

目录

目录.....	1
导言.....	4
第一章 单片机基础.....	5
1.1. 什么是单片机.....	5
1.2. 单片机是如何工作的.....	7
1.3. 单片机与电脑的区别.....	11
1.4. 晶振.....	13
1.5. 系统时钟和周期.....	14
1.6. 单片机指令和汇编语言.....	17
1.7. RAM/ROM 的作用.....	23
1.8. 单片机接口.....	25
1.9. 接口驱动能力.....	31
1.10. 方便实用的中断.....	31
1.11. 函数和堆栈.....	44
1.12. 单片机 PAGE/BANK 概念.....	47
1.13. CISC 与 RISC.....	49
1.14. 为什么 DSP 跑得快.....	51
1.15. 单片机产品开发常见用语.....	53
第二章 单片机应用小技巧.....	58
2.1. 用 IO 模拟接口.....	58
2.2. 交流特性显神通.....	63
2.3. 电阻网络低成本高速 AD.....	65
2.4. 利用电容充放电测电阻.....	66
2.5. 晶振也能控制电源.....	67
2.6. 如何降低功耗.....	68
2.7. 开机请用 NOP.....	68
2.8. 查表与乘除法.....	69
2.9. RAM 动态装载程序.....	69
2.10. 程序也可被压缩.....	76
2.11. 累计误差.....	79
2.12. 让定时更准一些.....	80
2.13. 寄存器也可当 RAM.....	82
2.14. 清中断标志的位置.....	84
2.15. 键盘扫描.....	84
2.16. 视觉暂留.....	87
2.17. 让耳朵优先.....	88
2.18. 1000 与 1024.....	89
2.19. PWM.....	90
第三章 单片机高级特性.....	92
3.1. Cache.....	93
3.2. 总线.....	102
3.3. DMA.....	111
3.4. 存储器管理.....	114
3.5. 嵌入式与操作系统.....	121
什么是嵌入式.....	121
嵌入式误区之不死机.....	127
嵌入式效率.....	132

第四章	单片机 C 语言.....	134
4.1.	单片机 C 语言.....	134
4.2.	for()和 while()循环.....	135
4.3.	循环里的 i++与 i--.....	143
4.4.	优化的方法与效果.....	145
4.5.	全局变量的风险.....	148
4.6.	变量类型与代码效率.....	154
4.7.	慎用 int.....	157
4.8.	危险的指针.....	158
4.9.	循环延时.....	166
4.10.	运算表达式.....	170
4.11.	溢出.....	176
4.12.	强制转换.....	176
4.13.	高效实用位运算.....	179
4.14.	宏和 register.....	184
4.15.	手机里的计算器.....	193
4.16.	函数设计.....	200
4.17.	某产品函数编写规则.....	206
第五章	问题分析与调试.....	208
5.1.	应该具备基本硬件能力.....	208
5.2.	将自己站在别人角度来思考问题.....	211
5.3.	先找自己原因再假定他人出错.....	216
5.4.	充分发掘 IDE 调试工具功能.....	218
5.5.	IDE 调试工具也会导致错误发生.....	227
5.6.	没有 IDE 调试工具的测试.....	227
5.7.	C 语言要多查看汇编代码.....	229
5.8.	养成查看寄存器内容的习惯.....	232
5.9.	中断的一些特殊情况.....	233
5.10.	别迷信文档与硬件.....	237
5.11.	程序暂停不代表所有模块暂停.....	239
5.12.	几种仪器好帮手.....	239
5.13.	多用电脑工具软件.....	242
5.14.	串口通讯不能使用隔离变压器分析实例.....	246
5.15.	Cache 导致录音有杂音分析实例.....	247
5.16.	Cache 导致 RAM 验证结果不对分析实例.....	251
5.17.	双口 RAM 读写竞争出错分析实例.....	252
第六章	实际产品开发.....	256
6.1.	如何开发一个产品.....	256
6.2.	学会看电气参数表.....	258
6.3.	接口的匹配.....	265
6.4.	电源和地的影响.....	268
6.5.	成本意识.....	272
6.6.	别烦流程图.....	276
6.7.	功能的全面与实用.....	277
6.8.	批量产品的替代方案.....	280
6.9.	多了解新器件.....	281
6.10.	尽可能让生产更方便.....	284
6.11.	误差分析.....	287
6.12.	电磁兼容.....	290
6.13.	上电与测试.....	291
6.14.	程序版本发放记录.....	293

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：dai Shangju@163.com MSN：sj_dai@hotmail.com

