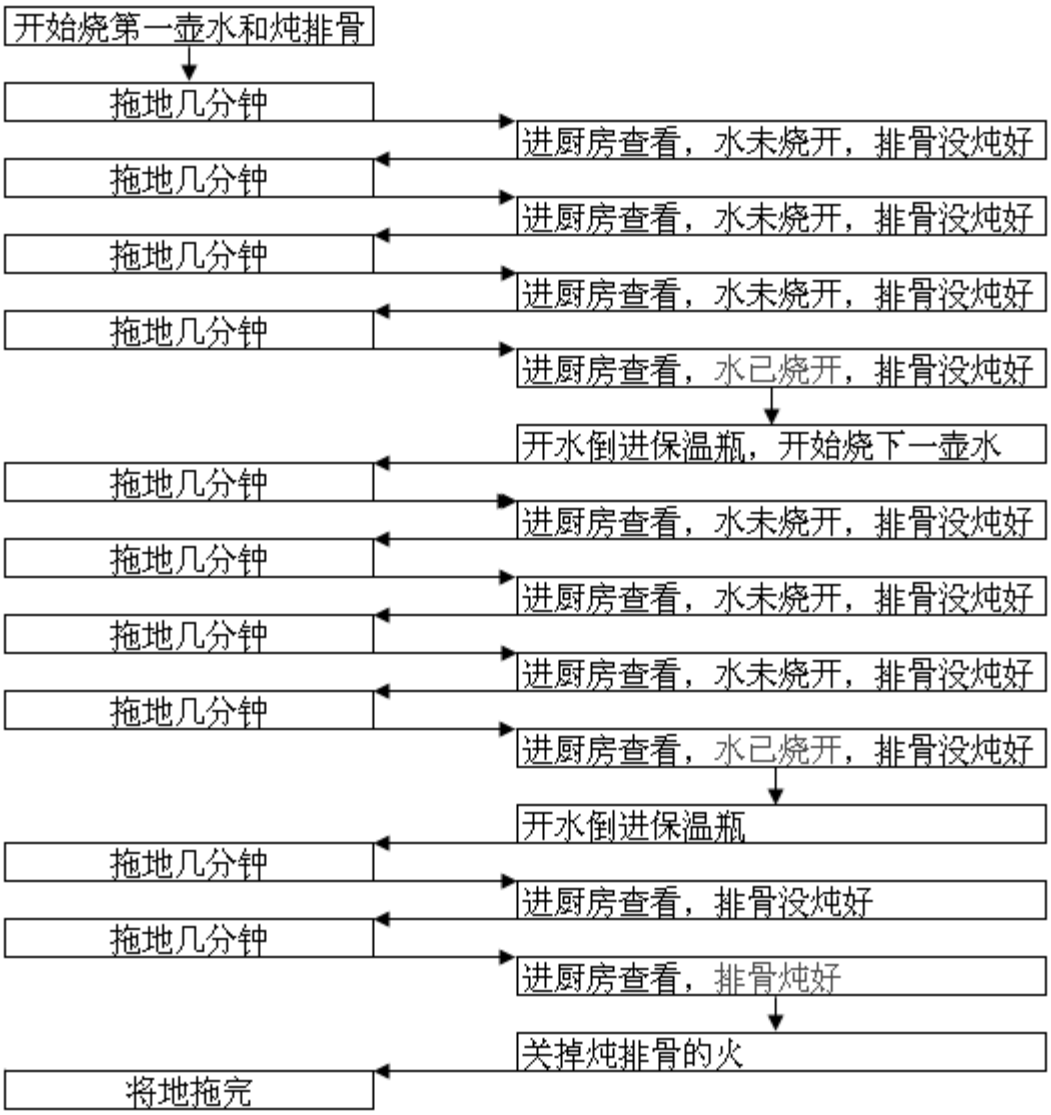


图 1.10.-1　丁丁第一周家务流程图

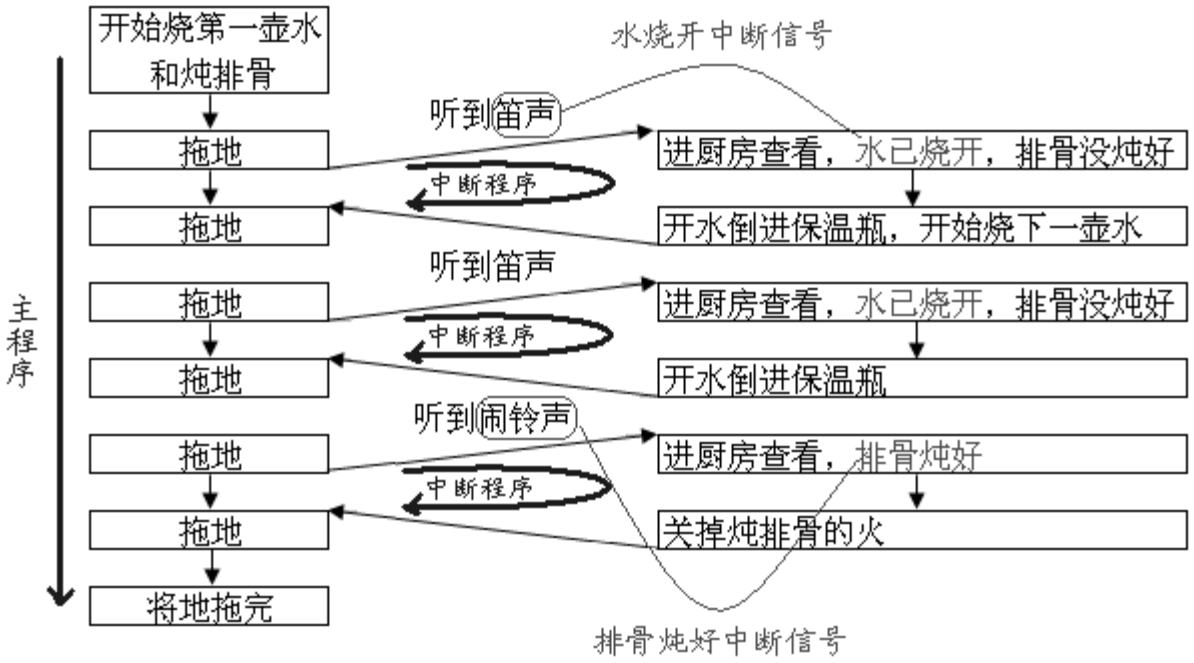


图 1.10.-2　丁丁第二周家务流程图

序言中就说过单片机技术是一门实用工程技术学科，和日常生活息息相关，正是为了应付丁丁小朋友做家务例子中烧水炖排骨这类问题，单片机有了中断的概念。中断就是在工作过程中突然有更紧要的事情去要去处理，于是将当前的工作打断，处理好更紧要的事情后再继续当前的工作。单片机的中断可分为两大类：一种是单片机内部控制电路在某种条件下产生的叫内部中断，另外一种则是由单片机外部器件产生的叫外部中断。

丁丁小朋友烧水和炖排骨对于他是两个独立的外部事情，这两个外部事情所产生的“中断信号”分别属于外部中断和内部中断。水烧开是水壶主动发出笛声，这个笛声和丁丁小朋友没有直接的关联，他不知道具体会在什么时候响，只要水开就会由水壶产生并传到丁丁小朋友的耳朵里，笛声是他的“外部中断信号”；闹铃是丁丁小朋友用他的闹钟来产生的，和炖排骨没有直接联系，只是因为丁丁小朋友知道排骨二十五分钟可以炖好才设置成这个时间，他自己是知道闹铃什么时候会响，只是他不想频繁地去看时间才用闹钟定时，闹铃声是他的“内部中断信号”。

通过丁丁小朋友做家务的例子我们明白了中断的原理和方法：单片机在工作的时候往往需要处理多个事情，有些事情只并不需要单片机时刻进行控制，只是需要在某些特定的条件下由单片机做出相应处理，有些事情则需要单片机花比较多的时间逐步控制，一旦停止控制就无法进行下一步操作，中断的引入可以让单片机面对这样的问题时有更高的工作效率，对于不需时刻进行控制的事情在需要被干预时发出中断信号让单片机来进行相应处理，需要时刻控制的就由单片机主程序循环持续控制。

单片机中断分为内部中断和外部中断两大类，外部中断由单片机外部设备产生，中断产生后通

过单片机的外部管脚传递给单片机，传递这个中断信号最简单的方法就是规定单片机的管脚在什么状态下有外部中断产生，这样单片机通常是有一个或多个 IO 口当在输入状态时可以用来检测外部中断信号。有外部中断产生的条件通常也就是这五种：IO 口输入为高、IO 口输入为低、IO 口输入由高变为低、IO 口输入由低变为高、IO 口输入由高变低或者由低变高。

一个连接到单片机的外部设备，如果想要使用单片机的外部中断，就必须在自己请求单片机中断响应的时候给单片机提供单片机在这五种信号中所支持的类型来触发单片机中断。程序运转中，一个中断不是只产生一次，一般都会间隔持续产生，这五种外部中断触发信号前四种都有一个问题，就是外设发出请求中断信号后如果信号请求线状态不改变，外设会无法向单片机提供下一次中断请求信号。让我们来看看以单片机和外部设备采用负跳变触发中断为例的触发情况。

外部设备以负跳变触发单片机中断，第一次中断请求外部设备的中断请求输出脚可以从高变低，触发单片机中断，第一次中断请求发生后中断请求脚保持输出低，外部设备无法产生第二次中断的触发负跳变信号。

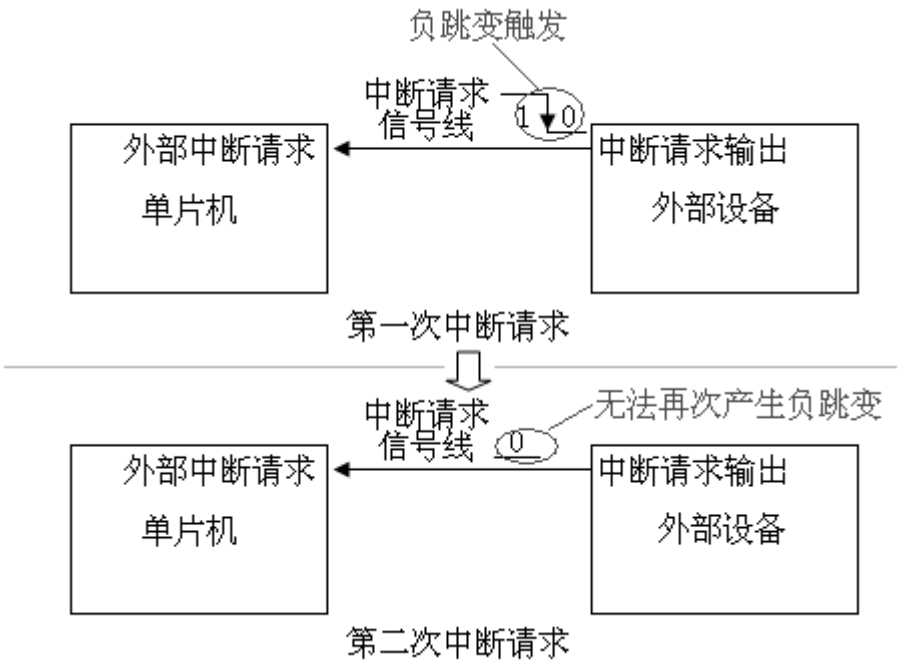


图 1.10.-3　外设只能产生一次中断请求信号示意图

将外部设备的中断请求信号做出修改，原来为需要中断时只是输出从高到低变化，现在改为输出先从高变到低，经过一小段时间后自己从低变回高，这样就可以每次需要中断时都能向单片机输出负跳变触发信号。

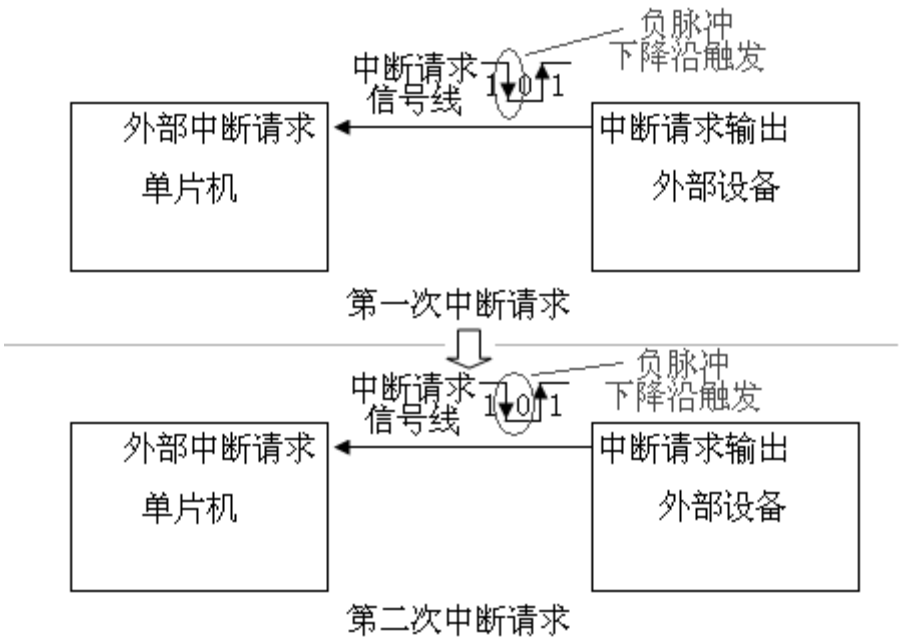


图 1.10.-4　外设可连续产生中断请求信号示意图一

或者是由外部设备提供某种接口，单片机通过该接口可以对外部设备进行中断清除操作，中断清除操作可以让外部设备的中断请求输出脚恢复到高。

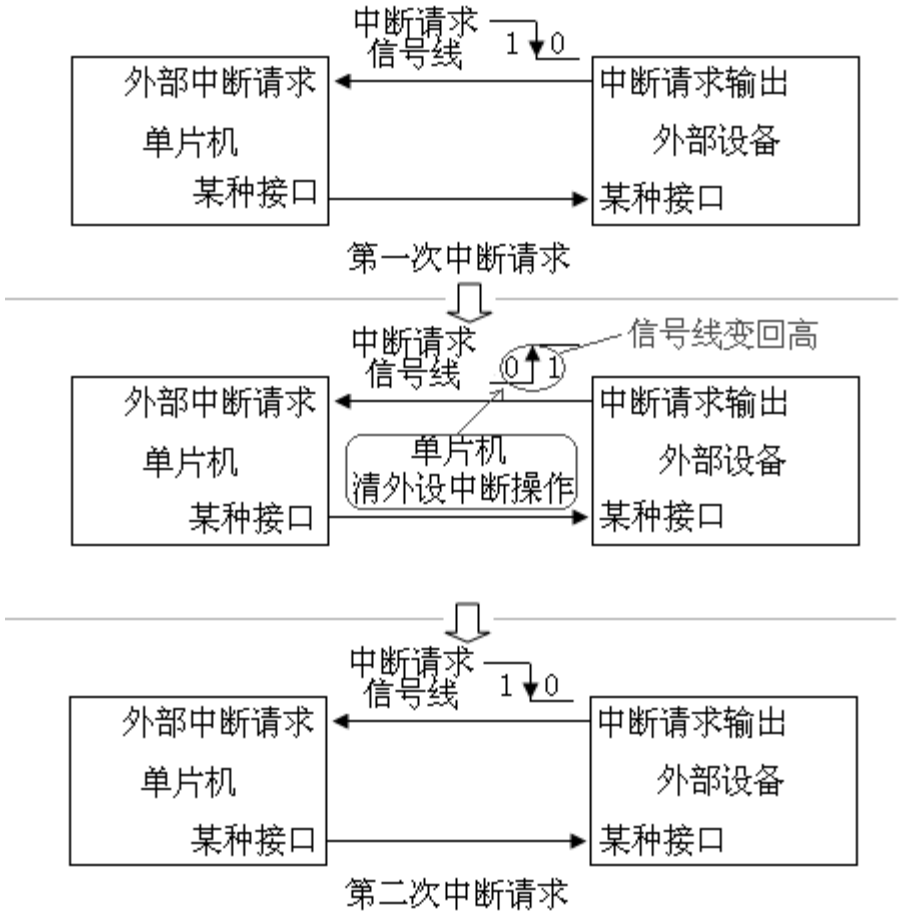


图 1.10.-5　外设可连续产生中断请求信号示意图二

外部中断触发还有一些特殊方式，比如外部脉冲宽度测量、外部脉冲计数等，这些方式都是在前面几种基本触发方式上进行功能扩展得来的，外部脉冲宽度测量就是当中断信号线跳变时会启动内部一个计时器，到下一次中断信号线跳变时通过计时器得到脉冲宽度并重新启动计时器，这些方式很少会使用到，不做详述。

内部中断是指单片机内部的功能模块产生中断信号，只要是单片机内部在 CPU 外围能独立工作的功能模块都会提供中断功能，常见的内部中断类型有时钟 Timer、串口 UART、模数转换 ADC 等。内部中断的工作流程和外部中断没太多区别，只是中断请求信号是在单片机内部进行传输，中断信号不是管脚上的电平状态，而是一个寄存器里面的相应标志位，通常当某个内部中断产生中断请求时就会将相应标志位置为 1，CPU 响应中断时将这个标志位清 0。

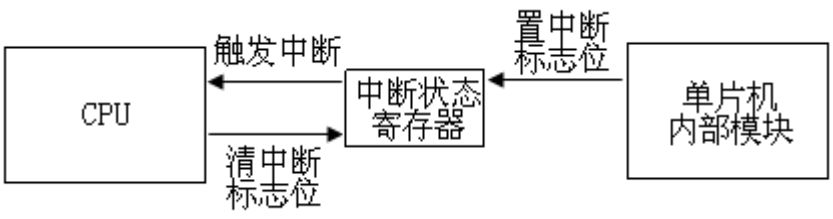


图 1.10.-6　内部中断触发示意图

单片机对中断标志位的处理方法没有统一标准，具体的约定方法要看单片机文档。大部分是标志位为 1 有中断产生，但有少数单片机是标志位为 0 有中断产生；有的单片机对中断标志位是 CPU 写入什么就是给改写成什么，有的则是规定必须通过写 1 或写 0 来实现清除操作，还有少数只要读一下中断标志位就会自动清除掉该标志位。

如果单片机不想被外部中断触发，大不了将用于连接外部中断触发信号的管脚接成不会触发中断的电压状态就可以，但内部中断无法去改变内部连线，所以单片机为了可以选择中断是否可以被除法，在其内部会有相关的寄存器来进行选择，通过里面的控制标志位开发人员可以根据实际情况决定是否使用中断。通常单片机里面有一个总控制位，这个位可以控制所有中断的开与关，然后每一种中断自己还有一个独立的控制位决定自己的开与关，如果想使用某个中断，就需要将总中断开关和对应中断的开关都打开。

当单片机有中断信号产生时，就会触发对应中断，不同的中断源会需要不同的响应方法，也就是说不同的中断产生的时候，需要单片机程序依照不同的中断源做出不同的响应，这就是中断服务程序。如果是 UART 收到新数据产生中断，应该是 UART 中断服务程序将数据读回来并做处理，如果是 ADC 转换完成产生的中断，需要的则是 ADC 中断服务程序将数据读回来并做处理。如果需要清中断标志位动作，一般都是在中断服务程序里面完成。

不同的中断源需要与之对应的中断服务程序，实际开发中并不是所有的中断都会被用到，开发人员为了节约程序代码空间会只写出自己要使用到的中断服务程序，也就是说会有一些中断没有与之对应的中断服务程序，如果触发了这样的中断，单片机程序会运行出错，前面中断各自独立的控制位就排上用场，将这些控制位关掉，相应中断就不会被触发。

单片机开始上电的时候，如果控制中断是否被打开的寄存器控制标志位被打开，可能会出现中断被误触发的情况，而这个中断如果没有与之相对应的中断服务程序的话程序就会跑飞，所以单片机上电的时候一般会自动将这些寄存器里面的标志位都关掉，以免误触发。

中断服务程序是单片机程序的一部分，具体内容由开发人员决定，这样中断服务程序的大小在单片机程序中的位置就不固定，当单片机的中断被触发后，单片机需要知道中断服务程序在什么位置才能执行它，单片机通过中断跳转表（中断向量表）来解决这个问题。

虽然中断服务程序的大小和在整个程序中的位置会不固定，但程序只要被烧进单片机系统，对于这个程序来说其中断服务程序的大小和在整个程序中的位置就会被固定下来，如果对单片机程序空间分配我们做出一些约定，将一个绝对固定地址专门分配给中断使用，程序编译时会将中断服务程序的起始地址（或者是跳转到中断服务程序的指令）填到这个绝对固定地址所在的空间，当中断产生时候，单片机先将绝对固定地址所在位置里面的内容读出，根据所读内容就可以跳转到中断服务程序。

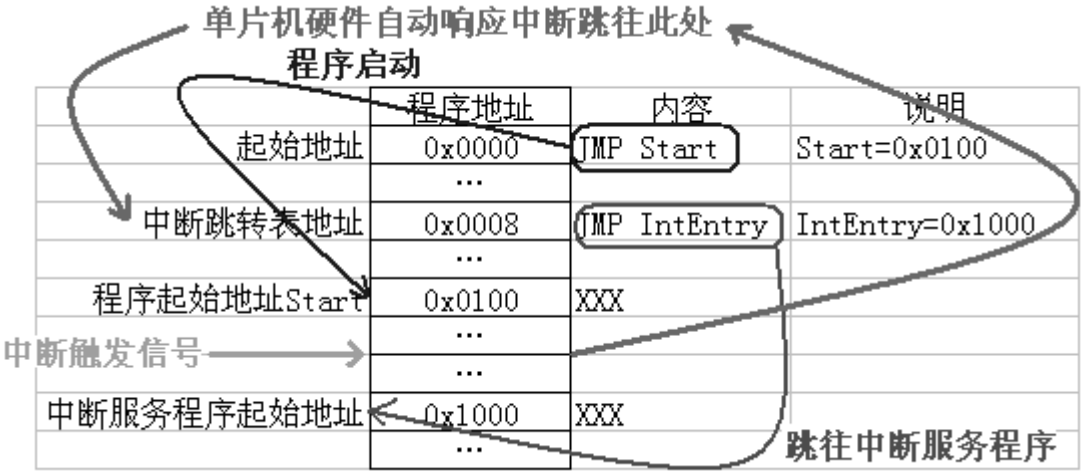


图 1.10.-7　中断响应示意图

简单的单片机所提供的中断种类有限，为了简化程序，会给每一个中断分配一个用来存放中断服务程序地址的地址空间，这种方法其实没什么不好的地方，只是单片机技术发展到现在遇到了瓶颈，高端单片机越来越复杂，于是一些专业厂商开始合作共享技术资源，例如 ARM 公司利用他们在 CPU 架构体系上的技术优势专门给另外的厂商提供 CPU 内核，另外的厂商在 ARM 内核的 CPU 外围增加功能模块，这些功能模块大都支持中断。