未经本人书面许可，任何人不得将本书部分或全部内容出版发行。

转载需注明出处并保留原作者署名。

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

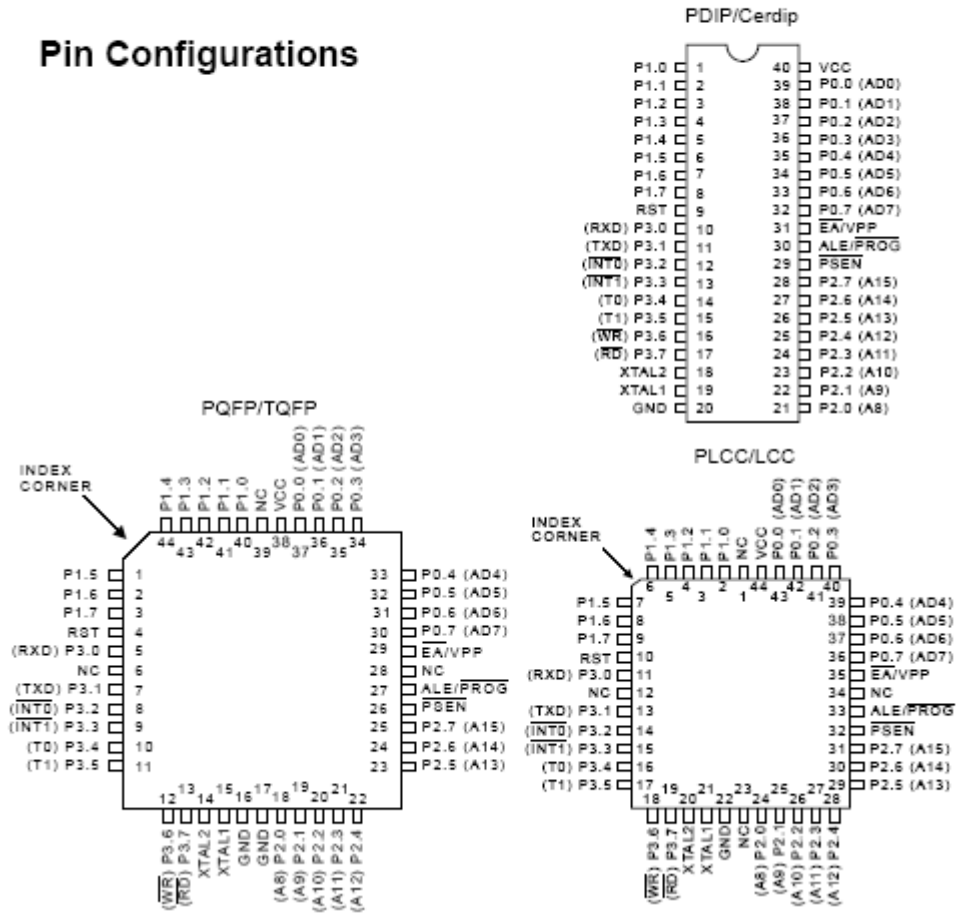


图 1.1. -1 AT89C51 封装图

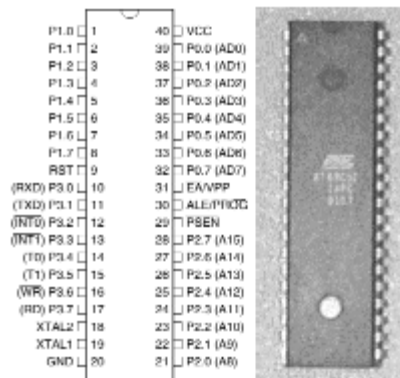


图 1.1. -2 AT89C51 DIP 封装和实物对照图

●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 后自己再将管脚绑定（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？

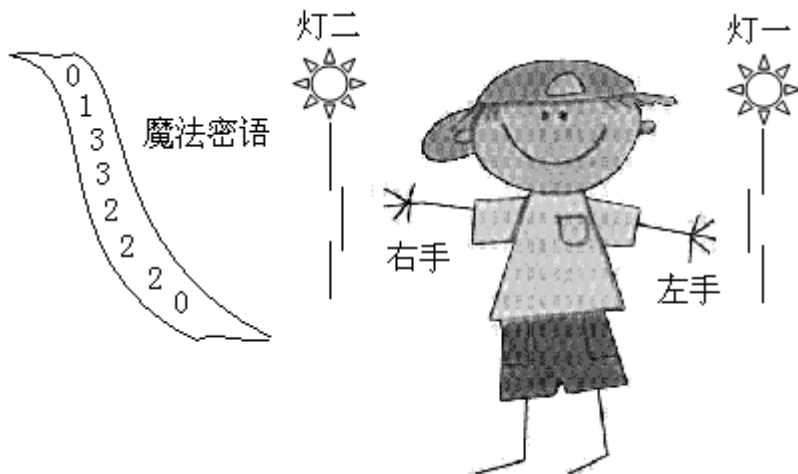


图 1.2. -4 魔法施展图

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。既然如此，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	LOR0	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	LOR1	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` 认识的魔法密码——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
LOR0	0	LOR0	0
L1R0	1	L1R0	1
L1R1	3	L1R1	3
L1R1	3	NOP	4
LOR1	2	LOR1	2
LOR1	2	NOP	4
LOR1	2	NOP	4
LOR0	0	LOR0	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？