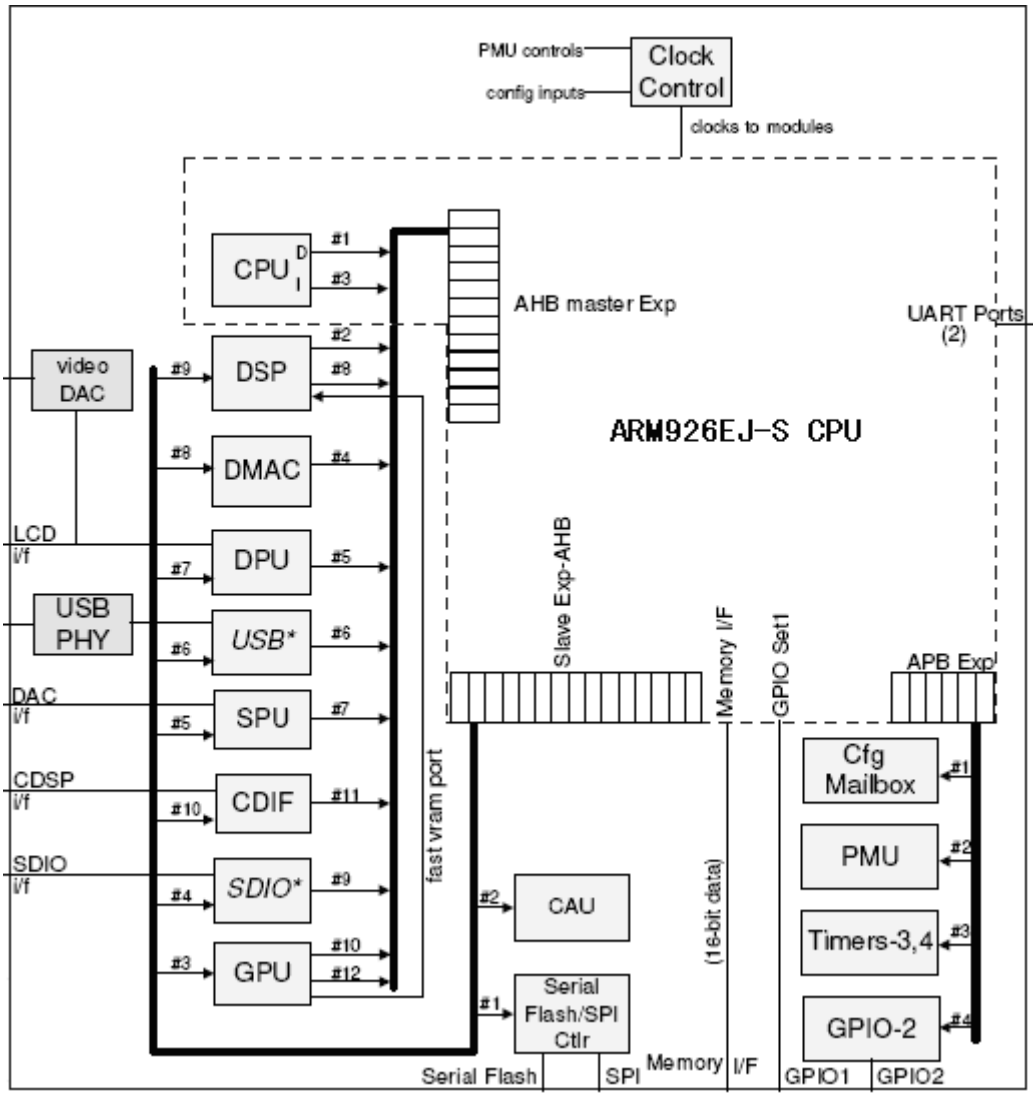


图 1.10.-8 ARM 内核单片机架构图

不同厂家在相同 CPU 内核基础上设计出来的单片机外围的功能模块会各不相同，从而中断的种类和个数也各不相同，而 CPU 处理中断的方法是一样的，如果延续简单的单片机给每个中断都分配一个地址空间的做法显然有问题，CPU 无法知道到底有多少种中断需要支持，这些中断又分别对应什么模块，于是采用另外一种中断处理方法，将所有中断地址都指向同一个，并将所有中断依次编号，中断产生时候 CPU 会告诉中断服务程序当前中断编号是多少，然后中断服务程序根据中断编号做出相应响应。

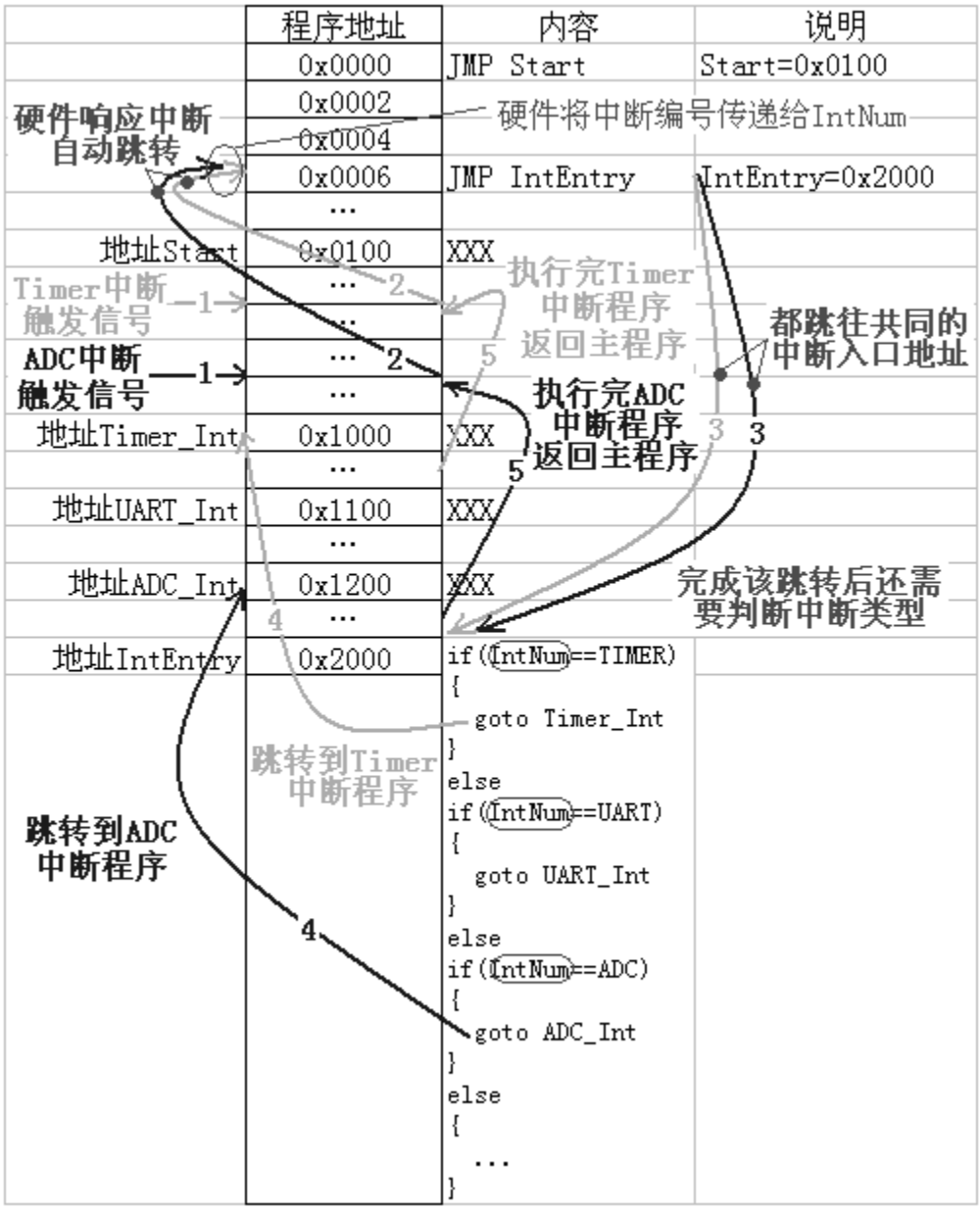


图 1.10.-9　公用中断入口中断响应流程图

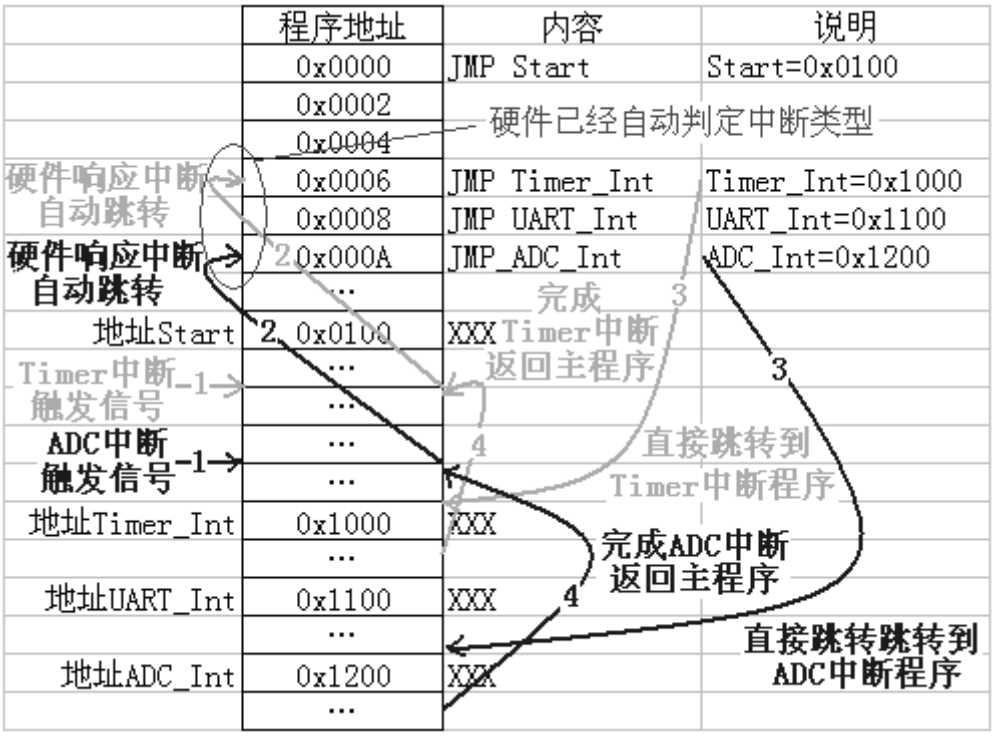


图 1.10.-10　独立中断入口中断响应流程图

所有中断使用同一个中断向量地址然后通过中断号判断中断类别的方法虽然解决了通用 CPU 内核中断不能直接对应中断向量地址的问题，但把它中断处理的流程和具有独立中断向量表的单片机相比就会发现中断的响应速度会变慢。具有独立中断向量表的单片机只要一条跳转指令就可以直接进入中断程序，而没有独立中断向量表的单片机需要先跳转到中断公共入口，然后通过代码判定中断类别，确定中断类别后才跳转到真正的中断程序中去。C 语言的代码会让这种情况更加恶化，所以如果是没有独立中断向量表的单片机一般采用汇编查表的方法加快中断响应速度。

	地址	内容	说明
IntEntry	0x2000	JMP IntEntry+IntNum*2+2	
	0x2002	JMP Timer_Int	IntNum=0
	0x2004	JMP UART_Int	IntNum=1
	0x2006	JMP ADC_Int	IntNum=2
	0x2008	XXX	
	…		
假定TIMER=0,UART=1,ADC=3,…			

图 1.10.-11　汇编中断快速跳转表

中断程序执行完毕后回返回继续执行主程序，这样就要求中断不改变主程序的运行状态，所以中断响应时需要将程序当前运行的状态信息保存起来，比如程序运行到什么位置、当前 CPU 状态寄存器的状态等信息。当中断程序执行完毕，可以通过这些信息将 CPU 状态寄存器恢复原来状态，并

能返回原程序继续执行。不同的单片机对此的处理方式也会有不同，一种是完全由硬件来完成，并不需要程序来进行管理；另外一种是将状态信息用相应指令保存在特定位置，返回时再用相应指令恢复原来状态。

单片机中断还有中断优先级和中断嵌套的概念，但不是所有的单片机都会支持这两种功能。中断优先级是不同的中断会有不同的优先级别，如果同时有两个中断产生，单片机会先响应优先级高的中断。中断嵌套是指在中断响应当中又有新的中断产生，单片机可以暂停当前的中断程序执行去响应新的中断，新中断程序执行完以后在接着执行当前中断程序。一般中断嵌套是高优先级的中断可以插入低优先级中断响应程序，同级或低级的中断不能插入当前中断响应程序。

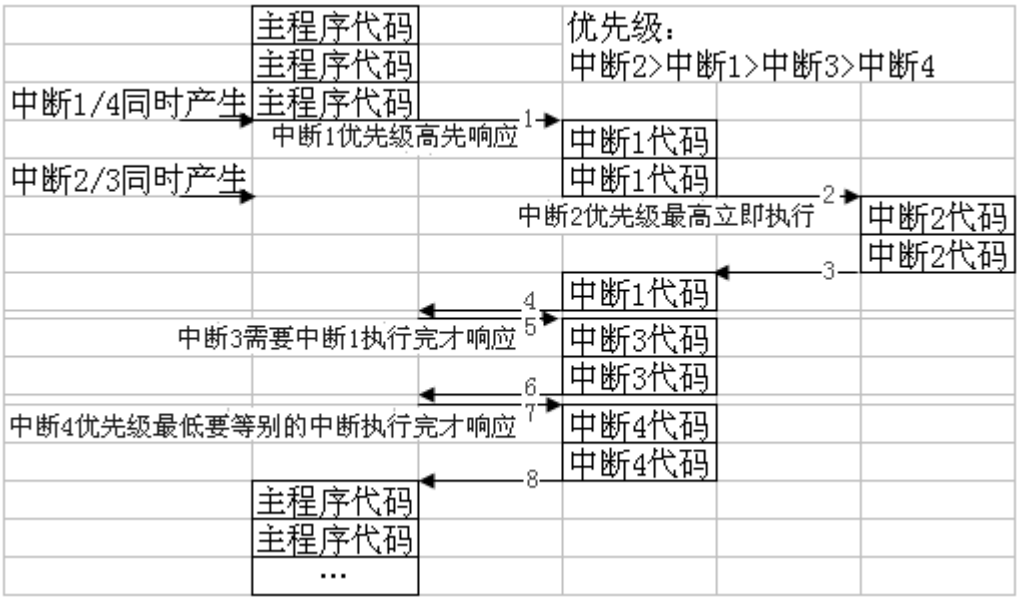


图 1.10.-12　中断嵌套示意图

中断步骤说明：

步骤①保存主程序现场，执行中断 1 服务程序

步骤②保存中断 1 服务程序现场，执行中断 2 服务程序

步骤③恢复中断 1 服务程序现场，继续执行中断 1 服务程序

步骤④恢复主程序现场，准备继续执行主程序，有新中断不能继续执行主程序

步骤⑤保存主程序现场，执行中断 3 服务程序

步骤⑥恢复主程序现场，准备继续执行主程序，有新中断不能继续执行主程序

步骤⑦保存主程序现场，执行中断 4 服务程序

步骤⑧恢复主程序现场，无中断产生继续执行主程序

有的单片机一进入中断函数就会自动将中断的总控制位关掉，需要开发人员在中断程序中用程序再次打开，否则一次中断后所有的中断就不能继续使用。对于中断标志位，在写单片机程序的时

候要依据单片机文档进行清除标志为操作，不然有可能会一旦产生某个中断就会连续不停的反复响应这个中断，导致主程序不能继续运行。

1.11. 函数和堆栈

要想用单片机来实现自己的想法，就需要编程，函数是让程序简洁规范的一个方法。应该不需要解释什么是函数了吧。刚开始接触程序编写的人可能对函数的优点没有很明显的感受，随着所写代码的增多，一定能逐渐感受到函数给程序员所带来的便利。

函数也常常被称呼为子程序，先来看看在一段程序中执行函数的流程。

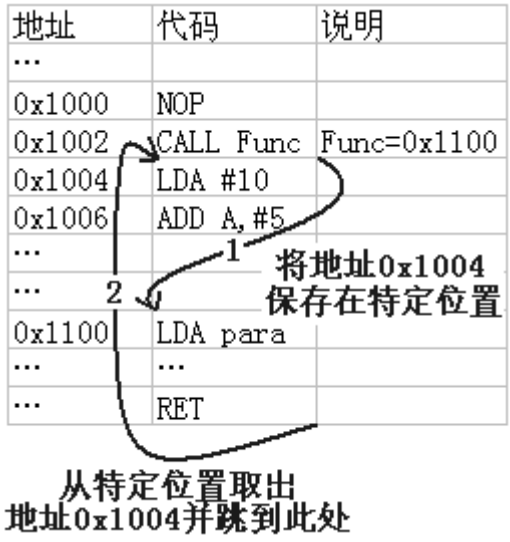


图 1.11.-1　函数执行示意图

地址为 0x1002 的指令是调用函数的指令 CALL，当程序执行完这一句后，会按①路线跳转地址 0x1100 位置，这个地址是函数 Func 入口地址，于是接下来开始执行函数 Func 的代码。当函数 Func 的代码被执行完，按函数的规定是一条用于函数返回的指令 RET，通过这条指令按②路线返回地址 0x1004。从①和②过程可以看出对于函数调用指令 CALL 和函数返回指令 RET 必须具备这样的功能：CALL 指令将自己后面指令所在的地址保存到特定的位置后转去执行自己所调用的函数，RET 指令从特定位置取出刚才保存的地址并返回到这个地址。

这个特定位置叫做**堆栈**，**堆栈**可以是在 RAM 中，也可以是硬件专门开辟的一块空间，对堆栈的操作存在一般情况下是存放和取出是一一对应的，就好比有一个一端开口的管子，当我们需要存放东西的时候就把东西从管子口塞进去，需要取东西同样也只能从管子口去拿，取出来的次序和放进去的次序刚好相反。看一个函数里面有调用其它函数的流程，从这个流程我们可以看出堆栈操作的次序关系。

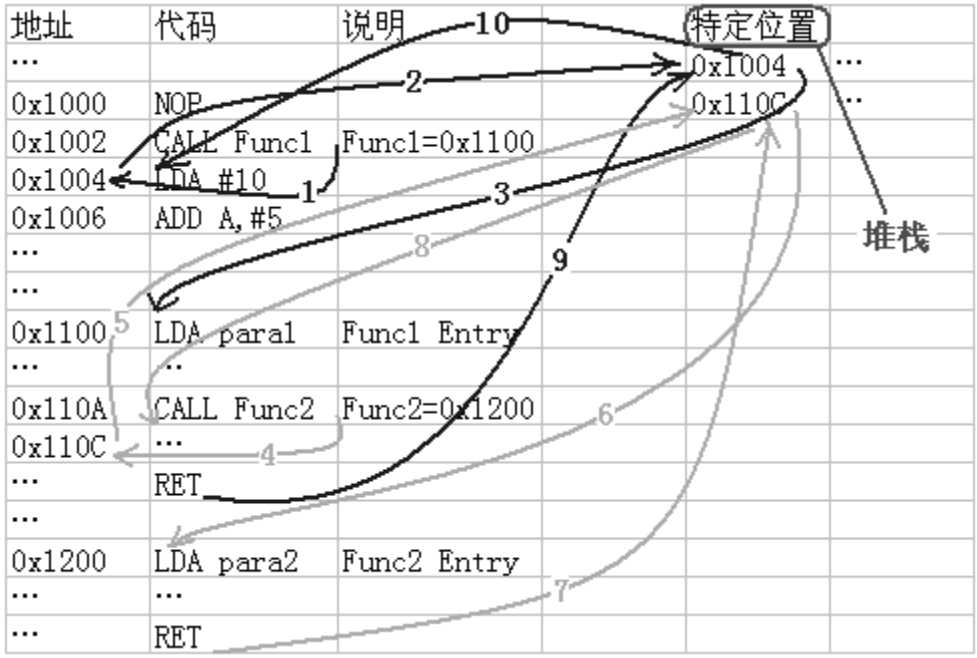


图 1.11.-2　调用函数堆栈存取示意图

步骤①②③是调用函数 Func1，把下一条指令所在地址 0x1004 和相关状态信息保存到堆栈，然后跳转到 Func1 所在地址开始执行 Func1 的代码。

函数 Func1 中包含有调用函数 Func2 的代码，当程序执行到这里时候，重复类似上面的操作。

步骤④⑤⑥会调用函数 Func2，把下一条指令所在地址 0x110C 和相关状态信息也保存在堆栈中，然后跳转到 Func2 所在地址开始执行 Func2 的代码。

函数 Func2 代码执行完毕，步骤⑦⑧从堆栈中去读取先前保存的信息，由于地址 0x110C 是后保存进去的，所以这次取到是地址 0x110C，正好是调用函数 Func2 所保存的内容，正确返回到 0x110C 位置继续执行 Func1 的其它代码。

步骤⑨⑩也到堆栈中去读取先前保存的信息，由于步骤⑦⑧已经取走 0x110C，这次取得 0x1004，同样可以正确返回到主程序中去。

如果在函数中使用一些特殊代码，会导致函数返回的位置发生变化。

……（详见完整版）

不管什么单片机，其能提供的堆栈空间是有限的，假如现在我们做这样的操作：写出许多函数 FuncN(),Func1()里面会调用 Func2()，Func2()里面会调用 Func3()，依次类推。这样所调用的函数层数每深入一层，堆栈就会被多占用一定的空间，最后结果一定是爆掉堆栈空间，函数无法按原路可靠的逐层返回。所以在编写单片机程序时候，要留意单片机最多能支持函数可以嵌套多少层，以免嵌套过多而让系统崩溃。

图 1.11.-5　函数堆栈溢出示意图

中断服务程序也是函数，和普通函数不一样的地方是中断函数在中断产生时由硬件启动调用，而普通函数是由程序员自己编写的代码来启动调用，由于中断随时都可能会产生，对单片机程序员来说难以预知其被调用的时间和位置，所以中断时单片机需要保存的信息比普通函数要多，这样两者返回使用的指令大多数时候会不一样。

中断服务函数里面如果需要调用其它函数，最好是将这些函数做为只供中断专用的函数，否则会出现主程序和中断同时需要调用同一个函数的可能，如果单片机不支持函数的重载，此时主程序调用函数的结果就可能出错，一定要留意这点，许多有经验的工程师都会忽视这个风险。

函数 Func 实现两个数相乘的功能，x/y 为输入参数，z 为输出参数，z=x*y。

```
Func:
  z=x*y                          ;完成 z=x*y 操作
  ret z                          ;将 z 的内容返回
中断函数：
  …
  x=val1                         ;此时 val1 为 10
  y=val2                         ;此时 val2 为 30
  CALL Func z=x*y                ;执行完此操作后中断产生
  …
  reti                           ;从中断返回
主程序：
x=val3                           ;此时 val3 为 20
y=val4                           ;此时 val4 为 50
```

```
CALL Func  z=x*y                    ;执行完此操作后 z 为 1000，中断产生执行中断函数
                …
                x=val1              ;此时 val1 为 10
                y=val2              ;此时 val2 为 30
                CALL Func  z=x*y    ;执行完此操作后 z 为 300
                           ret z
                …
                reti                ;从中断返回
           ret z                    ;这里将 z 错误返回为 300 而不是 1000
if(z>500) display(“val3*val4>500”)      ;正确应该显示是大于 500
else display(“val3*val4<= 500”)         ;实际结果错误显示为小于等于 500
```

1.12. 单片机 PAGE/BANK 概念

PAGE/BANK 一般只出现一些非常简单的单片机中，4 位单片机比较常见，少数 8 位单片机也有，使用 PAGE/BANK 的目的就是让这些简单的单片机能够使用更大的 RAM 和 ROM 空间。PAGE 一般是用在 ROM 上，而 BANK 则一般用在 RAM 上，两者起的作用基本一致，后面为简单起见只用 BANK 来进行说明。

还是用一款使用了 BANK 的单片机来做讲解。

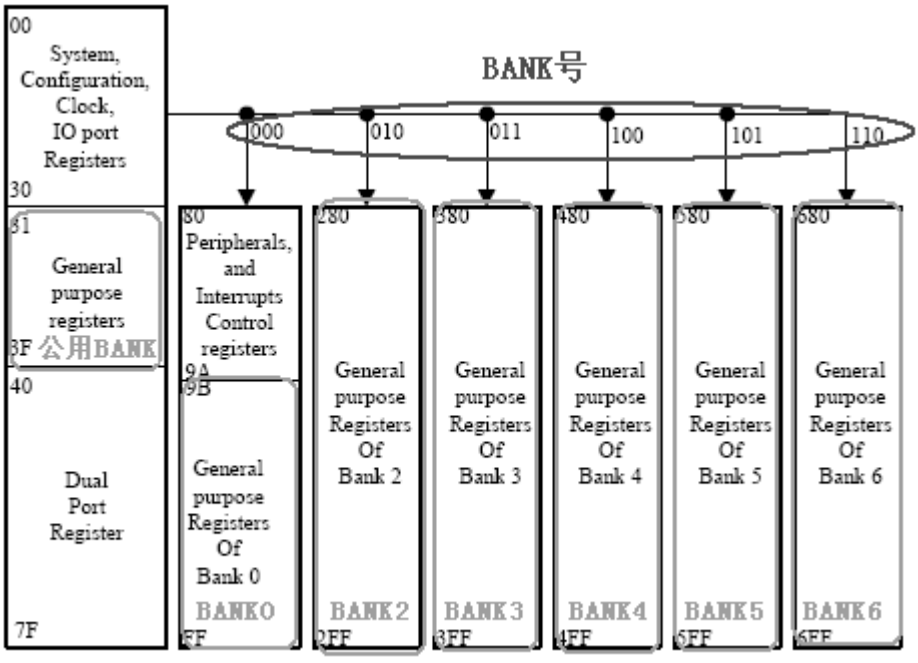


Fig. 6-2 of Data Memory (RAM) Configuration

图 1.12.-1 BANK 空间分配示意图

从这款单片机我们可以看出它的 RAM（寄存器）寻址空间是 256 字节，这 256 字节空间已经被

单片机系统占用了一部分做系统配置，但这款单片机通过 BANK 的方法可以让用户使用空间要大许多的RAM（寄存器）。

单片机会提供一个寄存器来让用户选择当前使用哪一个BANK。

7.1.5 RAM Bank Selector – RAMBS0 (0x04), and RAMBS1 (0x07)

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	-	-	RAMBSX2	RAMBSX1	RAMBSX0

As depicted in Fig. 6-2, there are seven available banks in the MCU. Each of them have 128 registers and can be accessed by defining the bits, RAMBSX0 ~ RAMBSX2, as shown below.

RAMBSX (0x04/0x07)	Bank
000	0
010	2
011	3
100	4
101	5
110	6

图 1.12.-2 BANK 控制寄存器图

再找出两条指令，一条是选择BANK 的“**BANK #k**”，另外一条是清寄存器的“**CLR r**”。

1010	1110	0000	0kkk	BANK #k	R4(RAMBS0) ← k (0~6)	None	1
1010	1111	rrrr	rrrr	CLR r	r ← 0	Z	1

BANK #k 选择BANK k

CLR r 将寄存器r(地址为r的RAM)清0

图 1.12.-3 BANK 操作指令图

观察“**CLR r**”指令的机器代码，前 8 位为操作类型，后面 8 位为可变的操作地址，这样操作地址范围为 0x00~xFF，也就是说这条指令只支持最大 256 字节的空间，无法访问地址超过 0xFF 的寄存器（RAM）空间。如果不采用 BANK 方法，这款单片机实际上只有公用 BANK 和 BANK0 所在的 116 个字节空间可以给用户使用。采用 BANK 方法后情况大为改观，将 BANK 技术将地址在 0x80~0xFF 的空间并行起来，每次选用一块用做地址为 0x80~0xFF 的空间，BANK 寄存器设为多少表示当前使用多少号 BANK（想像成内部电路有电子开关来切换这些 BANK），这样用户就可以使用 BANK2~6 提供另外提供的 640 个字节。

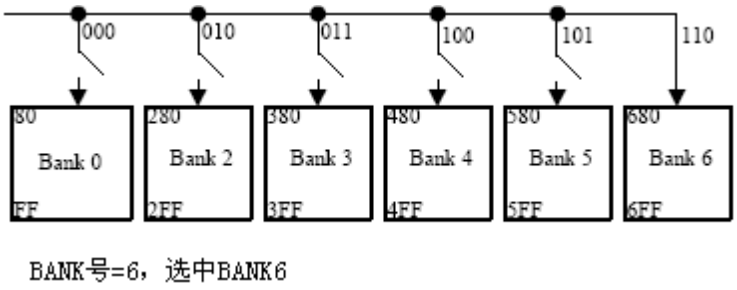


图 1.12.-4　BANK 选择示意图

① BANK　#2　　;选择 BANK2

② CLR　0x80　　;清 0x80 寄存器，当前为 BANK2，实际操作为清 BANK2 的第一字节

③ CLR　0x31　　;清 0x31 寄存器，为公用寄存器范围，清公用 BANK 第一字节

④ BANK　#4　　;选择 BANK4

⑤ CLR　0x80　　;清 0x80 寄存器，当前为 BANK4，实际操作为清 BANK4 的第一字节

⑥ CLR　0x31　　;清 0x31 寄存器，为公用寄存器范围，清公用 BANK 第一字节

代码行②⑤相同，但执行结果因为选不同 BANK 从而操作对象不一样，执行结果也自然不一样。代码行③⑥因为操作对象在公用寄存器，虽然执行这两条代码时的 BANK 不一样，但 BANK 不影响其操作对象所以结果一样。

BANK 虽然让单片机可以访问的空间加大，但使程序复杂性会增加，用户在使用容易将不同 BANK 的寄存器用混，最稳妥的方式是每次读写寄存器时都在前面加上设置 BANK 的指令，这样做的缺点是代码效率会降低，代码占用空间也会增大，对于接触单片机时间不长的人我是建议先用这种稳妥的方法。

1.13. CISC 与 RISC

CISC 和 RISC 对应中文意思是复杂指令集和精简指令集，两者主要区别是 CISC 提供大量的功能和对应指令以尽可能的用硬件性能直接满足用户需求，一般 CISC 指令集都有几百条指令，而 RISC 刚好相反，只是提供一些基本通用的功能和指令给用户，以求简单高效，让用户用软件来实现他们的不同需求，一般 RISC 指令集只有几十条指令。

从计算机技术开始诞生的那一天起，设计控制器集成电路的工程技术人员就一直就有这样一个梦想，自己设计的芯片最好能满足用户所有的需求，虽然他们自己也清楚这个想法实际上不可能实现，但他们还是尽可能的在这个方向努力。一旦发现有新的需求，这些硬件设计人员就会通过硬件来直接满足这些需求，这样就使得硬件越来越复杂，造价也越来越高。硬件平台每增加一种特殊功能，就会有相应的指令来让用户使用这种特殊功能，最后使得硬件对应的指令种类非常庞大，这就

是CISC。

家用电脑所用的CPU是CISC，从家用电脑CPU的发展历史我们就可以看到芯片设计人员想让自己设计的硬件直接满足用户所有需求这一愿望有多强烈。在家用电脑还没进入奔腾时代的时候，家用电脑还只能提供一些基本的办公需求，随着数字娱乐技术的兴起，人们发现家用电脑在数字娱乐方面的前景无限，局但限于当时技术水准，CPU 速度相对有限，还不能直接满足用户在数字媒介方面的娱乐需求，于是可以让人们进行用电脑看VCD这类娱乐活动的解压卡出现了。

设计CPU的技术人员很清楚，只要假以时日，当CPU速度够快的时候，就可以利用软件来满足人们的这种娱乐需求。他们没有去等待CPU速度能够快的那一天，而是直接在CPU 内部加入了MMX硬件功能和相应指令，也就是专门进行多媒体处理而设计的硬件电路，用户只要通过这些指令就可以直接进行一些多媒体数字信号处理的操作。

MMX 的出现让靠解压卡吃饭的厂商不得不另寻谋生之路，五年后，CPU 的速度已经非常快，家用电脑CPU在数字娱乐方面对MMX的依赖程度不再那么重要，十年后，CPU 的速度已经远超出当时人们所能想象的程度，人们对MMX这个词汇开始陌生。

在 CISC 技术日新月异的同时，一部分更具创新意识的技术人员跳出了用硬件来直接满足用户需求的思维局限。这些技术人员凭直觉感觉到 CISC 的不足是如同堆积木，会让硬件电路越来越复杂，越来越难以实现，而且永远也不可能把所有的想法都能实现。另外实际中有许多产品并不需要这么复杂的结构，尤其是单片机领域，简单高效是其首选。于是他们开始探索是否有另外一条道路，技术人员通过统计分析发现，对于CISC的各种指令，一样满足二八定律，程序中80%的指令都被包含在CISC 指令集中20%常用指令部分。实际中最常用到的指令就是存取数据、加减运算这些操作。技术人员将这些使用频率高的指令提取出来，指令种类的简化可以让技术人员在性能高效方面做出更多的改进，比如从 CISC 的指令不等长变等长指令、取指令和取操作数操作同时进行等，形成了相对CISC 简单高效的RISC。

简单价廉的 RISC 一样也有不足，功能的简化使得在应用的时候不适合做复杂的工作，大部分时候还是局限在实现逻辑控制这个范围。不过随着技术的发展，CISC 和 RISC 又逐渐在相互融合，新的电脑CPU 也开始采用RISC架构，在接受到CISC指令后再分解成RISC 指令执行。

RISC和CISC的区别：

①RISC 指令系统较小，种类的数量较少，只提供简单指令。CISC 指令系统大，种类的数量多，提供各种指令。

②RISC 指令长度、寻址方式、格式都整齐划一，这样可以充分利用流水线，基本上可实现一个时钟脉冲执行一条指令。CISC指令长度、寻址方式、格式不一，难通过流水线方式提升指令执行效率，无法做到一个时钟脉冲执行一条指令。

③RISC 的函数调用将现场状况保存在专用寄存器中来提升效率，参数也使用寄存器传递。CISC的函数调用一般通过堆栈保存现场，需要内存操作，效率要低。

1.14. 为什么 DSP 跑得快

提到 DSP，大多数人第一印象是它的运行速度比单片机要快许多，同样都是大规模半导体集成电路，为什么 DSP 就能比单片机快许多呢？这要通过 DSP 和单片机内部的结构体系来找答案。

计算机鼻祖冯·诺依曼最早提出了“数字计算机的数制采用二进制，计算机应该按照程序顺序执行”的现代计算机体系结构理论，这种理论强调的是顺序概念，凡是遵从这种理论而实现的计算机结构我们都称为冯·诺依曼结构，从最初的计算机模型到现在的电脑 CPU 都一直是采用此结构体系，而单片机是基于计算机技术基础在七十年代产生的，自然也采用冯·诺依曼结构。

我们知道冯·诺依曼结构里面存储数据的地址空间是独立唯一的，程序和数据共用地址空间，两者的地址不可以重复，并且程序和数据共用同一组地址总线和数据总线，这样读取程序代码和数据就不可以同时进行，必须分开。一些简单的单片机并不完全遵从冯·诺依曼结构，为了让系统构架更简单，ROM 和功能寄存器（RAM）的地址各自独立从 0 开始，通过指令来区分是对 ROM 还是对 RAM 的操作。

存储器的实现方法很简单，就是一个矩阵。当对存储器进行读操作时，地址总线控制输入进行选择，被选中的存储单元将自己的内容输出到数据总线上。当对存储器进行写操作时，同样由地址总线控制输入进行选择，同时将数据总线上的内容输入到被选中的存储单元中。

现在假定一个八位单片机要执行这样一个操作，将 RAM 中的一个数加 n，依照冯·诺依曼结构实现步骤可以如下。

a.地址总线指向代码指令字位置从数据总线上取出代码指令字部分。

b.地址总线指向代码操作数位置从数据总线上取出代码操作数 n。

c.依照指令系统查知指令字为将 RAM 中的某个数据加上操作数 n。

d.地址总线指向 RAM 数据位置从数据总线上取出数据内容。

e.将取到的数据和操作数 n 相加。

f.地址总线保持指向 RAM 数据位置不变将相加的结果存回 RAM 中。

对于这类操作冯·诺依曼结构的单片机处理因为要多次切换地址总线而导致效率不够高。如果连续多次进行这样的操作会将这一不足放大，在某些电子产品领域，比如无线电通讯、数字音频视频处理等，需要进行大量的数据处理，冯·诺依曼结构的单片机对于这类应用就显得力不从心，于是一种新的构架体系产生，那就是哈佛结构。

哈佛结构的处理器使用两个独立的存储器模块，分别存储指令和数据，每个存储模块都不允许指令和数据并存；使用独立的两条总线，分别作为 CPU 与每个存储器之间的专用通信路径，而这两条总线之间毫无关联。它在片内至少有 4 套总线：程序的数据总线，程序的地址总线，数据的数据

总线和数据的地址总线。这种分离的程序总线和数据总线，可允许同时获取指令字(来自程序存储器)和操作数(来自数据存储器)，而互不干扰。这意味着哈佛结构的处理器在一个机器周期内可以同时准备好指令和操作数。

现在再来看哈佛结构的处理器处理上面操作的流程。

a.程序的地址总线指向代码指令字位置从程序的数据总线上取出代码指令字，在此同时数据的地址总线指向代码操作数位置从数据的数据总线上取出代码操作数 n。

b.依照指令系统查知指令字为将 RAM 中的某个数据加上操作数 n。

c.数据的地址总线指向 RAM 数据位置从数据的数据总线上取出数据内容。

d.将取到的数据和操作数 n 相加。

e.数据的地址总线保持指向 RAM 数据位置不变将相加的结果存回 RAM 中。

这样哈佛结构的处理器就少了一次总线的切换过程，会让效率有所提升。简单的归纳一下就是冯·诺依曼结构是代码和数据串行处理，需要将总线在代码和数据中频繁切换，而哈佛结构改为并行处理，可以将代码和数据同时取得来提升效率。

哈佛结构虽然效率会有提升，但结构要复杂不少，所以价格也就比冯·诺依曼结构要贵。DSP 采用哈佛结构，但如果只是单纯将代码和数据总线独立出来的改进显然改善并不是很大，这种改善依然不能满足 DSP 的性能需求。于是 DSP 内部内部用硬件对一些数学算法进行实现，比如乘法器、硬件循环控制器等，没有乘法器的处理器实现一个乘法可能需要几十条上百条指令，而有乘法器的 DSP 则是一条指令一个周期就可以完成。

DSP 还采用指令流水线设计，传统单片机是“取指、译码、执行”三步，如果有一种设计可以让这三步能在一个触发信号周期内完成的，显然能将速度几乎提高三倍，指令流水线设计做到了这一点。流水线的原理是将串行依次操作用并行方式方式来提高速度，在执行一条指令的同时，对下一条指令译码，并取得再下一条指令。通过下图你可以看出，传统的单片机设计执行完三条指令需要九个时钟触发，而流水线设计则只需要三个时钟触发就可以完成。现在已经有不少单片机也采用流水线设计方法，不过真正的流水线实现起来比我所说的要复杂许多，比如指令 n 是跳转指令，执行完跳转指令 n 后程序不是接着执行指令 n+1，就需要找出其它应对方法。

传统	触发次序		流水线	
取指n-1	1	取指n-1	译码n-2	执行n-3
译码n-1	2	取指n	译码n-1	执行n-2
执行n-1	3	取指n+1	译码n	执行n-1
取指n	4	取指n+2	译码n+1	执行n
译码n	5	取指n+3	译码n-2	执行n+1
执行n	6			
取指n+1	7			
译码n+1	8			
执行n+1	9			

图 1.14.-1　流水线示意图

这样就得到开始所提问题的答案，DSP 是专门针对数字处理做出的设计，在进行数字信号处理时会比冯·诺依曼结构快许多，但如果将 DSP 程序只是去实现一些简单的逻辑循环控制时其并不一定会比冯·诺依曼结构快多少。

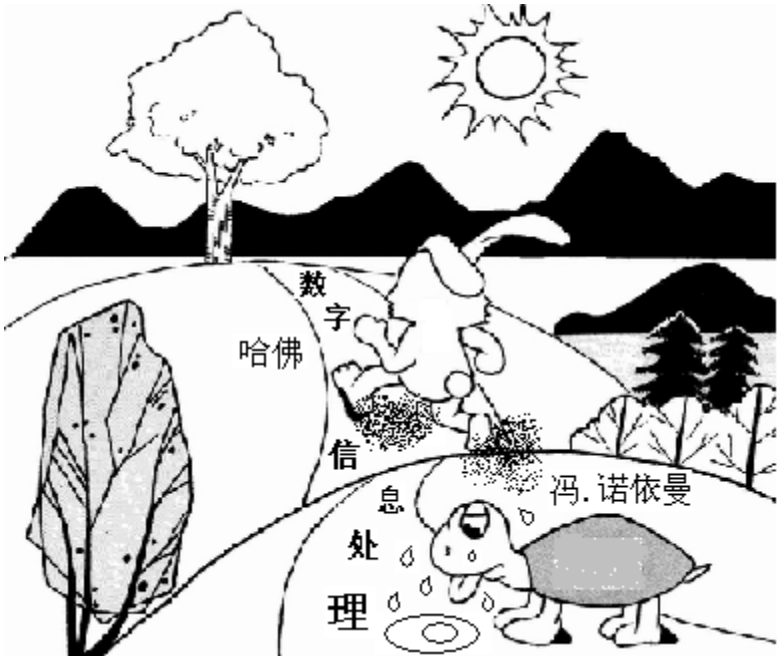


图 1.14.-2　哈佛和冯·诺依曼

不是所有的单片机都是冯·诺依曼结构，在移动数字通讯刚刚兴起的年代，DSP 那是望尽春色，近年来随着技术的发展单片机的速度是越跑越快，有一些单片机也开始采用哈佛结构，少数单片机内部甚至会加上一个小的 DSP 核，使得 DSP 在数字处理速度方面的优势逐步变小，加上一些专用器件内部采用硬内核进行数据处理，这些变化恐怕让 DSP 难以再现昔日辉煌。

1.15. 单片机产品开发常见用语

单片机应用开发过程中会遇到许多名词，有些甚至是英文缩写，这些名词会让刚接触单片机开

发工作的新人觉得头疼。虽然现在网络已经非常普及，网络资源也非常丰富，遇到这些缩写只要上网一搜就能得到相关解释，但不少公司对上网做了限制，上班的时候不一定可以自由上网进行搜索，这里我将一些常见的缩写列出来并加上注解。

将一些生产有关的用语也列出来是希望可以让不了解生产的人能对生产有一个初步的了解，知道生产大概是怎么进行的，了解生产对开发产品会有不小的帮助。

MCU/CPU

MCU 现在基本可以直接理解为单片机，而 CPU 主要用于电脑。

RAM/ROM

RAM/ROM 前面有一个章节做了专门阐述，这里不再重复。

EPROM/EEPROM/FLASH

这些都可以当做 ROM，现在还有一些新类型的器件也起到同样功能，但都还没有大量市场化，这里不做详述。

EPROM 出现最早，早期的单片机大多采用外挂 ROM 放置程序，这样就要给开发人员提供一种可以烧写程序的器件，断电后都还能保持烧写内容不变，EPROM 就是这种器件，能重复擦写，烧写程序时候先用紫外线将程序擦除干净，然后将程序烧写进去，EPROM 很好辨认，背上有个玻璃窗口，现在已经很少见了。