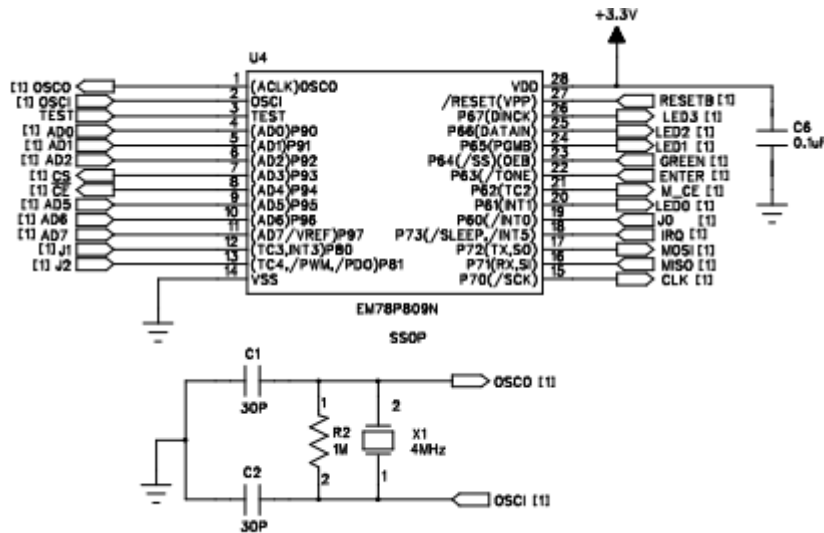


图 1.2.-7 某真实产品电路图

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。

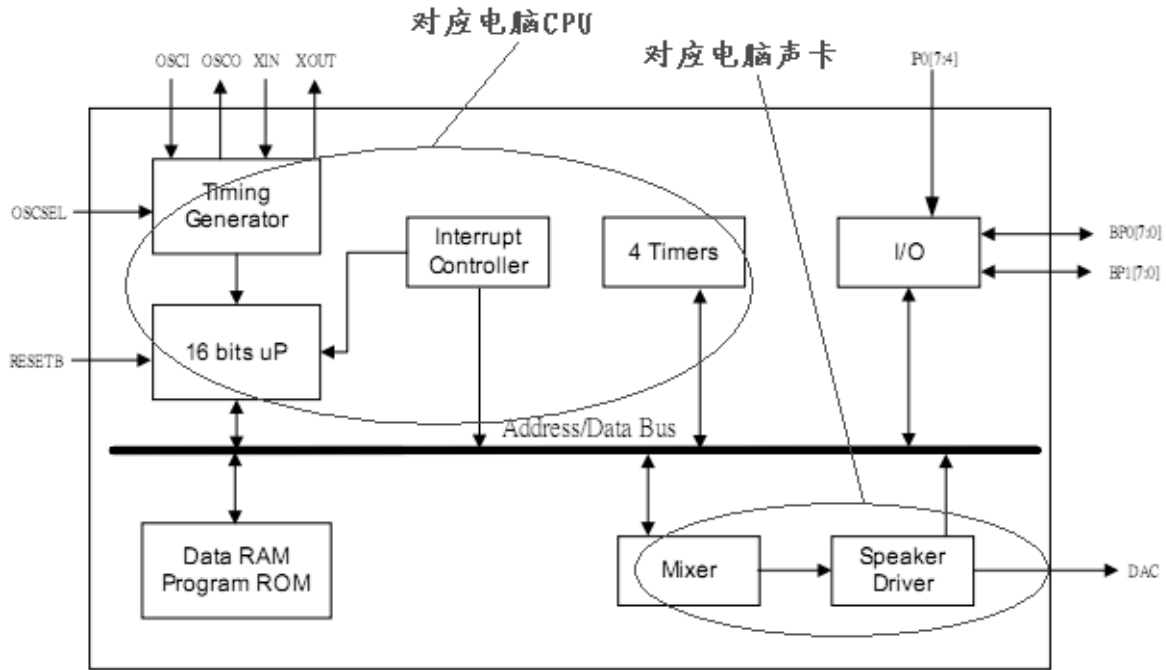


图 1.3.-1 某低端专用单片机构架图

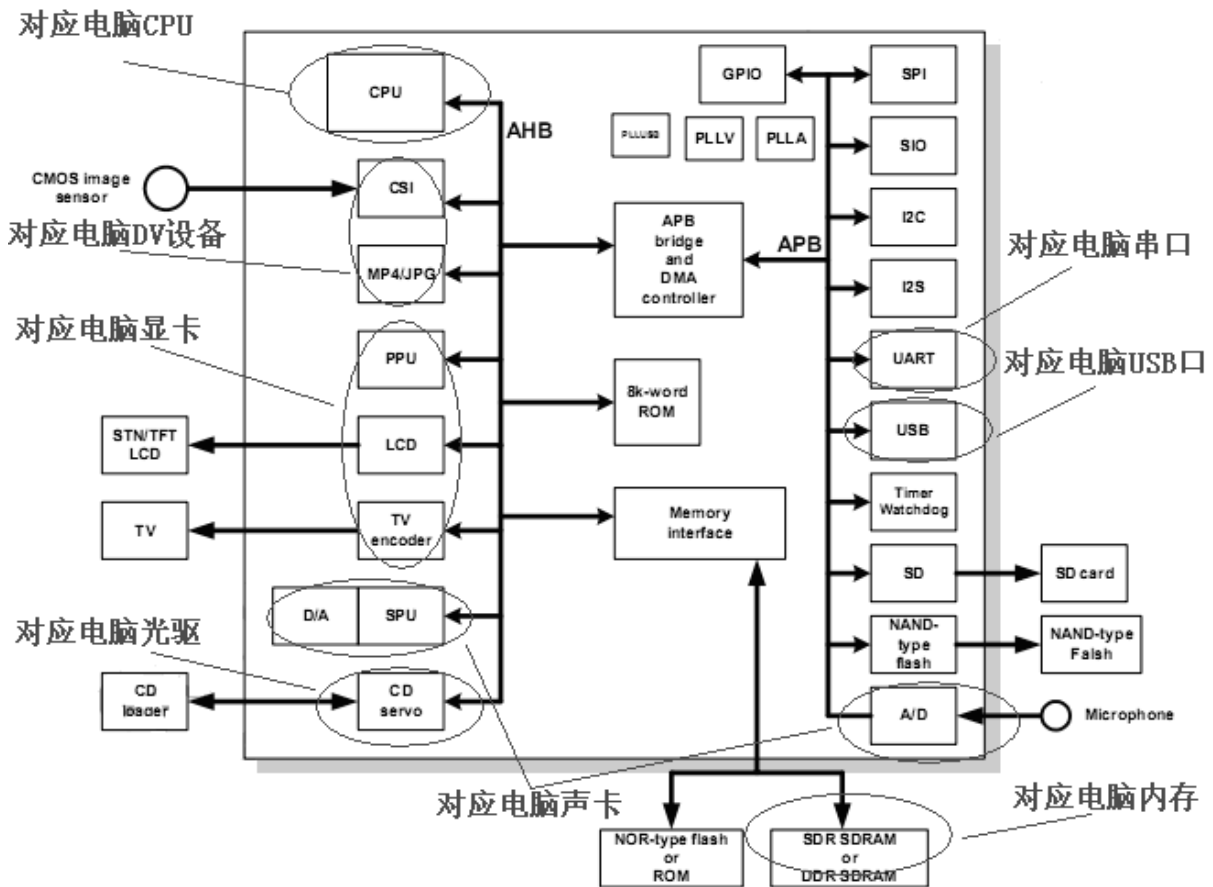


图 1.3.-2 某高端单片机构架图

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，只是性能上还有一定差距。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 振荡器呢？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恼，他们将一个叫 PLL（锁相环）的技术放到了单片机芯片内，通过该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。

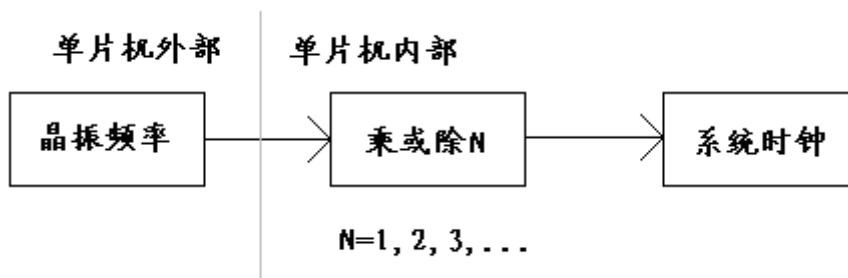


图 1.5. -1 晶振和系统时钟关系图

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，直到找与之一致的相同的代码。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 I/O 口设为指定状态，将状态直接送给 I/O 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 I/O 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		影响的 状态位	注释	
			MSb	LSb			
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
CLRWF	-	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	0xxx 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	$\overline{TO,PD}$	
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	$\overline{TO,PD}$	
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 TMRO 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 ($0 \sim 2047$)。

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处理或者不停的按异常指令处理。

如果指令系统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

CLRWF 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代码在 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	人工输入 100个	NOP	自动填充 100个	NOP
1		NOP		NOP
2		NOP		NOP
...	
98		NOP		NOP
99	NOP	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。