EEPROM 的出现让 EPROM 逐渐淡出历史舞台，这种器件和 EPROM 相比不需要紫外线擦除，开发人员用起来就更方便，EPROM 用紫外线擦除很慢，而 EEPROM 电擦除快了许多。

FLASH 采用更新的技术，可以实现大容量、高速度等特性，需要接口连线少有 SPI FLASH，放程序有 NOR FLASH（速度快、价格贵、容量相对较小），放数据有 NAND FLASH（容量大、价格低）。

MIPS

不要和一种叫 MIPS 的单片机架构名称混淆，这里是指用来衡量单片机速度的一个单位名称，我们知道单片机跑多快最直观的是指令周期，这个单位说单片机一秒内能跑多少条指令，1MIPS 就是一秒可以跑一百万条指令，如果是单周期指令，也就是每条指令耗时一微秒。

DICE

常见芯片是黑色的矩形外壳，四周或者下面会引出许多金属脚，这种芯片就是标准封装。标准封装已经规定好引脚的位置和大小，这样进行产品开发时候只要知道用的是什么封装就知道电路板应该怎么布线。

DICE 也叫 CHIP 或裸片，是没有封装的芯片，这种芯片需要通过一种叫 BONDING（邦定）的方

法将芯片上的引脚用非常细的线连出来，再点上胶直接固定在电路板上。DICE 没有封装所以价格比带封装的价格要便宜，所以如果一个产品生产的量非常之大，就会节省不少成本。另外有的时候担心产品的程序被人复制，这样比用标准封装程序被复制的难度要大许多，标准封装只能依赖单片机自身的防复制功能和把上面的丝印磨掉不让别人知道是什么型号的单片机来进行。

OTP/MTP/掩膜

程序放到外接的存储器会增大产品空间，如果将程序放在单片机内部显然可以降低成本，单片机厂家都支持这种做法。

OTP 是单片机自身带有一块可以编程一次的程序存储空间，产品开发好以后可以利用专用工具将程序写到这片空间里面去，但只能写一次，如果出错则无法修改。这种做法有一个很严重的缺限，如果产品生产出来后发现有 BUG，那么已经生产出来的产品就无法修正错误，MTP 可以解决这个问题，可以多次编程，为了防止误操作，会在编程时要求给某个管脚一个特殊电压。

无论是 OTP 还是 MTP，对于生产量特别大的产品都会带来麻烦，比如年产量超过百万台的产品，工厂烧写程序就需要耗费相当多的人力和设备，单片机厂商提供的掩膜服务可以省去这个烦恼，将程序交给单片机厂商，他们把程序直接固化在芯片内部。这样做不是一样会出现 OTP 那样发现 BUG 无法解决的问题吗？有应对方法，任何产品生产都是先小批量，后大批量的顺序进行，可以先用 MTP 或 OTP 小批量生产，经过这个阶段后一般问题都已经暴露出来，修正错误后再去掩膜就会安全许多。

不管什么方法，产品生产出来后才发现 BUG 都是不好的，即便是用 MTP，也会非常麻烦，需要将市场上还没销售的产品再运回工厂，然后开肠破肚重新烧写程序，会浪费巨大的人力财力，并影响自己产品的市场形象。好的做法是在开发设计的时候通过严谨的设计、完善的测试让 BUG 在生产前就全部暴露，不让其出现在真正产品中。

丝印

丝印是在产品表面印刷的图案或文字，没有什么特殊的含义。

DE/RD

DE 一般是指产品开发部门，也就是进行产品开发的部门，里面会包含电子、结构、外观等小组，产品开发流程大致如下。

根据市场提出产品概念→开发部门进行可行性分析→成本和市场价格预估→产品确认立项→开发部门进行开发→质量部门对产品进行测试→生产部门进行生产→市场部门推向市场

RD 也叫 R&D，同样也属于产品开发部门，但和 DE 却有着明显不同，这个部门对应的中文叫预研，主要负责产品的前期技术可行性研究，可以把这个部门当成是公司里面的科研院所。这个部门常常会尝试应用一些新技术，因为新技术的不成熟产品失败的几率要比真正的产品开发部门高许多，加上即使技术上能实现但还有原器件供货、成本等因素也可能导致不能真正产品化，所以这个

部门实际上是一些规模公司才玩得起的烧钱部门。

RD 一度在国内很流行，因为会给外人搞新技术的部门那肯定很厉害的感觉，就是在公司内部这个部门的员工都容易自认为高人一等。近年来厂家对产品和市场的重新定位，不再迷信技术至上，尤其在国内，市场还是被那些技术成熟价格合理的产品所占据，这样 RD 这个部门的角色变得比较尴尬，芯片供应商甚至都怕和 RD 打交道，知道他们的具体产品往往是遥遥无期。

不过有些规模有限的公司为了提升自己在外面的技术能力形象，现在还在借用 RD 一词，但实际工作内容都是传统 DE 所做的。

FAE

FAE 是供应商的技术支持工程师，由他们向开发人员解答技术方面遇到的问题。

EMC/EMI

这两个词是电磁兼容，也就是抗电磁干扰和减少自己对外的电磁干扰，如何实现有一套现成的方法，主要是通过添加特殊元器件和改良电路板步线来实现。不同的地区会依照自己的情况要求在其境内销售的产品必须能通过比如 FCC 这样一些认证测试， SGS 就是一家能提供国际认可报告的认证测试公司。目前国内市场消费电子产品虽然有相应标准要求执行，但实际情况是许多公司都没有在这方面做相应考虑。

PE/QE/QC/QA

一个产品被生产出来除了要有开发部门进行设计，还需要工厂里面许多其它部门的相互协作，这些缩写都是和产品生产密切相关的一些工作岗位。

PE 是生产工程师，属于生产部门。开发部门一般只是做出少数样板并对产品进技术参数确认，这些样板的制造不需要工人都可以完成。真正的产品生产则不一样，是由工人来生产装配，工人的技能素质远不如开发部门的工程师，PE 的作用在这里就得到充分体现，他们定制工艺流程文件来告诉工人如何做，这个工艺流程文件要细致到一条线是怎么连接、连这条线需要几秒钟这样的程度，这样工人不需要了解产品任何相关知识，只要照工艺流程文件操作就能把产品生产出来。

一个好的生产工程师除了能定出高效的工艺流程外，还要能从生产的角度给开发部提出一些建设性的意见，比如生产中发现一个产品的某个元件容易产生不良，而这个元件在另外一个产品生产中不会出现此问题，生产工程师就可以分析两个产品电路的不同，推测可能的原因并反馈给开发部门确认改进。在实际情况中有些规模不大的工厂没有将 PE 独立出来，由 DE 一并负责。

QC 也属于生产部门，产品被工人生产出来，不代表所有的产品功能都正常。工人操作出错、生产过程中元器件被损坏、元器件自身有问题或备料错误都有可能导致产品功能不正常，这样在产品下线之前，生产部门的 QC 会对每一个产品进行功能测试，只有功能正常的产品才被允许下线。QC 并不是只在最后对产品全面功能测试，如果这样做会把生产问题全堆积到最后，不利问题的发现与

解决，好的做法是在生产流程中安置一定 QC 位进行目测、部分电路功能测试这样的工作，这样就可以把部分问题提前发现并及时做出处理。QC 还可以细分为 IQC 和 OQC，IQC 负责来料检，对采购的元器件进行质量控制，一般是 1%抽检，OQC 则是产品出厂全检。

QE 也是上了规模的工厂才会设置的岗位，在工厂对某些器件进行寿命测试或产品自动功能测试时，需要做出一些用于测试的工具，比如某个按键，供应商承诺的寿命是 5000 次，现在要从来料中抽出十个进行寿命测试，如果是人来按 5000 下，让你做你愿意吗？QE 就会被要求设计出一个能自动进行测试的工具来完成这 5000 次按下的动作。

QA 属于质量部门，QC 虽然也进行功能测试，但只是简单看一下基本功能是否正常，有一些问题隐藏很深，QC 是很难发现的。比如硬件某些元件在长时间工作时会出现失效的现象，这种问题不可能让 QC 在生产中进行检测，如果这么做那生产别进行了，这种情况 QA 会抽样进行老化实验。软件上更容易发生此类问题，正常的操作都正常，但当进行了某种特殊次序的操作后，就有可能出现因为软件不完善而出错的情况，QA 对样机以用户的角度进行长时间高强度组合测试来减少或避免这类问题的发生。

SMT/回流焊/波峰焊

这几个都是生产焊接设备的名字，SMT 是自动贴片机，适合高速生产，设备昂贵，对操作员也有一定的技能要求；回流焊是为了让小工厂焊接贴片元件的一种低成本方法，先做一张钢网，这个钢网上面在焊接贴片元件的焊盘位置挖出一样大小的小懂，让后将电路板和钢网放到一个类似油印机的架子上，将锡浆刷到电路板焊盘位置，再人工将元器放上去，经过回流焊设备就完成了焊接；波封焊那就是更简单的生产设备，用于插接件的焊接，这个设备实际上就是一个大锡炉，插好元件的电路板从上面经过就将裸露的焊盘焊好，后面再用电锯一样的剪脚机把元件的管脚剪短。

啤机/开模

啤（bie）机我不清楚这个词是怎么来的，就是注塑机，电子产品大都需要塑料壳，所以这个设备也是电子厂重要的生产设备，一般由负责结构的来和它打交道。

注塑机生产塑料件就是将高温融化后的塑料粒注入相应模具，然后冷却成形。制作模具的过程叫开模，以前因为结构设计软件不发达，不能在电脑里面直接三维显示，所以模具的制作复杂而且昂贵，往往需要多次修改才能让模具效果比较理想，所以耗费的时间也相当长，现在有功能强大的三维软件辅助，已经变容易许多。金属壳体常采用模具冲压而成。

第二章　单片机应用小技巧

进入本章，我想你已经具备了基本的单片机功底，最基本的要求是指可以用某种单片机进行一些简单程序开发。通过本章内容的学习，一定会让你在产品开发方面的思维得到一些启迪，当你看完本章后不妨回过头去看看自己以前的产品或程序，如果你很容易就从以前的程序或产品中找出自己之前存在的不足，那恭喜你，再做两个项目你就可以向老板要求加薪。

本章内容大都是以实际工作经验为基础总结而得，内容多少不一，有的章节可能颇费纸墨，有的却可能只是寥寥数语，存在这种差异的原因是有些例子技巧性主要体现在实现的细节方面，而有的却只要找到方法就算成功。

2.1. 用 IO 模拟接口

有时选用的单片机并不提供外围器件所需的接口，这时可以用 IO 来模拟所需接口，只要 IO 口能满足接口规定的时序，就能用 IO 模拟的接口来和外围器件进行通讯。

用 IO 口模拟接口的方法对于大家我相信是一点就明，但要使 IO 口模拟的接口工作更加可靠稳定并不简单，往往需要在一些细节上多加处理才能做好，接下来我会通过用 IO 模拟 UART 和 I2C 来告诉大家，应该通过哪些细节展现你的技术功底。

IO 模拟 UART

模拟 UART 非常简单，一条 IO 模拟发送的 TX，一条 IO 模拟接收的 RX，另外将地 GND 引出就可以实现 UART 功能。在硬件上基本不用考虑太多，只需要注意 IO 口上下拉电阻的选择，如果 IO 口内部可以选择设置上下拉电阻，必须设为上拉电阻，如果 IO 口不提供内部上下拉电阻控制最好在外部连上 10k~51k 的上拉电阻。有了上拉电阻，就可以确保 TX 能可靠输出高低电平，RX 即使没有和另外的设备相连也能保证读到的状态是 1，这样是为了和 UART 通讯时序中用 1 来表示空闲的要求一致。

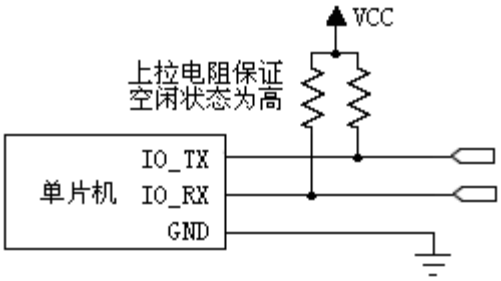


图 2.1.-1　IO 模拟 UART 示意图

要用 IO 软件模拟 UART，就需要用软件在 IO_TX 脚输出满足 UART 通讯时序的波形，还能检测出 IO_RX 脚上的波形是否与 UART 通讯时序一致并将数据正确读回。UART 可以设置成多种工作状态，限于篇幅这里只选用最常见的“9600/8/N/1”设置进行讲述。

“9600/8/N/1”表示波特率为 9600，这个速率收发一个位大约耗时 104us，8 位数据位，无校验位，1 位停止位。

IO_TX 的控制比较简单，先将对应 IO 设置成输出，然后输出 1 表示当前没有数据发送。当需要发送数据的时候，先输出一个 104us 宽的低电平做为起始位 0，然后按 104us/位的宽度按照先低位后高位的顺序依次输出所发数据的各个位，最后将输出 104us 宽的高电平做为停止位 1。这样一个字节的发送过程就全部完成，如果还有数据需要发送，按同样的方法操作即可。

IO_TX 发送过程最关键的地方是保证每个位宽为 104us。最简单的方法是用代码实现延时，在发送过程中最好关闭所有中断以保证延时准确。如果不想去数代码有多少周期也可以用定时中断来实现，让单片机产生一个 104us 的定时中断，然后在中断程序被调用后的同一时刻依次输出所有位，这个定时中断需要最高的优先级，否则其它中断会导致时间不准。

IO_RX 的控制要复杂一些，将对应 IO 设置成输入，然后需要让程序不停的检测 IO_RX 上有没有收到 0，一旦检测到 0 则表示一个数据开始传送，需要启动接收过程。接收程序最好是 IO_RX 刚从 1 变为 0 就能立刻检测到，这样才能保证接收过程 104us 间隔的时间基点准确。

检测数据开始传送的方法基本上为这三种：

① IO_RX 支持负跳变触发中断用中断检测。，

② 程序用不超过 52us 的定时中断程序定时检测。

③ 程序在主程序中循环检测。

这三种方法中负跳变中断的方法最好，时间基点可以控制得非常准，后两种方法时间基点误差相对都比较大。

检测到 IO_RX 从 1 变为 0 后就需要严格按照通讯时序来读取数据的各个位，我个人认为最好的方法如下：

①在检测到数据开始传送后 26us/52us/78us 三个点读 IO_RX 状态，要求这三点必须全为 0，否则错误退出。

②然后在 52+104*N us 位置读得 8 个数据位。

③再在 104*9+26us/52us/78us 三个点读 IO_RX 状态，要求这三点必须全为 1，否则错误退出。

●注：计算时间需要将中断响应时间考虑进去

不管是 IO_TX 还是 IO_RX，实际上都很难准确无误的做到 104us 的延时间隔，如果延时和绝对时间两者间误差达到一定限度时，就会出错，这里示意了 IO_RX 延时不够大的情况。

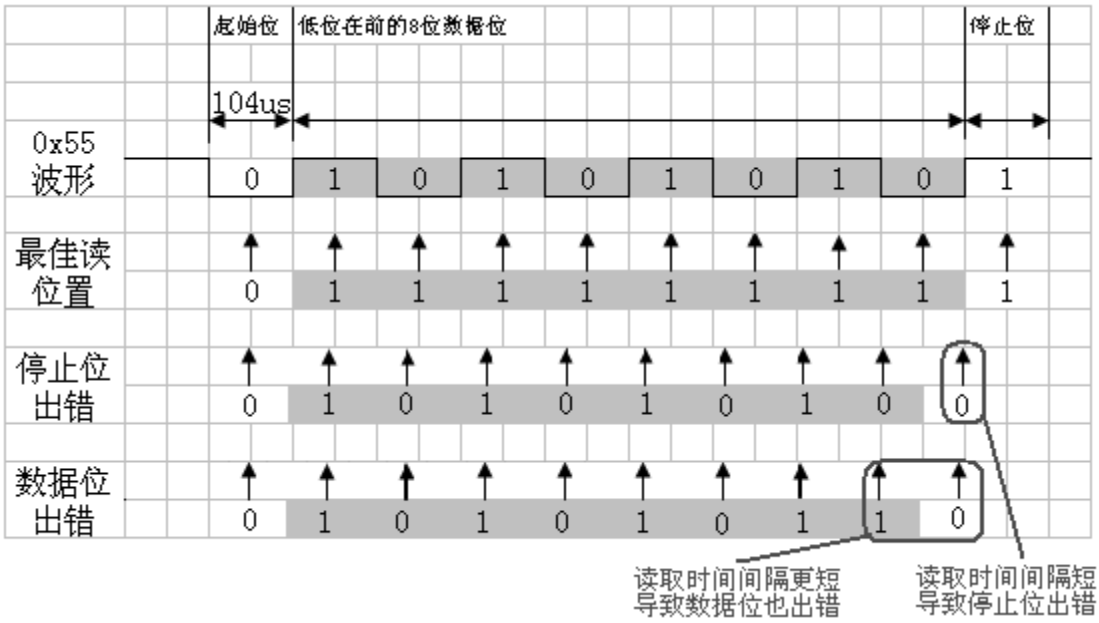


图 2.1.-2 UART 读数据位置示意图

那为什么①和③需要做一个 26us/52us/78us 的特殊处理呢？来看看我们发送 0xFF 时候的波形，这个波形很简单，就是一个宽度为 104us 的负脉冲。以 104us 间隔去读数据，我们可以正确读回 0xFF，但果以 52us 的间隔去读，我们会读到一个 0xFE。所以认为能读到数据就万事大吉，所读到的结果并不一定正确。反过来也一样，如果发送方改为以 4800 波特率（208us 间隔）发送 0xFF，接收方以 9600 波特率接收会误读到 0xFE。

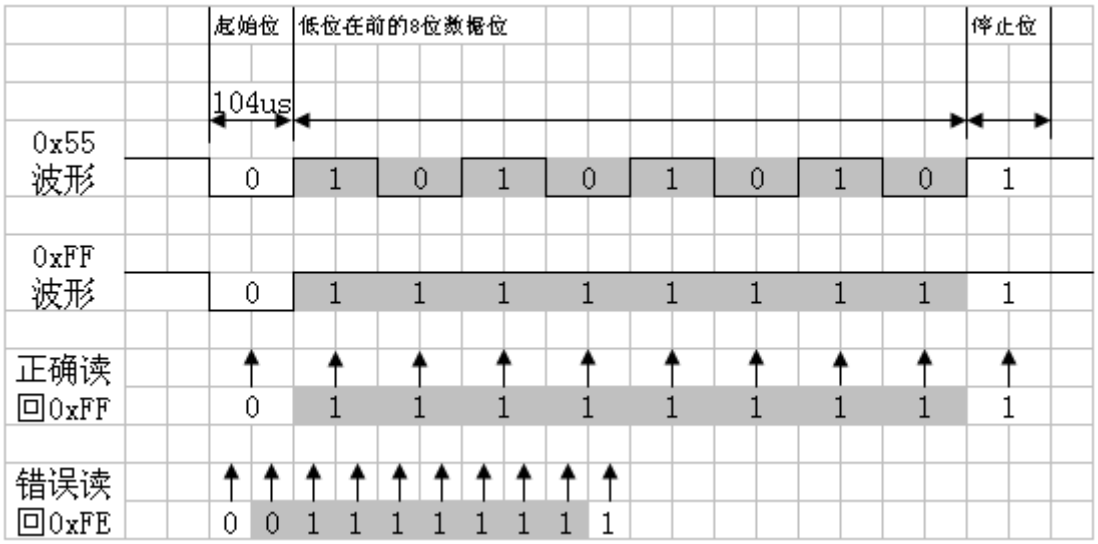


图 2.1.-3 UART 读数据出错示意图

①和③的特殊处理可以避免刚才所说的错误发生，这个处理是对起始位和结束位的宽度进行检

测，可以避免收发方波特率不一致产生的误接收。那为什么只在 26us/52us/78us 三点处理而不是在整个宽度内尽可能多次的判断呢？是因为我们日常应用中波特率不随意定的，前人已经选择了一些常用的波特率做为标准，这些波特率是 300/600/1200/2400/4800/…，它们间大多数呈现两倍的关系，26us/52us/78us 三点已经能将相邻的波特率检测出来。读的次数过多的话还会带来另外一个麻烦，那就是每个设备和绝对波特率时间间隔之间或多或少都会存在一定误差，也就是波特率 9600 的基准间隔大约为 104us，实际中的设备和这个间隔都存在一定误差，误差大的甚至 103us 和 105us 都有可能出现，如果读太多的话会让这个误差允许范围变得非常小，所以不要去读太多次，留足够的间隔来容纳误差。

那到底可以接受多大的误差呢？10 个位总宽度为 1040us，如果不做 26us/52us/78us 的特殊处理，最后读停止位的时间应该是 1040-52=988us，当延时间隔偏小时我们只要保证到这个点大于 104*9=936us 就行，也就是负偏差最大可以到(936/988-1)*100%=5.2%，考虑到收发双方都会存在误差，所以能接受的误差还要除以 2 为 2.6%，实际应用中一般认为 3%以内都可以被接受。

用 IO 模拟 UART 会有一些限制：首先是对高波特率的模拟难以实现；其次是在收发数据的时候为了保证时间间隔的精准会影响其它中断的使用；另外如果想能收发同时进行（全双工）需要比较高的程序技巧。如果编程语言不是汇编而是 C，去数指令周期数会比较麻烦，如果想偷懒的话就是关掉中断用示波器将延时调准。

IO 模拟 I2C

模拟 I2C 接口也只需要两条 IO，分别模拟 SDA 和 SCL，和 UART 不同的是模拟 SCL 的 IO 根据时序图在不同时刻所设的输入输出状态会不同。

还是以 EEPROM 芯片 AT24Cxx 为例，单片机为主设备，用 IO 模拟出 IO_SDA 和 IO_SCL 来读写 AT24Cxx。硬件连接上也没有特别要注意的地方，许多提供 I2C 接口的芯片都明确指出在这两条信号线上建议加 4.7k 上拉电阻，以保证信号线在空闲状态下保持高电平。这两个上拉电阻作用非常重要，用 IO 模拟必须加上。

I2C 接口有两个重要的时序状态，就是规定 SDA/SCL 从高变低和从低变高的特殊顺序产生 START 和 STOP 信号：SDA 和 SCL 都为高然后 SDA 先变低接着 SCL 变低为 START，SDA 和 SCL 都为低然后 SCL 先变高接着 SDA 变高为 STOP。

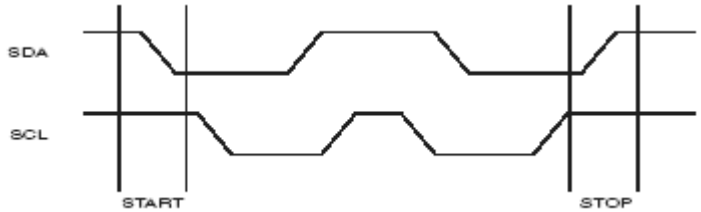


图 2.1.-4　I2C 接口 START 和 STOP 信号示意图

主设备向从设备传输一个位时先将 SDA 输出正确状态，然后 SCL 变高再变回低，SCL 的这个正脉冲使得双方完成一位的传输。

用 IO 模拟 I2C 接口来读写 AT24Cxx 时，首先要让单片机的 IO_SDA 和 IO_SCL 输出与通讯时序相同的波型，然后 IO_SDA 根据实际情况在输入输出两种状态中相互切换。来看一下对 AT24Cxx 进行读操作的时序图，IO_SDA 在整个流程中需要来回在输入和输出中切换。

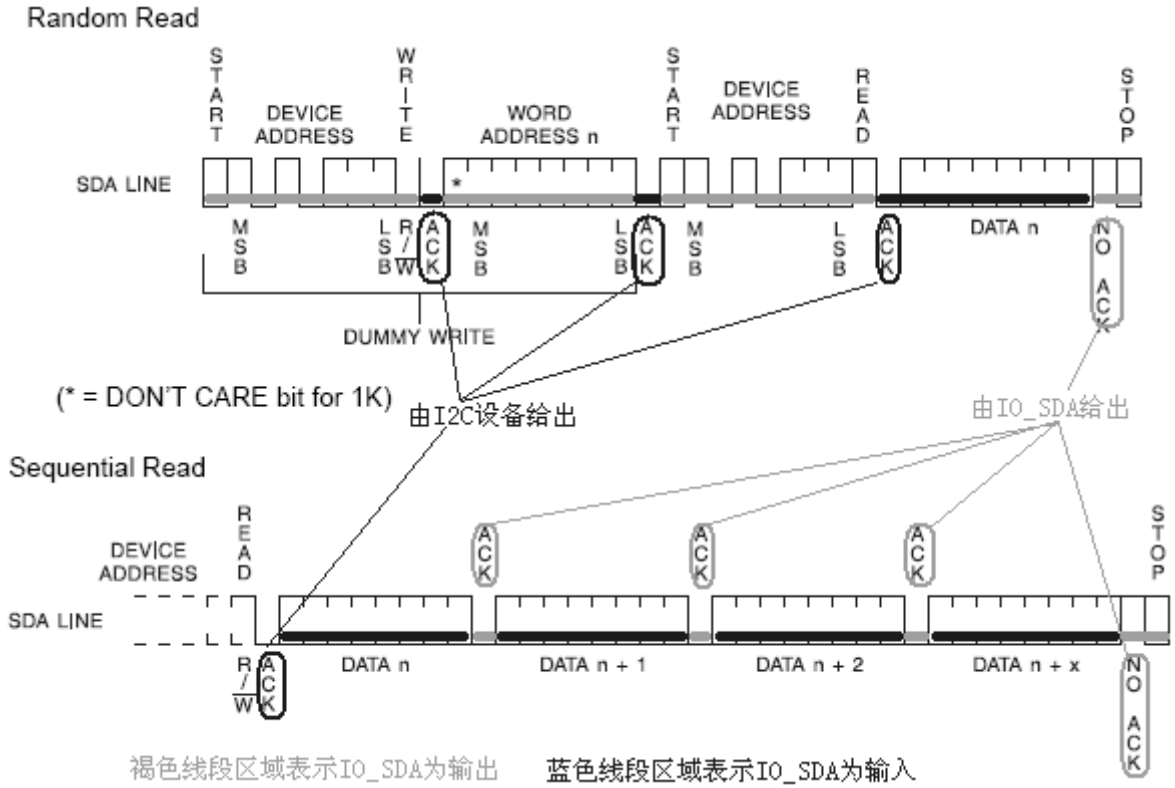


图 2.1.-5　I2C 时序图

当 AT24Cxx 给出 ACK 信号时，单片机来判断这个 ACK 信号，在蓝色框标识的 ACK 位置 IO_SDA 先要从输出转成输入，然后 IO_SCL 变高，接下来单片机去看 IO_SDA 是否与 ACK 状态一致，最后 IO_SCL 变回低 IO_SDA 转回输出。

下面是单片机对 AT24Cxx 返回的 ACK 信号进行处理的详细步骤：

①IO_SDA 从输出转成输入，此时 AT24Cxx 的 SDA 还是输入，如果没有上拉电阻，就会形成一个未知状态，有可能被识别成 STOP 信号而出错（必须加上拉电阻的原因）。

②IO_SCL 从低变高，AT24Cxx 的 SDA 输出 ACK，因为 IO_SDA 上一步已经转为输入，两者不会产生冲突。

③IO_SDA 判断 AT24Cxx 的 SDA 输出的 ACK 是否正确，IO_SCL 从高变回低，AT24Cxx 的 SDA 随即变回输入。

④IO_SDA 从输入转回输出，因为 AT24Cxx 的 SDA 上一步已经转为输入，不会产生冲突，接下来

进行下一位传输。

从前面流程可以看出 IO_SDA 输入输出转换需要严格遵循通讯时序，否则就有可能出错或者形成两边都是输出相互打架的局面，这是和 UART 接口最大的不同之处。如果对时序不能完全肯定，可以在主从设备的 SDA 间串联一个 100Ω 左右的电阻以起到保护作用。(其它接口也可以根据实际情况添加这样的保护电阻)

图 2.1.-6　IO 模拟 I2C 示意图

在以往的工作经历当中，我发现不少人在写 IO 模拟 I2C 程序的时候会出现一些疏漏，他们所写的程序正常运行都不会发生任何问题，但在长时间运行当中有时候会出现 I2C 设备突然不再响应主机命令的情况。检查他们写的程序发现大都是在 ACK 判断的地方存在问题，当他们检查到 ACK 不对时，会错误退出，这时他们只记得返回错误信息，而忘记 IO 给出 STOP 信号，导致从设备没有终止当前操作以释放 I2C 总线。当他们所写的程序进行下一步操作时，从设备会因无法解析时序而不知道如何响应命令，这样主机的新操作还会失败。

即便是程序完全正确，也有可能出现 ACK 不对的情况，比如外界的干扰信号扰乱了原本正确的通讯时序就会导致 ACK 不对，如何让 IO 模拟 I2C 工作更稳定可靠，两点建议。

①在 IO 对 I2C 进行操作函数一开始先让 IO_SDA 和 IO_SCL 产生一个 STOP 信号，因为 I2C 设备一般都默认任何时刻只要有 STOP 信号产生就会立刻退出并释放掉 I2C 总线，如果之前 I2C 设备出错，这个操作会让其释放 I2C 总线。

②别忘记在 ACK 不对这类错误退出之前产生出一个 STOP 信号。

2.2.　交流特性显神通

触摸屏还不便宜的时候，人们为了实现手指触摸功能想了许多方法，红外线就是其中一种。我所在的公司也尝试用红外线来实现手指触摸，我们公司的产品并不需要太高的精度，当时好象是只要做到每个区域大约手指头大小的 8*8 矩阵就可以满足应用要求，这里以 3*4 矩阵为例。

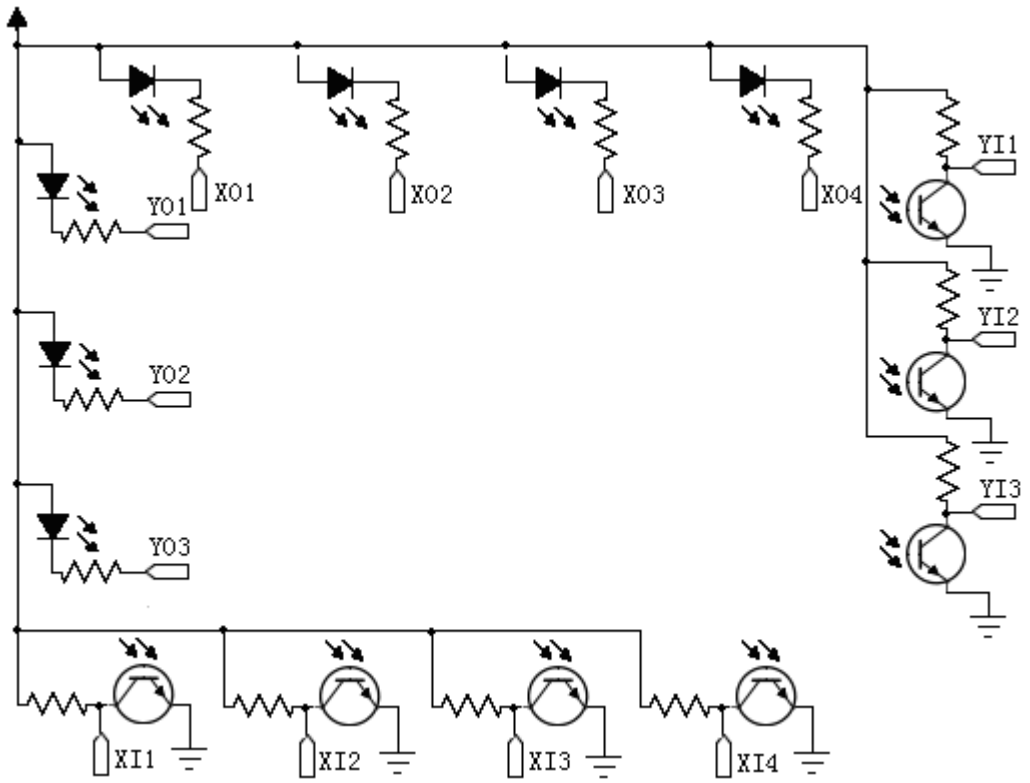


图 2.2.-1　红外矩阵示意图

光敏三极管的电流和所感应到光强度成正比，红外 LED 发光照在光敏三极管上，流过光敏三极管的电流大，如果有手指等物体挡在红外 LED 和光敏三极管之间，红外线就会被阻挡，流过光敏三极管的电流小，电流大小的变化通过电阻以电压形式表现出来，这样就可以用光敏三极管输出电压的高低来判定是否有手指。

采用类似扫描键盘的方法图示矩阵可以检测 3*4=12 点。

硬件电路和程序很快完成，开发阶段功能也都正常，好象已经满足设计要求，然而就在产品准备生产的时候，意外情况发生，有人发现晴天在窗户附近手指怎么点都没反映。那时候我就在烧钱的 R&D 部门，所以被叫过去救急。

原因很快就分析出来，晴天室外的光照强度太大，虽然产品有用深色的塑料片来过滤可见光，但晴天室外的可见光和红外线强度实在是太大，就是加了深色的塑料片也还能让光敏三极管饱和，无法再体现由程序控制的红外 LED 的亮灭引起的变化，从而失效。用示波器测量也验证这一分析结果，接下来是要想解决方法，总不能说取消产品吧。

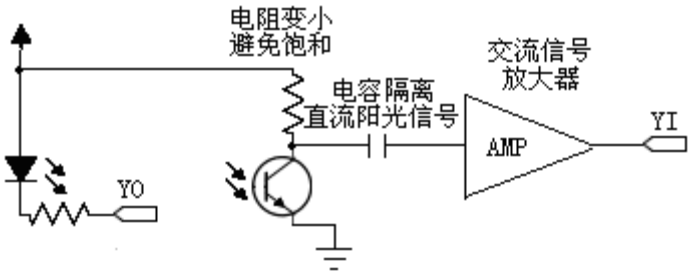


图 2.2.-2　红外信号处理示意图

要解决此问题就要找出一个可以将程序控制的红外 LED 信号和阳光分离的方法，阳光我们可以看成一个幅度很大的直流信号，如果红外 LED 信号是交流信号，那就很容易从阳光中分离出来。改变硬件电路，将光敏三极管的输出串接一个电容，这样阳光产生的直流信号就被电容阻隔住，当程序控制红外 LED 点亮时，光敏三极管会因红外 LED 从灭到亮的变化而产生一个跳变的交流信号，这个信号可以传过串接的电容，然后由程序对这个交流信号进行判别。

单做这一点改动还不能完全解决问题，在阳光下光敏三极管已经工作在饱和状态，电气特性就已经不能体现红外 LED 产生的变化。

光敏三极管是电流性器件，导通电流和感应的光强度成正比，做这样的简化假定：

光敏三极管电流 I，光照强度ɣ，I=K*ɣ

光敏三极管外接电阻 R，电源电压 U，光敏三极管压降 Uce，U=R*I+Uce=R*K*ɣ+Uce

当ɣ大到一定程度后 R*K*ɣ≈U，Uce 接近为 0，光敏三极管饱和，其电流 I 不能继续随光照强度ɣ变大而变大。我们需要避免阳光下出现饱和状态，只能是减小 R 或 K，简单起见选择减小电阻 R。

当红外 LED 被点亮时，光强度会产生一个△ɣ，对应有电压变化△U=R*K*△ɣ，△U 可以通过电容，但为了防止阳光下产生饱和这个电阻变得非常小，所以△U 也相当小，不能直接被单片机处理，所以我们用一个放大电路来放大△U，到这里已经可以输出单片机程序想要的 YI 了。

验证电路效果不错，即便是中午在室外测试也能稳定工作。既然做了改动，就要看看有没有其它方面需要进行完善。放大电路相对成本比较高，如果每一路都用独立的放大电路显然不合算，可以将不同光敏三极管的输出用电容并联在放大器输入端，这样所用通道就可以共用一个放大器，所增加的成本就会变小。因为是对交流信号进行放大处理，其它灯光的闪烁可能会造成干扰，所以程序需要增加一些抗干扰措施。

2.3.　电阻网络低成本高速 AD

不少人都有这样一个观点，就是在学校书本上的东西基本上都没什么用，我不大赞同这种说法，学校里面学的大都是理论基础，要直接用到实际工作中确实比较难，但许多时候将这些理论基础做

一定延伸往往能找出解决问题的方法，前面用交直流的原理解决了红外线在室外饱和的问题，这里给一个基于电路理论实现低成本 AD 的例子。

……（详见完整版）

2.4. 利用电容充放电测电阻

电容充放电符合下面公式：

$$U_t = U_0 + (U_1 - U_0) * (1 - e^{-\frac{t}{RC}})$$

U_0 电容上初始电压

U_1 电容最终能达到电压

假定初始电压 U_0 为 0

$$U_t = U_1 * (1 - e^{-\frac{t}{RC}})$$

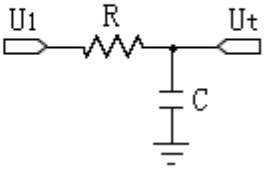


图 2.4.-1 电阻电容充放电示意图

如果 Ut 和 U1 恒定，对于初始电压为 0 的情况有：t=RC*Ln(U1/(U1-Ut))

也就是当 Ut 和 U1 选用恒定的值，对于相同的电容 C，充电时间 t 和电阻 R 大小成线性正比关系 t=K*R，比例系数 K=C*Ln(U1/(U1-Ut))。

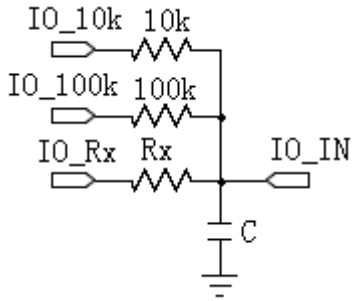


图 2.4.-2 电容效应测电阻示意图

测量流程如下：

①IO_IN 设为输入，IO_10k、IO_100k 和 IO_Rx 设为输出并输出 0，等待一段时间后将 IO_IN 也改为输出 0 一段时间，确保电容 C 放电充分。

②IO_IN、IO_100k 和 IO_Rx 设为输入，IO_10k 输出 1，单片机开始计时，当 IO_IN 检测到 1 时

候计时停止，这个时间 T10 为 10k 大小参考电阻充电时间。

③IO_IN 设为输入，IO_10k、IO_100k 和 IO_Rx 设为输出并输出 0，等待一段时间后将 IO_IN 也改为输出 0 一段时间，确保电容 C 放电充分。

④IO_IN、IO_10k 和 IO_Rx 设为输入，IO_100k 输出 1，单片机开始计时，当 IO_IN 检测到 1 时候计时停止，这个时间 T100 为 100k 大小参考电阻充电时间。

⑤IO_IN 设为输入，IO_10k、IO_100k 和 IO_Rx 设为输出并输出 0，等待一段时间后将 IO_IN 也改为输出 0 一段时间，确保电容 C 放电充分。

⑥IO_IN、IO_10k 和 IO_100k 设为输入，IO_Rx 输出 1，单片机开始计时，当 IO_IN 检测到 1 时候计时停止，这个时间 Tx 为电阻 Rx 充电时间。

虽然我们并不清楚 IO_IN 检测到 1 的具体电压（也就是 Ut）是多少，电容 C 也不容易控制误差，但是通过前面的公式我们可以将这个电压 Ut 和电容 C 约掉。

基本公式 t=RC*Ln(U1/(U1-Ut))

T10=(10k)*C* Ln(U1/(U1-Ut))　式①

T100=(100k)*C* Ln(U1/(U1-Ut))　式②

Tx=Rx*C* Ln(U1/(U1-Ut))　式③

式③与式①相除得 Tx/T10=Rx/10k　→ Rx=(10k)*Tx/T10

式③与式②相除得 Tx/T100=Rx/100k　→ Rx=(100k)*Tx/T100

已经可以测量出电阻 Rx 的大小，这种测试方法虽然可以通过比较来消除 IO_IN 检测到 1 的具体电压和电容 C 大小不一带来的误差，但还是存在一些局限，IO 输出 1 的时候电压并不完全相同，会带来一定的误差。

通过 10k/100k 两种电阻做参照档可以使测量范围加大，但单片机 IO 在输入状态下会有一个比较大的电阻，所以测量需要选用 100k 档的大电阻误差会高一些。因为接触电阻、IO 驱动能力等原因需要以 1k 为参照档的小电阻不太适合本方法。

另外软件需要对 Rx 进行是否有接电阻的特殊检测，不然当 IO_Rx 输出 1 时可能永远无法充到 IO_IN 检测到 1。

2.5. 晶振也能控制电源

曾经遇到这样一个产品，要求单片机在工作时能对一个元件供电，单片机停止工作（软关机）时关断这个电源。这个要求其实非常简单，正常情况随便用一个 IO 来控制这个电源就可以实现。问题出在当时所用的单片机身上，这个单片机非常简单便宜，能提供的 IO 口有限，当完成其它功能需求后已经没有多余的 IO 可以使用。

不用怀疑是不是真的没有多余的 IO，或者是有没有可以共用的 IO，这些问题当时已经把单片机的 IO 资源翻了许多遍都没找到。也不是没解决方法，用 74HC373 之类的锁存器进行 IO 扩展就可

以实现，但这么做会显著增加成本，是属于没有办法时才会用的办法。

望着电路图，好象还真的是没有什么办法了，忽然看到晶振，一个想法产生：能不能利用晶振来控制这个电源呢？晶振在单片机工作时可以输出一个稳定的正弦波，单片机工作时晶振停振停止输出，如果我们利用到这个特性来控制电源不就达到了目的吗？

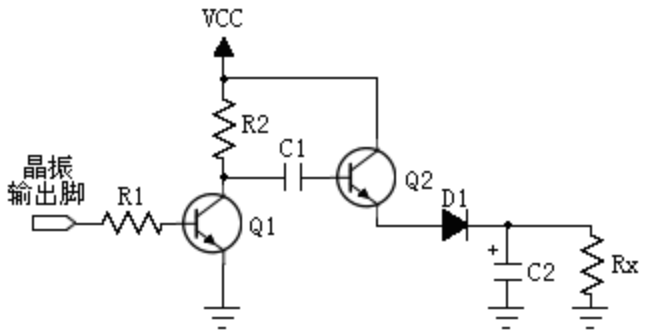


图 2.5.-1　晶振控制电源示意图

晶振输出脚在单片机工作时输出为稳定正弦波，通过电阻 R1 加在三极管 Q1 基极上，这样三极管 Q1 随着晶振正弦波周期性的通断，并在导通期间将正弦波反相放大，电容 C1 隔直流通交流的特性可以通过交流分量（又用到了交直流通断这样最基本的原理），三极管 Q2 也会周期性通断并对二极管 D1 和电容 C2 组成的充电电路充电，只要充电频率足够快，电容 C2 的电容值够大，就能向负载 Rx 提供工作电流。（和当时实际电路有差异，只供示意用）

当单片机停止工作，晶振停振，三极管 Q1 关断其集电极为高，没有交流信号电容 C1 停止导通，三极管 Q 也随之停止导通，电容 C2 上的电荷被负载 Rx 消耗完，输出电压降为 0。

2.6.　如何降低功耗

但凡提到飞利浦手机，人们第一反应那就是待机时间长，曾几何时，飞利浦几乎就是超长待机的代名词。有人说飞利浦待机时间长是其电池容量大，这只是一其待机时间长的一个因素，它以牺牲体积、重量等方面的性能来保证电池容量足够大，但这一点并不能使其待机时间比其它品牌的两倍都要长，更重要的一点是它在如何降低功耗方面下了大量功夫。

……（详见完整版）

2.7.　开机请用 NOP

单片机和自然界的其它事物会具备一些共性，一辆汽车，发动到匀速前进需要一个加速稳定的

过程，单片机也一样，上电后到它正常稳定工作也需要一段时间来稳定，只是这个时间非常之短。

单片机上电时，系统内部并不是即刻到达理想状态，晶振起振到稳定需要稳定时间，系统内部的各种逻辑电路高低电平的形成需要稳定时间，外部接口充放电过程到稳定状态需要时间等等，在这些操作没完成之前如果就让单片机执行实际工作代码，就难保证执行结果准确可靠。