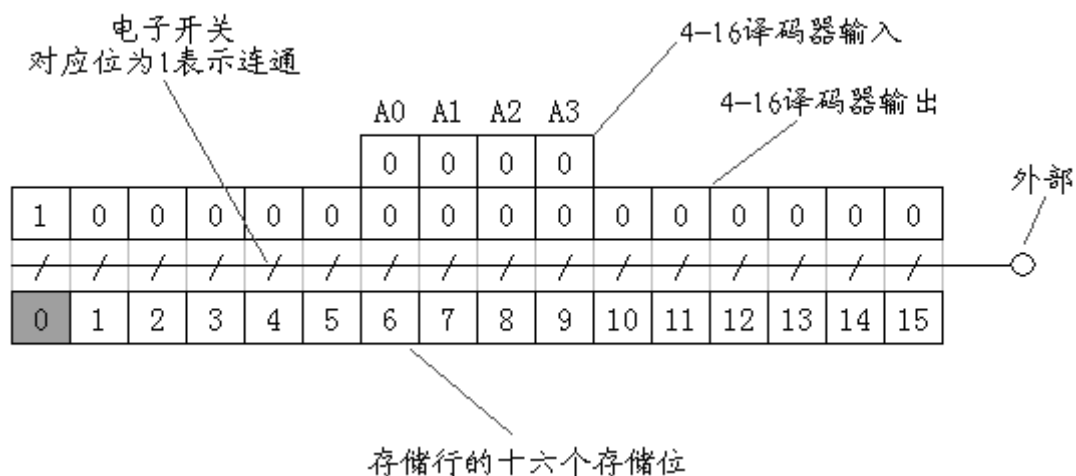


图 1.7. -1 十六位宽存储行选通示意图

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。

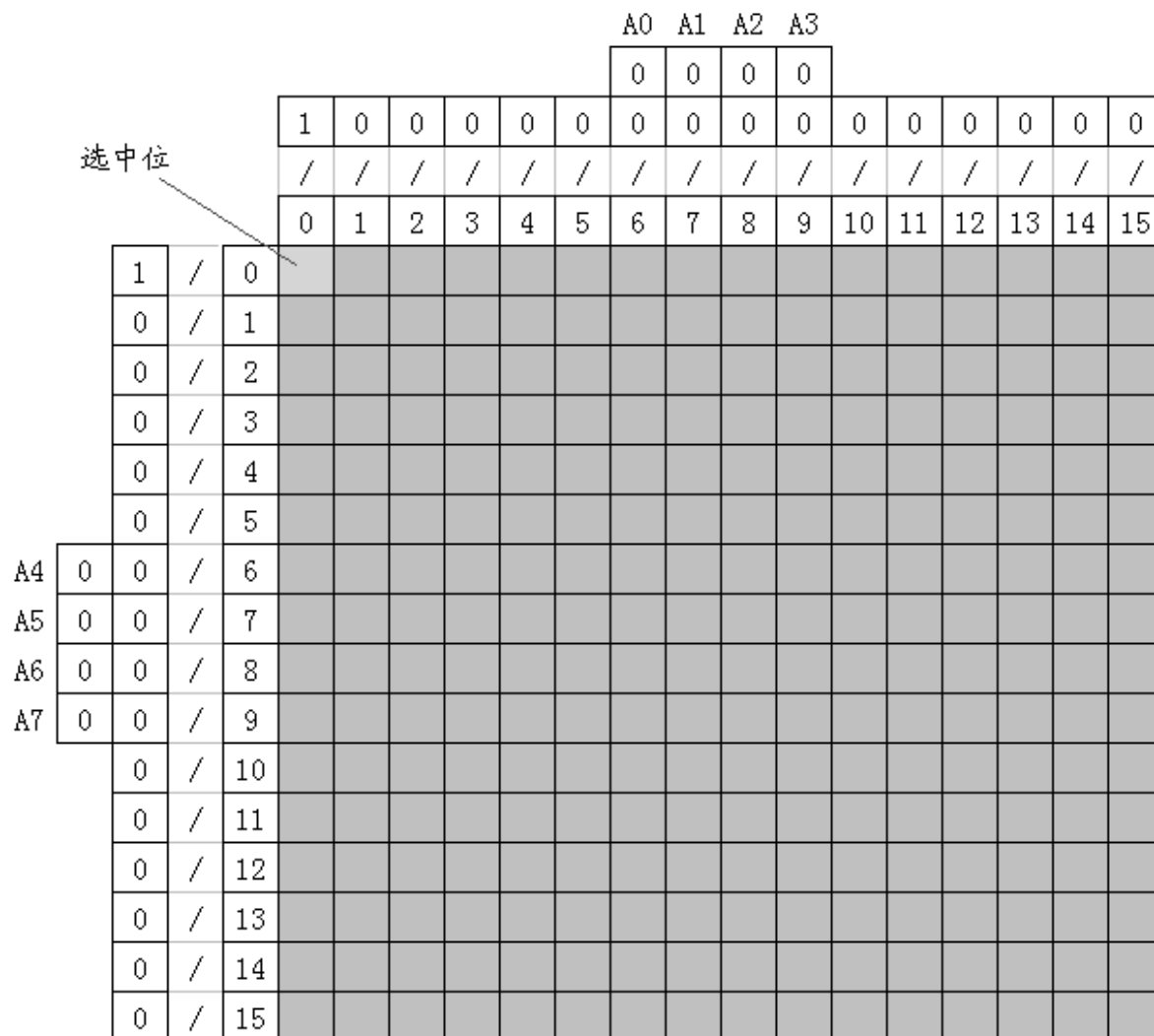


图 1.7.-2 16*16 存储阵列选通示意图

将这个原理实现的存储行进行扩展，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。

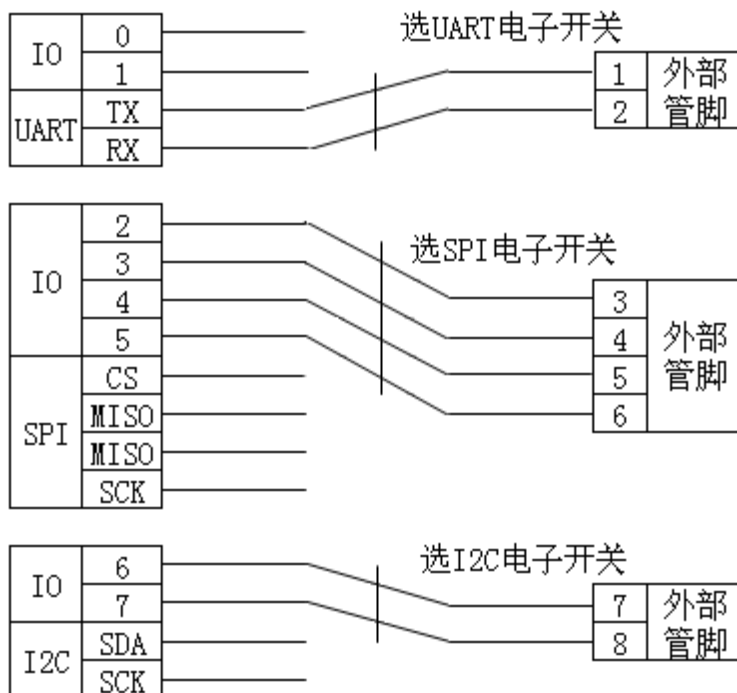


图 1.8. -1 IO 复用示意图

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。

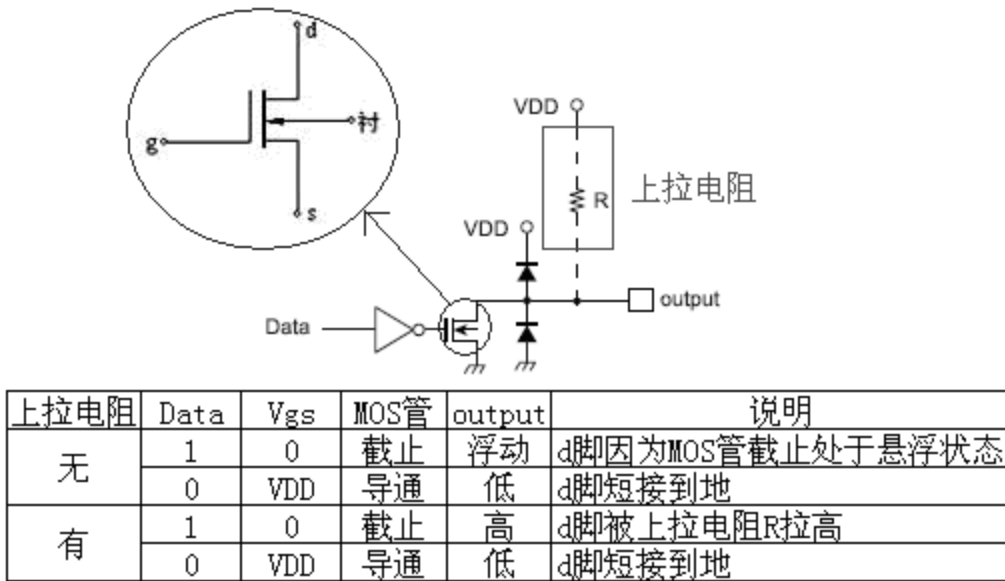


图 1.8.-2 Open Drain 输出示意图

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→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。