一般 NOP 指令是空操作，就是 MCU 没有做什么实质性的操作，好比做了一个小小的延时等待，在所有的指令中，这条指令需要使用到的系统资源是最少的，也就是如果 MCU 真有一个不稳定的状态，执行这条指令的安全性最高。

我们是无法知道 MCU 到底什么时候会稳定下来，设计 MCU 的工程师不会让这个时间太长，一般来说等到 MCU 复位完开始执行代码基本已经稳定下来，如果还没有稳定也只会持续一个非常段的时间。如果我们程序启动一开始用上十几个 NOP，可以说 MCU 执行完这些 NOP 肯定已经稳定下来，如果还没稳定那只能说这个 MCU 设计得太烂。有的 MCU 会在复位后自动等待一段时间，这个时间就是让整个芯片稳定下来，然后才开始执行程序。

用 NOP 并没有严格的理论依据来支持，只是从经验方面做出这样的预估，我工作的的时间已经不算短，还没有遇到用不用 NOP 运行结果会不相同的实际经历，但从经验方面看这么做好处不一定能体现出来，坏处肯定是没有，所以我还是建议新人在写程序的时候在启动的位置多加几个 NOP。

2.8. 查表与乘除法

查表法是单片机程序提升执行速度的一个有效方法，尤其在进行一些运算的时候，可以显著提高速度。网上有不少如何算法加速的资料和文章供大家参考，但单片机应用程序大多数时候都只用到加减乘除这样的基本运算，这里只是用查表来实现乘法来介绍查表法的优点，并利用乘法来告诉大家如何实现除法操作。

……（详见完整版）

2.9. RAM 动态装载程序

简单的单片机，程序都是存放在 ROM 里面，现在这些单片机一般都自带有内部 ROM 来存放程序，但这部分 ROM 空间有限，空间大小可能会不能满足用户的需要，所以会支持用户在外部对存储空间进行扩展，如果单片机器支持程序存放在扩展空间，就可以实现 RAM 动态状态程序的功能。

RAM 动态装载程序是指将程序并不存放在可以直接执行的 ROM 区域，而是存放在其它存储介质里，当需要执行程序的时候，先将程序从其它存储介质读到指定的 RAM 位置，当 RAM 和所读的程序满足一定规则时就可以在 RAM 里面执行这些程序。

采用 RAM 动态装载程序有什么好处呢？这种方法只是针对某些特殊产品才能展现优势，如插卡的游戏机，我们常见的游戏卡都是一个很宽的插槽，上面有几十条线，正是因为这些线的存在，导

致游戏卡体积都比较大，实际上游戏卡内部就是一片体积很小的 ROM，根本不需要这么大的体积。这种大的游戏卡对于掌上游戏机不是一个好的选择，掌机追求的是轻巧，于是游戏卡成了轻巧化的障碍，如果能把游戏卡做到优盘那样的大小一定是件美好的事情。

今天我就用 RAM 动态装载程序的方法为大家实现这一想法，将快有手掌那么大的游戏卡做到优盘的大小。

找到一款结构可以实现 RAM 动态装载的单片机。

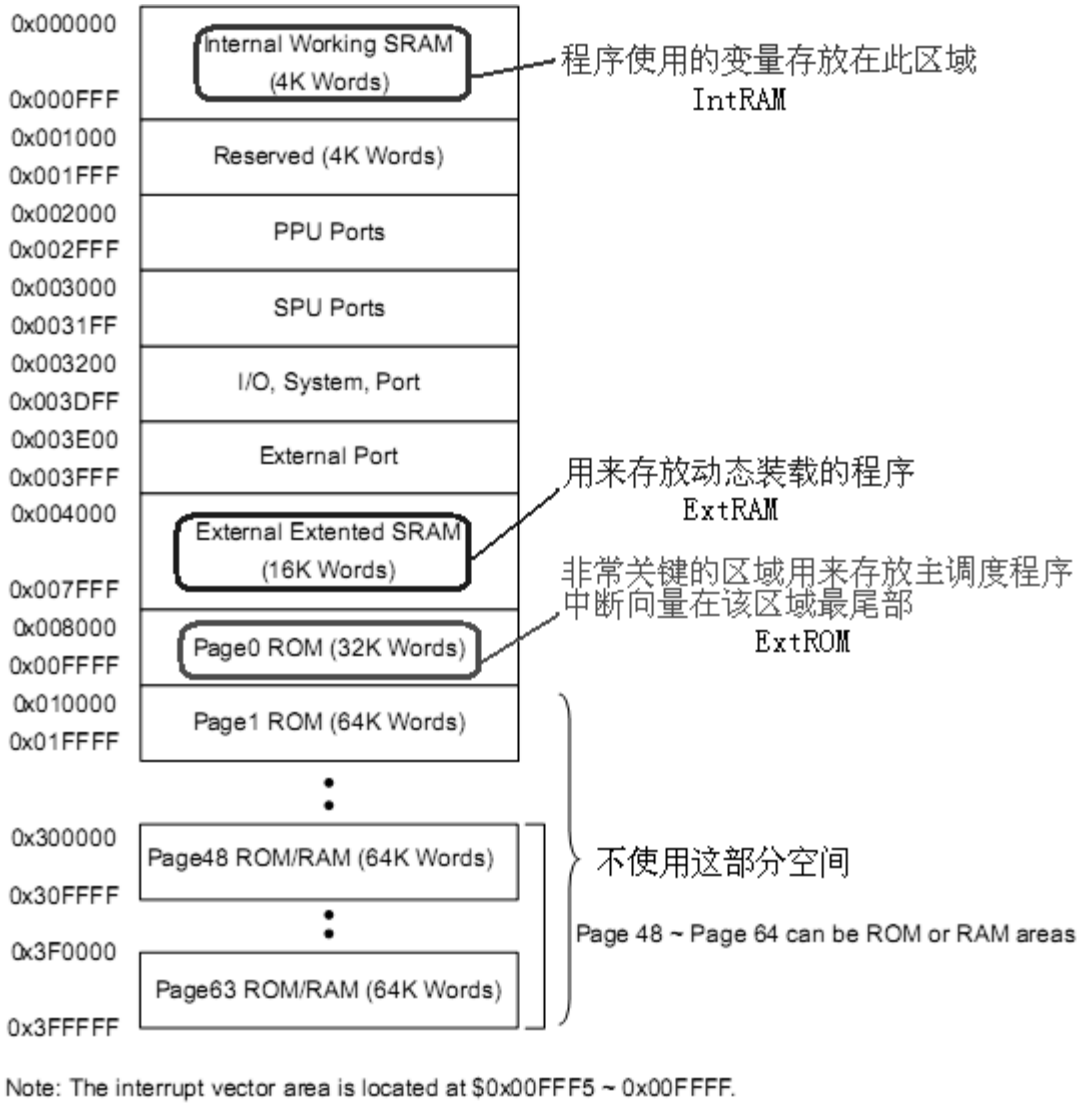


图 2.9.-1　样例 MCU 内存分布示意图

从图我们可以知道这款单片机内部自带 4k 字的 IntRAM，我们将程序中需要使用的变量放在这部分 RAM 中，外挂 16k 字的 ExtRAM 用来动态装载应用程序，另外还会外挂 32k 字的 ExtROM 存放执行装载功能的主调度程序。

要动态装载应用程序，还需要有一个地方存储应用程序，我们采用带 SPI 接口的 Flash 芯片，

因为采用 SPI 接口插槽只需要六条线，这样完全可以把外卡做到优盘的大小。

正常情况下该单片机的程序是按照下面方式编写：

```
;JMP    x              跳转到地址 x
;CALL   x              调用在地址 x 的函数
;RET                   从函数返回
;RETI                  从中断函数返回
;ORG    x              下一条代码位置从地址 x 开始

ORG 0x0000            ;定位到内部 RAM 空间，存放程序用的变量
  ……
ORG 0x4000            ;定位到外部 RAM 空间，如果没有外挂则不使用该区域
  ……
ORG 0x8000            ;定位到外部 ROM 空间，主程序存放在这部分区域
Start :               ;主程序入口
  ……
Main_Loop:            ;主程序循环地址
  ……
  CALL  Sub_Routine1  ;跳转到地址 Sub_Routine1 直到 RET 指令返回到下一行
  ……
  CALL  Sub_Routine2  ;跳转到地址 Sub_Routine2 直到 RET 指令返回到下一行
  ……
  JMP  Main_Loop      ;跳转到 Main_Loop 位置继续循环执行主程序代码
Sub_Routine1:         ;函数（子程序）1
  ……
  RET
Sub_Routine2:         ;函数（子程序）2
  ……
  RET
INT_Routine1:         ;中断服务程序 1
  ……
  RETI
INT_Routine2:         ;中断服务程序 2
  ……
  RETI
INT_Routine3:         ;中断服务程序 3
```

```
    ……
    RETI
    ……
INT_Routine10:          ;中断服务程序 10
    ……
    RETI
ORG 0xFFF5              ;定位中断向量位置
    DW INT_Routine1     ;存放中断服务程序 1 地址
    DW INT_Routine2     ;存放中断服务程序 2 地址
    DW INT_Routine3     ;存放中断服务程序 3 地址
    ……
    DW INT_Routine10    ;存放中断服务程序 10 地址
    DW Start            ;存放复位向量，跳转到主程序入口 Start
```

这里我们侧重看 CALL/JMP 指令的效果，从程序结构可以看出实际上就是向指定地址进行跳转，如果我们能够在 ExtRAM 中放有程序并能跳转到这个程序位置就可以执行该程序。通过对单片机指令和编译器的分析，可以满足 ExtRAM 中有程序这一要求。

```
……
ORG 0x4000              ;定位到外部 RAM 空间，现在里面放有程序
    ……
    JMP  Main_Loop      ;跳转到 Main_Loop 位置继续循环执行主程序代码
ExtSub_Routine1:        ;外部 RAM 中函数（子程序）1
    ……
    RET
    ……
ORG 0x8000              ;定位到外部 ROM 空间，主程序存放在这部分区域
Start :                 ;主程序入口
    ……
Main_Loop:              ;主程序循环地址
    ……
    CALL  Sub_Routine1  ;跳转到地址 Sub_Routine1 直到 RET 指令返回到下一行
    ……
    CALL  ExtSub_Routine1  ;跳转到地址 ExtSub_Routine1 直到 RET 指令返回到下一行
    ……
```

```
  ;JMP  Main_Loop          ;屏蔽掉这条指令
  JMP  0x4000              ;改为跳转到地址 0x4000
Sub_Routine1:              ;函数（子程序）1
  ……
  RET
```

改动后的程序在原来跳回 Main_Loop 的位置改跳转到 0x4000，执行完里面的代码后再跳回 Main_Loop，另外也可以在 0x8000~0xFFFF 区域中的代码调用里面的函数 ExtSub_Routine1。虽然在理论层面可以让编译器生成可以满足 ExtRAM 中放有程序的机器代码，但存放机器代码的时候存在问题，断电后 0x8000~0xFFFF 区域中的代码可以由 ExtROM 来保存，0x4000~0x7FFF 区域中的代码因为是 ExtRAM 位置，断电即丢失，显然不能直接存放在 0x4000~0x7FFF 区域。

这个问题很容易解决,我们自己将编译器生成的机器代码分成 0x4000~0x7FFF 和 0x8000~0xFFFF 两部分，把 0x8000~0xFFFF 部分直接写入 ExtROM，0x4000~0x7FFF 部分存放到 SPI Flash 中，这样处理后单片机上电后 0x4000~0x7FFF 区域里面内容空白，并没有对应程序，需要在 0x8000~0xFFFF 位置的程序中增加一段将程序从 SPI Flash 装载到 0x4000~0x7FFF 位置 ExtRAM 中的代码。

```
ORG 0x0000                 ;定位到内部 RAM 空间，存放程序用的变量
  ……
ORG 0x4000                 ;定位到外部 RAM 空间，如果没有外挂则不使用该区域
  ……
ORG 0x8000                 ;定位到外部 ROM 空间，主程序存放在这部分区域
Start :                    ;主程序入口
  ……
  CALL  Load_ExtCode       ;完成从 SPI Flash 装载代码到 ExtRAM 的功能
Main_Loop:                 ;主程序循环地址
  ……
  CALL  Sub_Routine1       ;跳转到地址 Sub_Routine1 直到 RET 指令返回到下一行
  ……
  CALL  ExtSub_Routine1    ;跳转到地址 ExtSub_Routine1 直到 RET 指令返回到下一行
  ……
  JMP  0x4000              ;跳转到地址 0x4000
Load_ExtCode:
  ……
  RET
```

现在当程序第一次运行到Main_Loop位置的时候，已经通过调用函数Load_ExtCode将SPI Flash中的代码装载ExtRAM中，当程序执行完 JMP 0x4000 指令时候就可以执行在 ExtRAM 中的代码，成功实现 RAM 装载程序并运行。

因为 ExtRAM 空间不大，如果程序比较大就需要来来回回反复装载不同的程序到 ExtRAM 中执行，这就是动态装载。要实现动态装载不难，在前面的基础上对函数 Load_ExtCode 做一个特殊约定就可以做到。

```
ORG 0x0000                  ;定位到外部 ROM 空间
……
  ExtLoadAddr               ;函数 Load_ExtCode 从 SPI Flash 装载代码的起始地址
  ExtLoadSize               ;函数 Load_ExtCode 从 SPI Flash 装载代码的大小
……
ORG 0x4000                  ;定位到外部 RAM 空间，现在里面放有程序
  ……
ORG 0x8000                  ;定位到外部 ROM 空间
  CALL  Load_ExtCode        ;在跳到 0x8000 之前已经设置好 ExtLoadAddr 和 ExtLoadSize
  JMP  0x4000               ;装载完新的代码跳回 0x4000 执行新代码
Start:                      ;主程序入口，此时并不是在 0x8000 位置
  ……
  ExtLoadAddr=0x0000        ;从 SPI Flash 地址 0x0000 开始装载代码
  ExtLoadSize=0x4000        ;装载代码大小为 0x4000
  CALL  Load_ExtCode        ;完成从 SPI Flash 装载代码到 ExtRAM 的功能
Main:                       ;主程序地址
  ……
  CALL  Sub_Routine1        ;跳转到地址 Sub_Routine1 直到 RET 指令返回到下一行
  ……
  CALL  ExtSub_Routine1     ;跳转到地址 ExtSub_Routine1 直到 RET 指令返回到下一行
  ……
  JMP  0x4000               ;跳转到地址 0x4000
Load_ExtCode:               ;依据 ExtLoadAddr 和 ExtLoadSize 进行代码装载
  ……
  RET
```

存放在 SPI Flash 中的代码数据格式：

SPI Flash 内部空间 0x0000~0x7FFF（字节为单位）

```
;程序 1
ORG 0x4000  ;定位到外部 RAM 空间，现在里面放有程序
  ……                    ;程序 1 代码
  ;接下来装载程序 2
  ExtLoadAddr=0x0000    ;从 SPI Flash 地址 0x8000 开始装载代码（字节为单位）
  ExtLoadSize =0x8000   ;装载代码大小为 0x8000（字节为单位）
  JMP  0x8000           ;注意这里因为不同程序 Main_Loop 位置可能会改变改为 0x8000
```

SPI Flash 内部空间 0x8000~0xFFFF（字节为单位）

```
;程序 2
ORG 0x4000              ;定位到外部 RAM 空间，现在里面放有程序
  ……  ;程序 2 代码
  ;接下来装载程序 3
  ExtLoadAddr=0x8000    ;从 SPI Flash 地址 0x10000 开始装载代码（字节为单位）
  ExtLoadSize =0x8000   ;装载代码大小为 0x8000（字节为单位）
  JMP  0x8000           ;注意这里因为不同程序 Main_Loop 位置可能会改变改为 0x8000
```

SPI Flash 内部空间 0x10000~0x17FFF（字节为单位）

```
;程序 3
ORG 0x4000              ;定位到外部 RAM 空间，现在里面放有程序
  ……                    ;程序 3 代码
  ;接下来装载程序 4
  ExtLoadAddr=0x10000   ;从 SPI Flash 地址 0x18000 开始装载代码（字节为单位）
  ExtLoadSize =0x8000   ;装载代码大小为 0x8000（字节为单位）
  JMP  0x8000           ;注意这里因为不同程序 Main_Loop 位置可能会改变改为 0x8000
```

单片机会按照这样的次序运行：

①上电后在运行到 Main 之前将程序 1 代码从 SPI Flash 装载到 0x4000~0x7FFF 区域。

②跳转到 0x4000 开始执行程序 1 代码。

③程序 1 代码执行完跳到 0x8000 位置将将程序 1 代码从 SPI Flash 装载到 0x4000~0x7FFF 区域。

④跳转到 0x4000 开始执行程序 2 代码。

⑤程序 2 代码执行完跳到 0x8000 位置将将程序 3 代码从 SPI Flash 装载到 0x4000~0x7FFF 区

域。

⑥跳转到 0x4000 开始执行程序 3 代码。

……

只要保证程序跳转到 0x4000 之前已经将新的代码装载到 0x4000~0x7FFF 区域，在向 0x8000 跳转之前设置好 ExtLoadAddr 和 ExtLoadSize 这两个参数，就可以循环动态调用多个存储在 SPIFlash 中的不同程序，实现 RAM 动态装载功能。

现在高端单片机应用程序大都已经不是在 ROM 里直接运行，ROM 只是用来存放程序，单片机上电后通过一小段在 ROM 中直接执行的代码将程序装载到 RAM 中，然后在 RAM 中执行。由于这类单片机 RAM 空间都比较大，而且支持多种大容量存储设备，所以实现动态装载更为容易，可以将整个程序编译好的机器代码存放在存储设备中，需要运行程序时候就将整个程序的机器代码一次性装载到 RAM 中执行，不过有一点要留意，进行装载时要考虑到中断的影响。

2.10. 程序也可被压缩

想必大家都熟悉功能强大的压缩软件 WINRAR/WINZIP，在保证数据百分之百正确的情况下可以将数据压缩到原来的几分之一甚至几十分之一，许多时候单片机开发人员都因为存储空间不够而苦恼，如果能把这个压缩功能应用到单片机程序存储方面，该是一件多好的事情。

能够在 RAM 中实现程序的动态加载是实现程序被压缩的基础，只要明白了动态加载就很容易理解程序的压缩，如果我们的代码能够实现 WINRAR/WINZIP 的功能，存储的程序已经被压缩过，只要我们在动态加载过程中加入解压缩功能就可以将原始程序代码加载到指定位置。

不要被压缩算法吓倒，我们不需要做到 WINRAR/WINZIP 那么强大的功能，复杂的算法对于单片机速度来说也是一个应用上的障碍，所以我们可以选用一些简单的压缩算法，只要能将程序压缩两三倍，对于单片机存储空间来说已经是革命性的改良。

刚好我们在以往的产品中用到压缩功能，不妨用我们当时压缩和解压缩程序来进行说明，为了让大家对采用压缩功能实际效果有一个直观的了解，这里我用一个 ARM 的程序进行压缩和解压缩功能演示。

这是我放在电脑里面与压缩和解压缩有关的一些文件，可以看到压缩的代码会多，C 代码大约有 20k 的样子，而解压缩代码相对较少，大约 8k，为方便演示将这部分压缩和解压缩代码分别生成可以在 PC 上运行的程序（实际上压缩部分必须放在 PC 上）。

图 2.10.-1 电脑模拟程序资源图

现在我用 PC 版的压缩程序来压缩 ARM 的测试程序 ARM_Code.bin，电脑给我们显示了压缩的结果，压缩后的文件大小大约是原始文件的 1/3，效果还算不错。

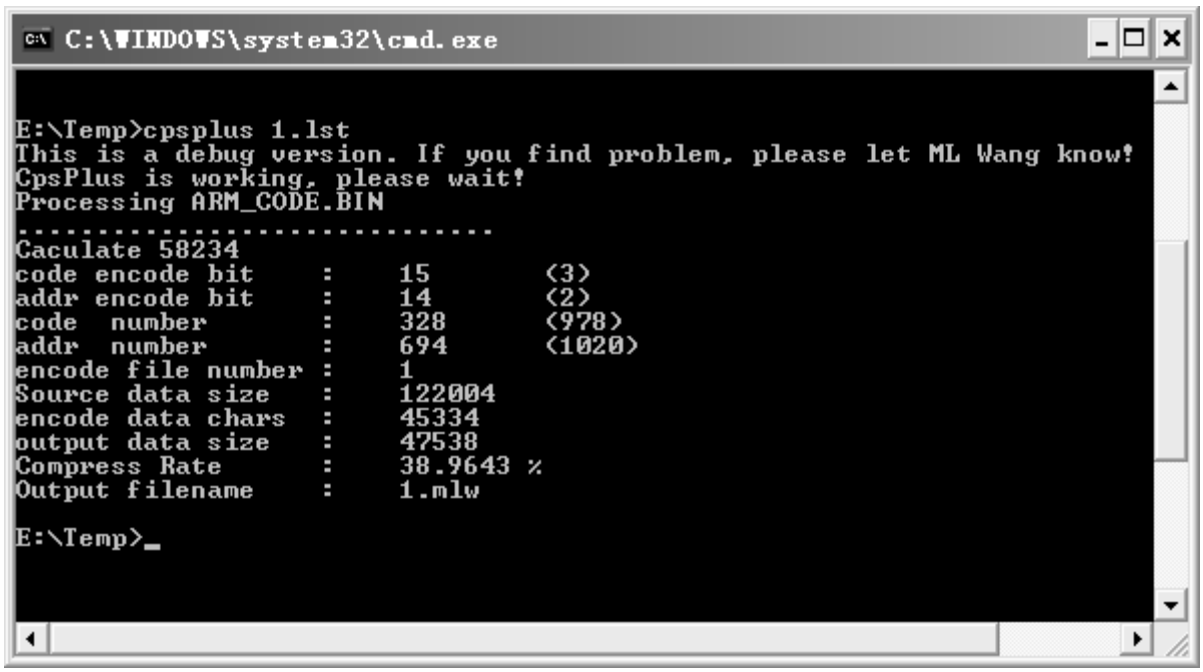


图 2.10.-2 电脑模拟压缩结果图

实际上压缩过程并不需要单片机进行，我们只要将压缩好的数据存到存储器中，当单片机读取

这些数据的同时进行解压缩，然后放到 RAM 中指定位置。既然是单片机来完成解压缩功能，我们就要考虑单片机是否有足够的空间来存放解压缩代码、单片机对数据解压缩的速度是否够快。我把解压缩的代码放到一个 ARM 工程里面，用 ADS 编译，编译结果显示解压缩只需要 1744 字节来存储代码，另外在提供 6484 字节给解压缩时的中间变量使用就够了，这个结果真有点出人意料，对许多单片机来说这简直就不是什么问题。速度测试结果同样让人满意，ARM 内核的 MCU 在主频 120MHz 的情况下解压缩出 8Mbytes 的数据耗时不到 2 秒。