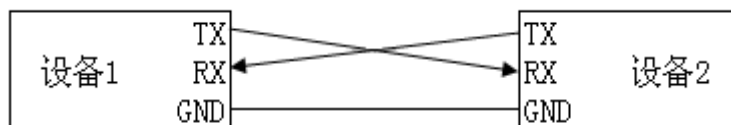


图 1.8.-3 UART 通讯连接图

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。

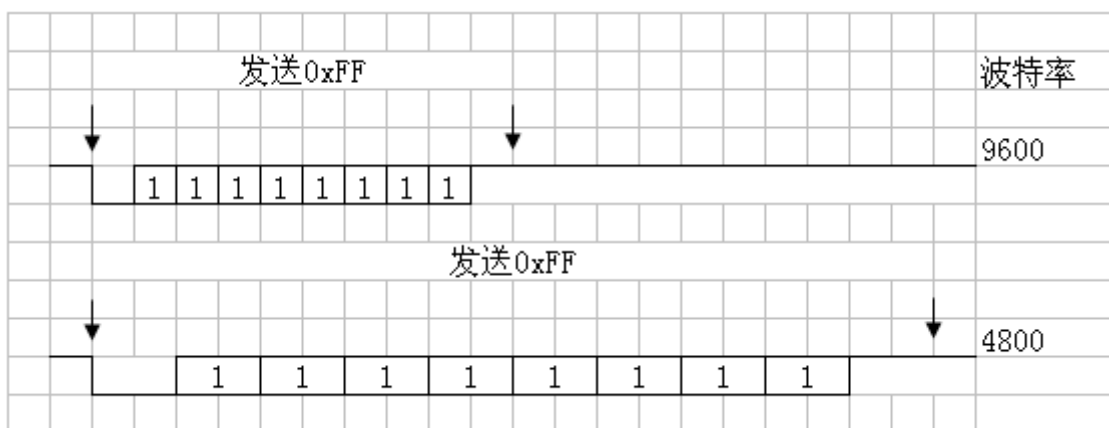


图 1.8. -6 UART 发送 0xFF 波形图

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。

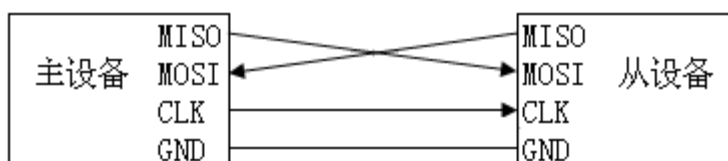


图 1.8. -7 SPI 通讯连接图

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。

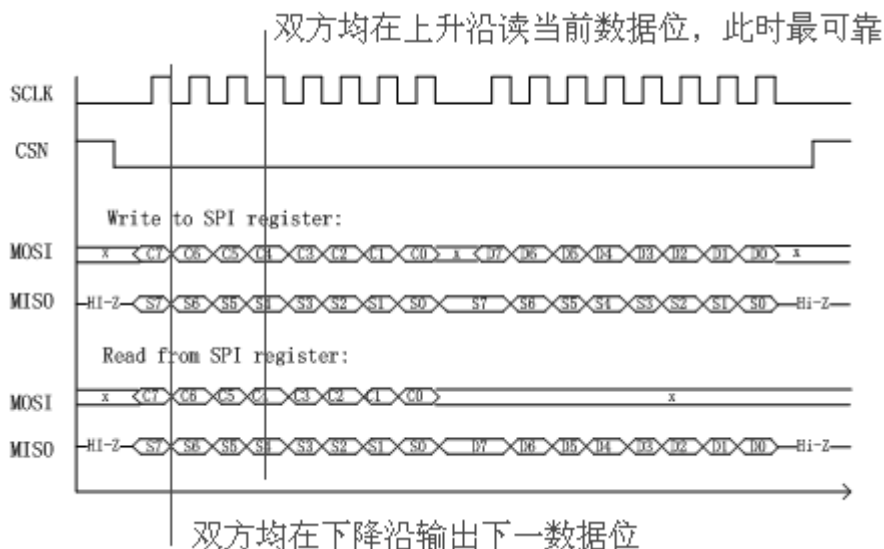


图 1.8.-8 SPI 通讯时序图

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，不会出现前面异步串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 SI0、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 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。