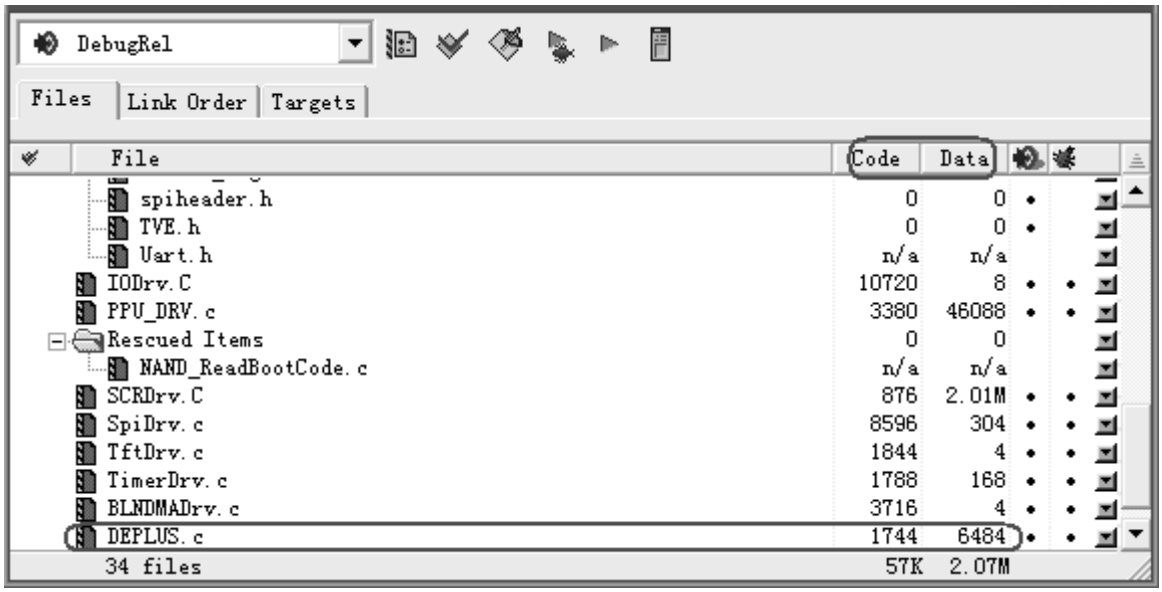


图 2.10.-3　解压程序耗用 ARM 资源示意图

●注：本章中压缩与解压缩代码为我一友人提供，他在系统构建和软件工程方面有着深厚的技术功底，这里要特别感谢他提供相关代码

数据压缩是一项与数学理论联系非常紧密的技术，现在数据压缩技术已经广泛应用到数字通讯、数字音视频信号存储和传输、图像存储等各个方面，象 DVD、MP3、数码相机、手机、网络电视无一不用到数据压缩技术。

数据压缩分为有损压缩和无损压缩两类，有损压缩是压缩后的数据再解压缩回来会有少量的数据和原始数据不同，无损压缩则是要求百分百还原。日常生活中的数字视听信号采用的是有损压缩方式，WINRAR/WINZIP 的文件压缩和我介绍的程序压缩是无损压缩。可以用一个实验来比对两种压缩方式的区别：用电脑将一张内容丰富的 BMP 图片保存成 JPG 格式，然后打开另存为 BMP 格式，再打开这个 BMP 文件另存为 JPG 格式，往复多次，你会看到图片某些细节变模糊（JPG 是有损压缩）；同样的 BMP 图片用 WINRAR/WINZIP 压缩然后解压缩，多次重复，图片效果始终保持不变。

网上有一篇《笨笨数据压缩教程》，写得浅显易懂，如果你想对数据压缩了解多一些，不妨自己找过来看看。

2.11. 累计误差

使用手机时间当手表的朋友常苦恼手机的时间不怎么准，早些年这几乎是所有手机的通病，厉害的一个月就能差上一两分钟，路边一个五元钱的电子表一年下来也差不了几秒，这和几千块的高科技产品形象是严重不符，着实让人糊涂，我也没弄清楚真正的原因，但一点可以肯定这种不准是误差加软件错误导致的。

物理学原理已经告诉我们误差是永远存在的，误差不能避免但可以通过某些方法减小，而错误是可以避免的。单片机是基于物理学基础一项电子科学技术，同样摆脱不了物理学误差的束缚，在用单片机进行产品开发时，需要对误差做出充分的考虑。

手机也是单片机做的，时间不准是必然的事情，虽然我自己没有做过手机的开发，但可以从基本原理上做出一些猜测。

单片机的时钟基准是由晶振（为表述方便不提 RC 振荡器）提供，晶振自身具有一定误差，我们用 ppm 来表示晶振误差的大小，1ppm 表示误差为百万分之一，误差为 1ppm 的 1M 晶振其实际频率在 999999Hz 到 1000001Hz 之间。ppm 值越小表示晶振精度越高，价钱相应也越贵，一般的电子产品如果不是对时间有特殊要求用的都是 5ppm 以上的晶振。

误差虽小，可不能累积啊！1ppm 的晶振误差为百万分之一，粗一看会感觉非常之准，但仔细分析后就不要小瞧这百万分之一的误差，以手机时间为例，用 1ppm 的晶振和绝对时间之间的差距是百万分之一，也就是说手机时间一秒和绝对时间一秒最大可能相差百万分之一秒。

一天 24 小时，一小时 3600 秒，一天下来总共有 3600 秒*24=86400 秒，百万分之一的误差一天最多可以有 0.864 秒的误差，累积下来一个月就是 25.92 秒，接近半分钟，一年可以达到五、六分钟的大小，这就是累计误差的威力。

要想让手机时间变得更准，方法就只有这两种：一是提高晶振精度让同样时间之内的累计误差变小，用 1ppm 的晶振一年可能有五、六分钟误差，那改用 0.1ppm 的晶振就只有半分钟样子的误差了，一年半分钟的误差对人来说已经不容易察觉到，但现在单片机用 1ppm 晶振价格都不便宜，更别提 0.1ppm 的晶振，成本难以接受；二是想方法不让误差累积，如果你观察过固定电话就会发现固定电话每次有电话呼入的时候，上面的时间就会自动被调准，这是固定电话在呼入时交换机向其发送了带呼叫时间的来电信息，固定电话通过这个时间将自己时间校准，但 GSM 手机好像没有提供此项功能，具体原因不甚清楚，不过现在运营商针对这个问题推出了时间同步服务功能。

思维活跃的朋友此时一定产生了一个疑问：既然单片机用 1ppm 晶振价格不便宜，可我带的电子表或石英表价格很便宜，时间也很准，这是什么原因？这个问题真难到了我，只能说说我个人揣测的原因给大家做个参考，表的晶振振荡频率基本上都是 32768Hz，可能是实现技术最简单、市场消耗量大等因素使得频率为这种晶振价格要比其它频率晶振便宜不少，同样的价格可以买到精度更高的这种晶振。虽然频率为 32768Hz 晶振便宜，但其并不适合单片机直接用来做主频，即使是采用了 PLL 技术在内部倍频，也不是可以无限制的倍频到所需高频率，倍频出来的最高频率会有个上限。

不少高端单片机现在提供 RTC（实时时钟）功能，这种单片机有两个晶振，一个给主频率用，另外一个频率为 32768Hz 晶振和一颗备用电池向 RTC 保证时间准确与连续。

不要把软件错误当成是误差，如果手机程序在时间累加处理方面存在一些问题，会使计时变得更为不准，来看看我对手机程序处理上的一个假设。

为了实现时间功能，我们需要用一个 Timer 的定时中断来累加我们用于时间处理的计数器，手机屏幕上显示的时间只要提供秒的精度就可以，但手机需要提供秒表功能，秒的精度显然不够，需要毫秒级，工程师想着为了少用中断资源决定做一个一毫秒的定时中断，除去晶振带来的系统误差，这个定时中断非常准确，没有错误。

假定手机在通话时会产生一些中断函数执行时间会超过一毫秒的特殊中断（注意只是假设，我不知道到底有没有这种情况），这样手机通话时就会出现一些时间片没有响应一毫秒定时中断，我们用于时间处理的计数器在通话过程中出现漏加的情况，可能是原本一秒会累加 1000 次，现在通话的时候只累加了 900 次，使得计时额外变慢了 0.1 秒，这就是软件的错误，不是误差。

软件需要采取一些方法来避免这样的错误，如果手机的 MCU 支持中断优先级并支持嵌套，就可以将定时中断设到最高优先级来保证定时准确，也可以计算出其它中断可能会产生的最大延时，保证定时时间大于最大延时，从而避免出现定时中断漏进入的情况。

2.12. 让定时更准一些

以前让一个同事用 IO 来模拟 UART 发送数据，该同事采用定时中断来进行发送，每中断一次发送出一位，他自己用电脑串口接收 UART 发出的数据功能正常，可将产品送到另外的部门 A 联调时他们反馈用他们的设备好像不能稳定接收所发出的数据，经常出现产品发出数据后他们的设备没有做出相应响应。

同事用电脑对产品进行过测试这是事实，别的部门也肯定不会无中生有，麻烦的是这个部门和我们还不在一个地方办公，不能马上过去查找问题原因，于是我让部门 A 的同事帮忙用示波器看一下波形，波形图很快传了过来，另外部门 A 的同事告知我波形时间好像有点问题。所抓波形图为整个字节的宽度，所以对每个位的具体宽度时间显示并不是很清楚，从整体宽度看确实是出了问题，比正常的要宽。

让同事自己用他所写的程序连续发送 0x55（这样可以得到 010101 的波形，方便查看位宽），用示波器看每个位都宽了几个微秒。当时波特率为 9600，正常每个位宽度应该为 104 微秒的样子，这多出的几个微秒的宽度已经让发送的波形处于出错的临界状态。电脑的波特率设置比较准，所以还能正常接收产品发出的数据，但部门 A 的设备的实际波特率可能往另外一个方向发生偏差，如果是这样就难以接收到产品所发出的数据。

问同事程序实现的方法，回答是用将定时中断设为 104 微秒，每中断一次发送一位，这样看没

什么问题，不至于产生几个微秒的偏差来，接着问进定时中断程序后程序具体操作的步骤，回答先将定时的时间重设为 104 微秒。问题出在这里，该同事所用的单片机不支持自动重载功能，每次定时时间到产生中断后需要用户重设定时时间，否则就从零开始计数，加到最大后再触发中断。

同事少算了中断响应时间和进中断后代码运行到他重设定时时间位置的时间，加上他所设定的值并不是真正的 104 微秒，有零点几微秒的偏差，最后导致他每个位的宽度多出几个微秒。后面在部门内部培训时提到这个例子，另外一个同事说了句让我大为惊讶的话：“我以前也遇到过这样的情况”。客观的说这种问题的产生确实不应该，让我惊讶的是遇到这种问题居然不是个案。

单片机实现定时的方法都是内部有一个计数器，这个计数器以设定的频率自加或自减，当加减到某一个条件时就会触发中断，如果支持自动重载功能此时会从指定位置自动向计数器装入新的值，下面用一个 8bits 的定时器列举一下常见的定时方式。

①自加到 0xFF 后产生中断，计数器回到 0x00，需要在中断程序中重新设定计数器的值，定时时间等于 delay+(0xFF-val+1)/f，delay 为中断产生到重设计数器的时间，val 为重设的值，f 为计数器自加操作的频率。

②自减到 0x00 后产生中断，计数器回到 FF，需要在中断程序中重新设定计数器的值，定时时间等于 delay+(val-0x00+1)/f，delay 为中断产生到重设计数器的时间，val 为重设的值，f 为计数器自减操作的频率。

③自加到 0xFF 后产生中断，计数器自动回到 val，定时时间等于(0xFF-val+1)/f，val 为自动重载的值，f 为计数器自加操作的频率。

④自减到 0x00 后产生中断，计数器自动回到 val，定时时间等于(val+1)/f，val 为重设的值，f 为计数器自减操作的频率。

⑤自加到 val 后产生中断，计数器自动回到 0x00，定时时间等于(val+1)/f，val 为自动重载的值，f 为计数器自加操作的频率。

⑥自减到 val 后产生中断，计数器自动回到 0xFF，定时时间等于(0xFF-val+1)/f，val 为重设的值，f 为计数器自减操作的频率。

●注：有的单片机定时时间计算公式不需要另外加一

为什么①②中需要另外加上一个 delay 呢？因为定时时间的计算公式是从设定好 val 后的位置开始计算的，中断产生时候计数器里面的值并不等于 val，而是 0x00 或 0xFF，是程序在中断中更改了计数器的内容，所以需要加上前面的这段时间进行校正。不是所有的单片机都是写入新的 val 后就立即生效，有的需要等到下一次中断产生后才生效，开发时要留意具体细节。

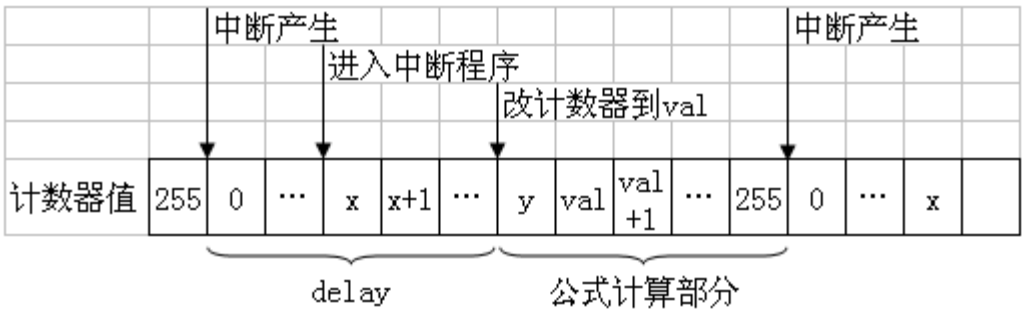


图 2.12.-1　定时中断校正示意图

2.13. 寄存器也可当 RAM

在使用一些简单的单片机进行产品开发时，工程师往往会因为 RAM 空间紧张而头疼，这些简单的单片机可能只提供几十个字节的 RAM 空间给工程师使用，当工程师编写较为复杂的逻辑功能控制程序的时候需要一定数量的空间来存放程序变量，经验不足的新工程师因不会共享变量空间而导致 RAM 有浪费会让情况更糟糕，程序写到最后变成挤 RAM 的工作。

遇到这样的问题开发工程师具有良好的 RAM 分配意识是很有必要的，如果只用来表示有和无的状态尽量用位变量，让相互之间不存在调用与被调用关系的函数使用公用 RAM 做输入输出参数，不需要保存状态信息的变量尽量不要占用固定空间等都是行之有效的方法。

如果程序写到最后出现为几个字节空间发愁的情况，这里教你一个小方法：用特殊功能寄存器来当 RAM 用。通常情况下即使是构架非常简单的单片机也都有几十个字节的特殊功能寄存器，这些特殊功能寄存器来配置 Timer/Interrupt/IO 等功能，进行产品开发时通常都不会全部使用这些功能，也就是说实际上会有一些特殊功能寄存器是闲置状态，我们可以将这些特殊功能寄存器用做来当 RAM 变量。

我们以最常见的 51 系列单片机为例来看看哪些特殊功能寄存器可以用来当 RAM。

表 1　P89C51/89C52/89C54/89C58 特殊功能寄存器

名称	说明	地址	位地址和位功能								复位值
ACC*	累加器	E0H	E7	E6	E5	E4	E3	E2	E1	E0	00H
AUXR#	辅助功能寄存器	8EH	-	-	-	-	-	-	-	A0	XXXXXXX0B
AUXR#	辅助功能寄存器 1	A2H	-	-	-	-	-	-	-	DPS	XXXX00X0B
B*	B 寄存器	F0H	F7	F6	F5	F4	F3	F2	F1	F0	00H
DPTR:	数据指针(双字节)										
DPH	数据指针高字节	83H									00H
DPL	数据指针低字节	82H									00H
			AF	AE	AD	AC	AB	AA	A9	A8	
IE*	中断使能	A8H	EA	-	ET2	ES	ET1	EX1	ET0	EX0	0X000000B
			BF	BE	BD	DC	BB	BA	B9	B8	
IP*	中断优先级	B8H	-	-	PT2	PS	PT1	PX1	ET0	EX0	① XX000000B
			B7	B6	B5	B4	B3	B2	B1	B0	
IP1#	中断优先级高字节	B7H	-	-	PT2H	PSH	PT1H	PX1H	PT0H	PX0H	XX000000B
			87	86	85	84	83	82	81	80	
P0*	I/O 口 0	80H	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0	FFH
			97	96	95	94	93	92	91	90	
P1*	I/O 口 1	90H	-	-	-	-	-	-	T2EX	T2	FFH
			A7	A6	A5	A4	A3	A2	A1	A0	
P2*	I/O 口 2	A0H	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8	FFH
			B7	B6	B5	B4	B3	B2	B1	B0	
P3*	I/O 口 3	B0H	RD	WR	T1	T0	$\overline{INT1}$	$\overline{INT0}$	TxD	RxD	FFH
PCON#[1]	电源控制	87H	SMOD1	SMOD	-	POF[2]	GF1	GF0	PD	IDL	00XXX000B
			D7	D6	D5	D4	D3	D2	D1	D0	
PSW*	程序状态字	D0H	CY	AC	F0	RS1	RS0	OV	-	P	000000X0B
RACAP2H#	定时器 2 捕获高字节	CBH	③								00H
RACAP2L#	定时器 2 捕获低字节	CAH	③								00H
SADDR#	从地址	A9H	②								00H
SADEN#	从地址屏蔽	B9H	②								00H
SBUF	串口数据缓冲区	99H									XXXXXXXXB
			9F	9E	9D	9C	9B	9A	99	98	
SCON*	串行口控制	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	00H
SP	堆栈指针	81H									07H
			8F	8E	8D	8C	8B	8A	89	88	
TCON*	定时器控制	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00H
			CF	CE	CD	CC	CB	CA	C9	C8	
T2CON*	定时器 2 控制	C8H	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	$C/\overline{T2}$	$CP/\overline{RL2}$	00H
T2MOD#	定时器 2 模式控制	C9H	-	-	-	-	-	-	T2OE	DCEN	XXXXXX00B
TH0	定时器高字节 0	8CH	④								00H
TH1	定时器高字节 1	8DH	④								00H
TH2#	定时器高字节 2	CDH	④								00H
TL0	定时器低字节 0	8AH	④								00H
TL1	定时器低字节 1	8BH	④								00H
TL2#	定时器低字节 2	CCH	④								00H
TMOD	定时器模式	89H	GATE	$C/\overline{T}$	M1	M0	GATE	$C/\overline{T}$	M1	M0	00H

图 2.13.-1　P89C5X 寄存器表

从特殊功能寄存器表可以看出 51 支持中断、定时器和 UART 功能，我们从这几个地方来寻找可以利用的特殊功能寄存器。

位置①的特殊功能寄存器用来设置各个中断的优先级，每个中断可以从四个不同的优先级中选择一个优先级，当一个中断产生后正在执行中断程序时，如果有一个更高优先级的中断产生，会转去先响应这个高优先级的中断。如果我们所用的中断对优先级并没有特殊需求的话可以不用理睬中断优先级的设置,这时我们就可以将这两个特殊功能寄存器用做 RAM,但每个寄存器只能提供 6bits，不能直接用作 8bits 的数据存储。

位置②的特殊功能寄存器只有在进行 UART 通讯而且是要求 51 单片机自动支持从地址判断的模式才会用到，该模式采用的是 9bits 数据，现在实际应用已经比较少采用到这种模式，如果实在需

要进行地址判断我们也可以通过 8bis 数据模式在数据包中增加目的地址来达到同样功效。只要产品不需要使用 UART 或者是需要 UART 但会关闭从地址自动判断功能，那么我们就可以将这两个特殊功能寄存器用做 RAM。

位置③和位置④是用来对定时器进行控制的特殊功能寄存器，其中位置③用来对 UART 的波特率进行设定，通常产品并不需要三个定时器都打开，就算可能有多个地方需要计时，有经验的工程师也可以共用同一个基准定时器，比如程序中做一个 1ms 的基准定时器，只要程序结构设计好是可以让程序中所有需要定时的地方都用这个 1ms 的基准完成定时。这样从这里也可以找出几个字节用做 RAM。

特殊功能寄存器另外还有一个特殊的用法，有时候产品想区分上电复位、硬件 Reset 复位和软件 Reset 复位操作，但所用的单片机并没有提供此功能，可以在特殊功能寄存器中寻找内容不受复位影响的特殊功能寄存器来实现此功能。上电复位后该寄存器状态可能未知，程序启动后马上将寄存器写入特殊值一，如果软件复位操作入特殊值二。程序启动后根据寄存器内容来判断复位方式，不是两个特殊值基本上可以肯定是上电复位，特殊值一为硬件 Reset，特殊值二为软件 Reset。在 51 单片机的特殊功能寄存器中我没有找到合适的寄存器，这里不进行举例。

2.14. 清中断标志的位置

可能有人有疑问，中断标志位不是在中断函数中清掉就可以了吗？难道还有什么特殊的要求？没错，在中断函数中清中断标志位其实也需要一定的技巧，清除位置的不同有可能得到不同的工作结果。

……（详见完整版）

2.15. 键盘扫描

扫描键盘是单片机需要程序完成的一项最基本功能，许多资料都有介绍进行键盘扫描的原理和方法，这里不再详述，只是针对键盘扫描中一些需要注意的地方加以强调。

说到键盘扫描都会提及去抖动处理，去抖动的方法有许多，连续多次重复一致、间隔几个毫秒两次状态一致都是常用的方法，去抖动最主要的目的是防止串进来的干扰导致误判按键动作，另外就是将按键的按下和松开可靠的区分开，后面一点不会对功能产生明显的不良影响。

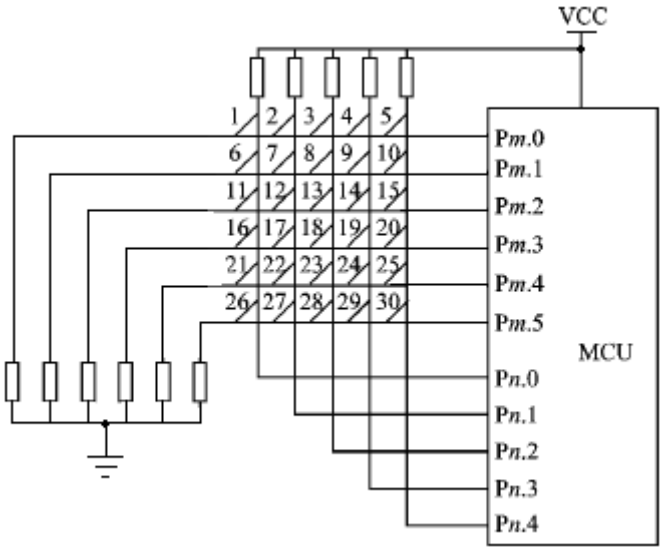


图 2.15.-1　单片机键盘矩阵示意图

当按键非常多的时候常会采用 IO 口扫描键盘矩阵的方式来实现，这样可以用比较少的 IO 口就可以得到足够多的按键数。例如图示中的 30 个按键，如果一个 IO 对应一个按键需要 30 个 IO，但做成 5*6 矩阵模式只要 11 个 IO 就可以满足需求。

扫描键盘矩阵时 Pn 为输出口，Pm 为输入口，扫描键盘时 Pn.0~4 依次输出高电平，比如当前 Pn.0 输出高，如果最左边一行有键按下对应行的 Pm.x 就能读到 1，否则读到的是 0，要是现在 Pm.3 读到 1，说明键 16 按下。

这种键盘矩阵处理方式存在一个问题，如果同时按下键 1 和键 2，Pn.0 和 Pn.1 之间是直接短路状态，Pn.0 输出高而 Pn.1 输出低会让两者输出状态产生冲突，所以程序在 Pn.x 输出高之前要先将其它 Pn.y 改为输入状态才能避免冲突的发生。但这样改动后即便没有按键 Pm.x 也能读到 1，程序还要做另外一个处理，Pn.x 分别输出高和低两个状态，对应的 Pm.x 能相应准确读回 1 和 0 才说明有键按下。

虽然避免了冲突，但还是有问题，如果同时按下键 1、键 2 和键 6，当 Pn.0 输出高低 Pm.0 和 Pm.1 都能随之读到 1 和 0，判断为键 1 和键 6 按下，当 Pn.1 输出高低 Pm.0 和 Pm.1 也都能随之读到 1 和 0，判断为键 2 和键 7 按下，对于键 2 状态做出错误判断，错误的原因是此时 Pn.0、Pn.1、Pm.0 和 Pm.1 四点形成短路关系。

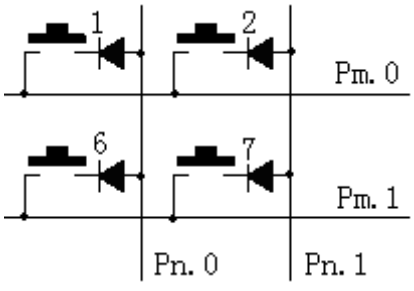


图 2.15.-2　键盘矩阵二极管保护示意图

我们给键盘矩阵加上一些二极管，同时扫描键盘的程序改回 Pn.x 输出高的同时其它 Pn.y 输出低，键被按下的行输入状态会随其所在列输出高低状态相应变化。再来看看同时按下多个键的情况，依然同时按下键 1、键 2 和键 6，当 Pn.0 输出高和低时 Pm.0 和 Pm.1 都能随之读到 1 和 0，判断为键 1 和键 6 按下，当 Pn.1 输出高和低时候只有 Pm.1 能读到 1 和 0，Pm.0 变为始终读到的都是 0，可以正确判断出只有键 7 按下。

所加的二极管起到了多个按键时候不会形成短路状态，保证行列扫描的时候只有当前列输出可以通过按键传递到按键所在列，不会发生误传递，如果从电路可靠性来说加上二极管还是比较重要的，可是许多有经验的硬件工程师都不知道这个风险。但加二极管会让成本稍有增加，如果不需要支持多个按键可以不要二极管，程序检测到有多个按键的时候判定按键无效。

键盘扫描还可以用电阻分压然后用 ADC 测量电压来区分按键。

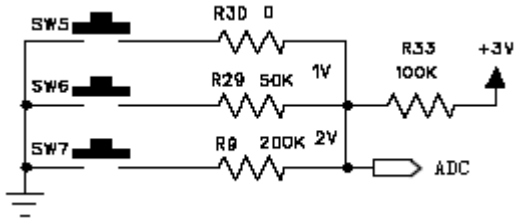


图 2.15.-3　电阻分压键盘示意图

SW5 单独按下，Uadc=0V

SW6 单独按下，Uadc=1V

SW7 单独按下，Uadc=2V

当按下不同键的时候 ADC 测量到的电压会不相同，从而判断是哪一个键按下，但这种方式多个按键被按下的时候会出错，所以不支持多按键模式，另外因为电阻阻值大使得按键按下或者放开的时候有一个比较明显的充放电过程，所以对去抖动的处理要比普通按键要求严格，否则会测到按下或松开按键的中间过程导致键值判断出错，按键应用接触电阻小的金属按键，不要用导电橡胶，不然导电橡胶在似按非按的状态下会产生一个比较大的接触电阻，从而导致电阻分压和理想状况出现比较大的差异。

这里给大家推荐一种键盘去抖动的处理方法，用一个变量来记录按键状态，规定该变量为 0 表示按键松开，达到规定值表示按键按下。先将这个变量初始值设为规定值的一半，在程序主循环或者定时中断中循环间隔查看按键状态，松开减一，按下则加一，直到变量到达 0 或者规定值，这种方法一样可以起到很好的去抖效果，而且不需延时等待时间。

2.16. 视觉暂留

视觉暂留是人的一种生理现象，物体消失后在视网膜上的影像还能持续一段时间，很简单的例子就是手慢慢的动我们可以很将手看得很清楚，如果手快速挥动我们则看到手变出许多虚影，这就是视觉暂留现象，人们利用视觉暂留特性发明了电影、电视等给生活带来无限精彩的产品。

单片机做显示的时候也可以利用视觉暂留特性，给出一个用单片机控制数码管显示的例子。

常见数码管有八条管脚，其中七条管脚分别对应用于显示的七个 LED，另外一条管脚是公共脚，根据共阴、共阳的类别另外的这条脚选择接地或电源。图示中七个 LED 编号为 a~g，bc 点亮显示 1，而 abged 点亮显示 2，依次类推可以显示出 0~9 这十个数字，还可以现实出其它状态来表达特定信息。

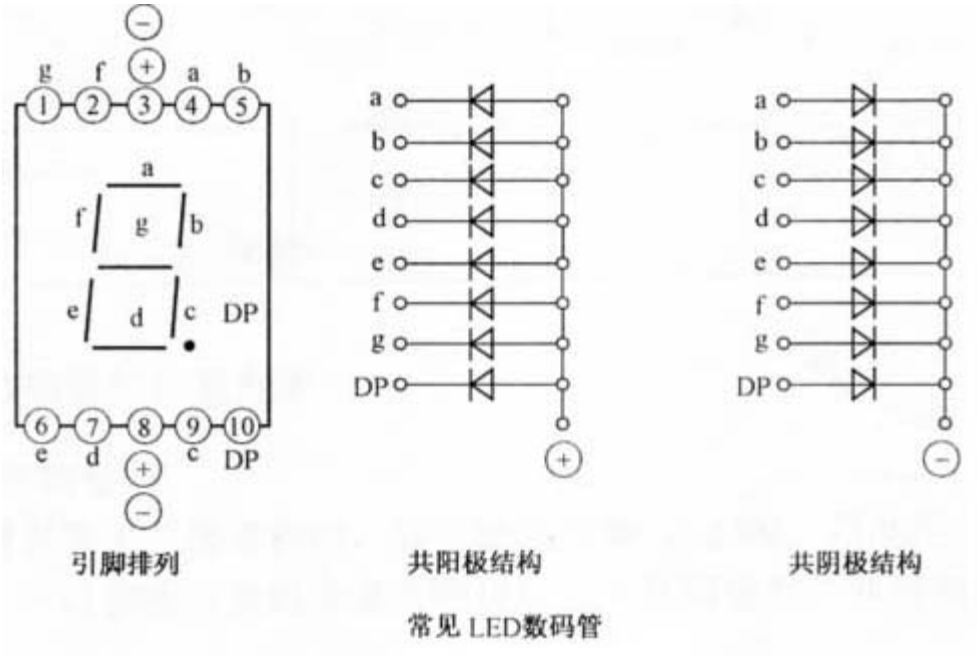


图 2.16.-1　数码管示意图

用 2051 单片机做一个显示时间的产品，要求可以现实时分秒的时间，数码管采用共阳类型，如果不利用视觉暂留，每个数码管需要八条 IO 来进行控制，六个数码管总共需要四十八条 IO，就需要另外增加器件对 2051 进行 IO 扩展，但如果利用视觉暂留特性，我们不用增加任何器件，将数码管控制 LED 的七条管脚并联在一起，另外再用六条 IO 分别控制每个数码管的公共极，当控制 LED 的 IO 输出时，只有公共极被选中的数码管才会被点亮。

我们已经知道视觉暂留的时间会超过 0.1 秒，显示程序按时分秒的顺序依次显示这六个数码管，每个数码管显示 0.01 秒后就切换到下一个，当显示到第六个数码管时第一个数码管还只显示结束 0.05 秒，视觉暂留效应让人察觉不到第一个数码管停止输出显示，此时人眼感觉第一个数码管仍然

保持输出显示，接下来循环再次显示第一个数码管，这样就可以让人感觉六个数码管好像在同时输出一样。

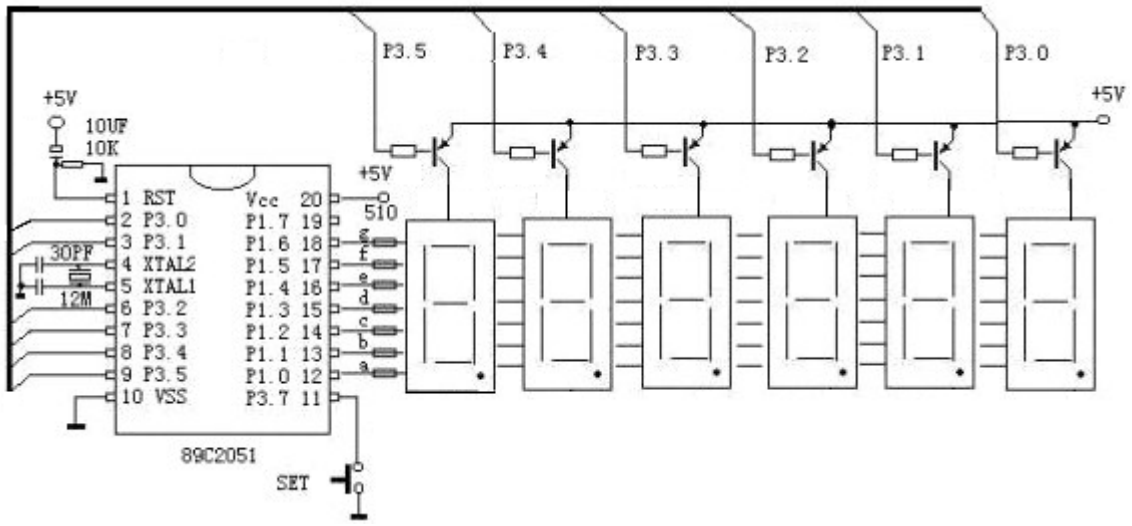


图 2.16.-2　数码管利用视觉暂留应用示意图

2.17. 让耳朵优先

生活中常用耳聪目明这样的词来形容一个人听觉和视觉敏锐，实际上人的听觉和视觉的敏锐度除了人与人之间存在差异外，正常人的听觉敏锐度和视觉敏锐度也有比较大的差异，人的听觉要比视觉敏感。

视觉暂留特性告诉我们视觉存在着比较大的惰性，对于变化达到几十 Hz 的图像，视觉就已经跟不上响应速度，而听觉不一样，经验告诉我们普通人对几十 kHz 的声音变化都能察觉到，电影和电视只要每秒输出几十帧画面就会让人觉得图像流畅，而随声听、MP3 则需要 44.1kHz 的声音输出才能让人不感觉到明显失真。

既然人的听觉要敏感那么当用单片机来同时处理音频和视频数据的时候，应该优先音频，最大可能的保证音频时间轴和实际情况一致。单片机也为用户考虑了这一点，通常情况下有关音频处理的中断优先级要比视频高，有的甚至把音频处理的中断优先级放到最高来保证音频处理的实时性。如果你用的单片机系统需要同时处理音频和视频，现在速度不够，请记住先牺牲视频性能以保证音频性能。

人的视觉还有一些有意思的特性，对亮度敏感而对颜色迟钝，一张照片如果把亮度提高，人就会产生这张图片非常清晰的错觉。你可以自己做个小实验来验证一下视觉的这个特性，一张白纸上画一条黑线，另外一张彩纸上面用另外颜色画一条同样粗细的线（比如黄纸上画绿色的线），两张纸并排贴在墙上，然后逐渐远离纸张，你会发现彩色的线看不清楚的时候黑线依然很清楚。

2.18. 1000 与 1024

单片机（计算机）采用二进制来进行数据处理，而人们日常生活是采用十进制来进行数字表达，如果想要单片机和人一样用十进制来处理数据，从技术上说目前无法实现，如果要人都去适应单片机的二进制，那要颠覆人们上千年的计数习惯，基本上是异想天开。实际上也并不是要共用同一种进制才行，单片机用它的二进制，人继续自己的十进制，是独木桥和阳关道的关系，两者并不冲突，只是在一些相关技术的发展中，出现了一些容易混淆的东西。

人们将用小写字母 k 来表示千，1000 就是 1k，单片机也它的 k，不过因为二进制的原因，它的 k 不是 1000 而是 1024（2 的 10 次方）。单片机用 1024 做为它的 k 有其苦衷，数字电子技术只有 0 和 1 两种逻辑状态,这样决定了单片机无论是数据运算还是存储都是 2 的整数次方为单位最为方便。如果一定要把 1000 当做单片机的 k，会给单片机在数据运算和存储方面带来许多不便，就好比银行将钱以一扎一万捆起来以便清点，你坚持一扎是一万零一块也行，只怕银行职员会恨你到咬牙切齿，正是这个原因，单片机选了一个和 1000 最为接近而且是 2 的整数次方倍的数 1024 当做 k。

单片机会将一些理论实用化，可从事理论研究的人都是用数学方法来进行理论分析，这样决定了理论出来的结果都是以十进制进行数字表达，当把这些理论用到单片机上去的时候矛盾就会显现出来。

语音处理是单片机经常会用到的一项技术，最简单的应用是播放数字化的语音信息，原本是连续的模拟语音信号经过数字化转换成离散的数字语音信息，数字化是对模拟信号等间隔离散采样，这个间隔就是采样率，采样率越高，数字语音信息就和原始模拟语音信号越接近。采样率越高，同样的信号所得到的数字信息就越多，实际应用是期望数字信息越少越好，这样就对实际应用构成一个负担，太高的采样率可能实际应用处理不过来，从事基础技术研究的人就开始研究不同采样率得到的数字语音信息回放出来人耳可以感觉到的差异大小，对于话音、声效、音乐各自需要多高的采样率人耳才可以接受，分析结果对采样率描述是用 k 来表示，比如 8k 采样率就是一秒采样 8000 次。

可单片机处理和存储数据的 k 是 1024，在程序里面如果说 1k 数据表示 1024 个数据，8k 数据则是 8192 个数据，和采样率 8k 表示的 8000 并不相同，这个问题让不少人混淆不清，最常见的就是把采样率 8k 理解成每秒采样 8192 次。

晶振用的兆（M）也存在同样的问题，频率 1MHz 的晶振频率是 1000000Hz 而不是 1024*1024Hz，不过有个特例，32kHz 的晶振是 32768Hz（也有人表述为 32.768k 的）。

在用单片机进行产品开发时候。一定要留意到这一点，如果把采样率和晶振频率的 k/M 与单片机数据处理和存储的 k/M 混淆在一起，表演看可能所编写的程序功能正常，但实际上缺在时间方面出现了大约 2.4%的偏差，比如用软件方式播放 8k 采样率声音每一秒输出 8192 个数据就会让回放的声音比实际要快一点，音调变高。

2.19. PWM

我们知道交流电经过变压器降压后再通过整流桥整流可以得到输出波形全为正弦波正半周的连续波形。

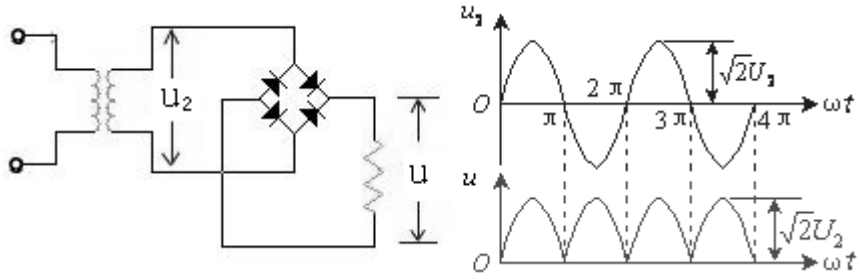


图 2.19.-1　交直流整流示意图

如果在整流桥输出加上一个大的整流电容，因为电容的充放电效应使得原来的正弦波正半周变成上下起伏的锯齿波，负载电阻越小，锯齿波的上下波动幅度越大。

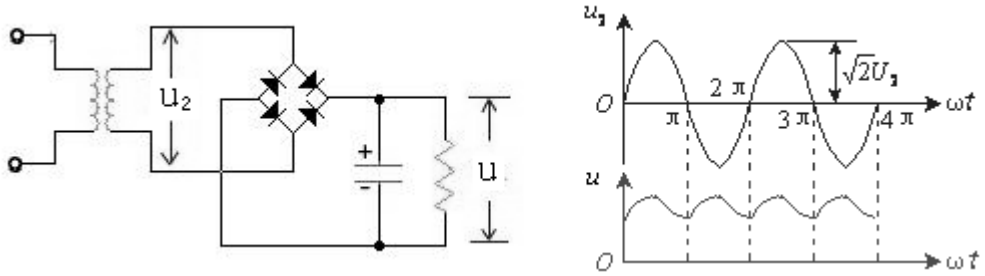


图 2.19.-2　带滤波电容交直流整流示意图

将 U2 变成方波，负载电阻两端的电压还是锯齿波，只是上升和下降的速度发生了一些变化。

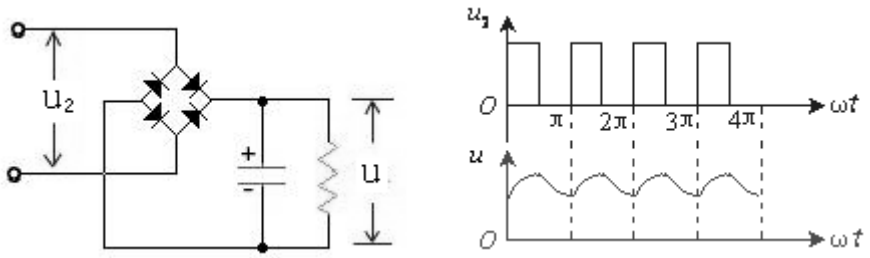


图 2.19.-3　方波整流示意图

U2 为方波的方式就是 PWM 用在电源管理上的一种表现形式。在周期不变的情况下，如果方波输出高电平的比例越大，锯齿波上升部分就越多，从而电压的平均幅度越高，如果整个周期都输出高

在不考虑负载影响的情况下电容两端的电压等于方波的幅度。如果将周期变短，锯齿波上升和下降时间也随着变短，锯齿波电压波动的幅度就越小。

对比波形可以发现：方波与交流信号存在一些不同，方波不会出现小于零的电压。这样对于方波电路，实际上我们可以把二极管整流电路去除，在负载两侧会得到几乎完全一样的波形，这就是 PWM 调压的原理

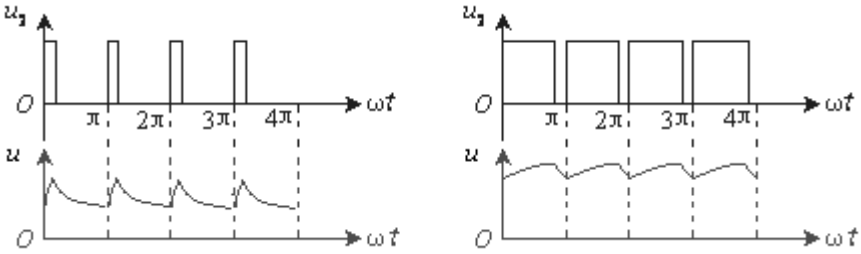


图 2.19.-4　PWM 效果示意图

PWM 是输出一个周期和占空比可调的方波，如果将这个方波用于控制电源可以得到一个输出平均幅度与占空比成正比、波动幅度和方波频率成反比的电源。开关电源就是用此原理来实现，开关电源的控制芯片会尽量让自己的工作在比较高的开关频率下，这样可以使其输出纹波（锯齿波电压波动）变小。

●注：占空比是方波输出高所占的周期比例

PWM 主要用途是通过对一个的电源开关控制可以得到输出电压大小和占空比成正比的电源，虽然 DAC 也可以输出与数字对应的电压，但这个电压驱动能力很弱，要想提供比较大的驱动能力实现起来很难，但 PWM 很简单，只要用三极管等做为开关控制元件向后面的大负载提供电源。对电源的控制实际上也是对输出功率的控制，所以 PWM 在马达转速、灯光亮暗这类控制上有着广泛的应用。