I/O 接口模拟驱动能力

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

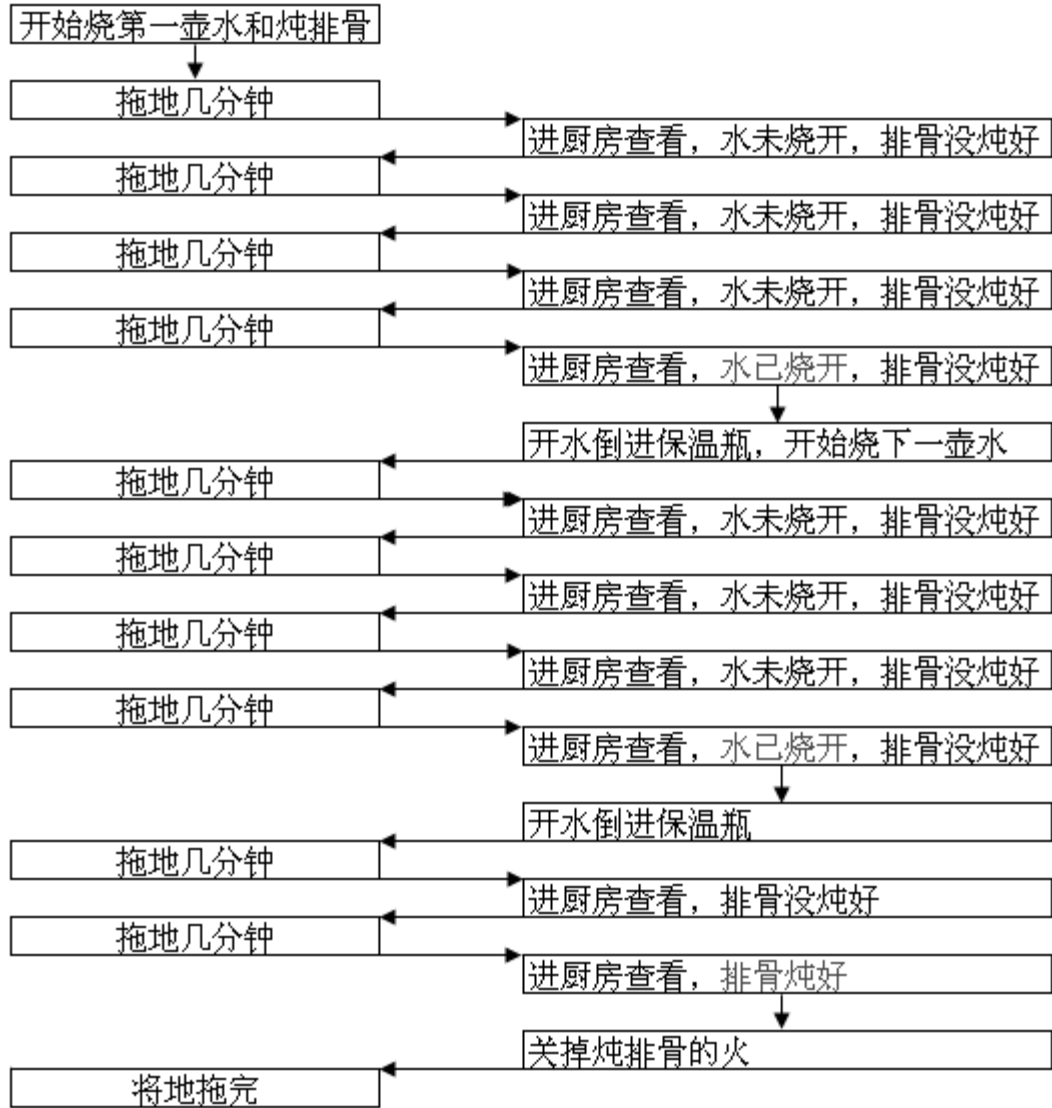


图 1.10.-1 丁丁第一周家务流程图

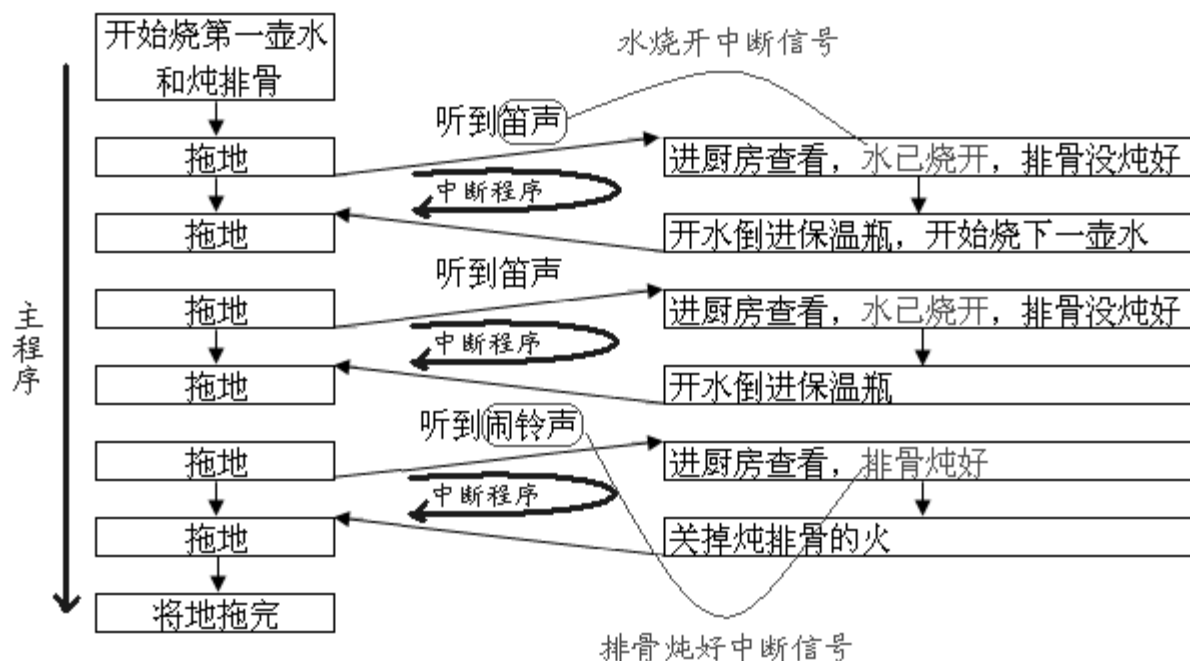


图 1.10.-2 丁丁第二周家务流程图

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 I/O 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：I/O 口输入为高、I/O 口输入为低、I/O 口输入由高变为低、I/O 口输入由低变为高、I/O 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，外设将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。

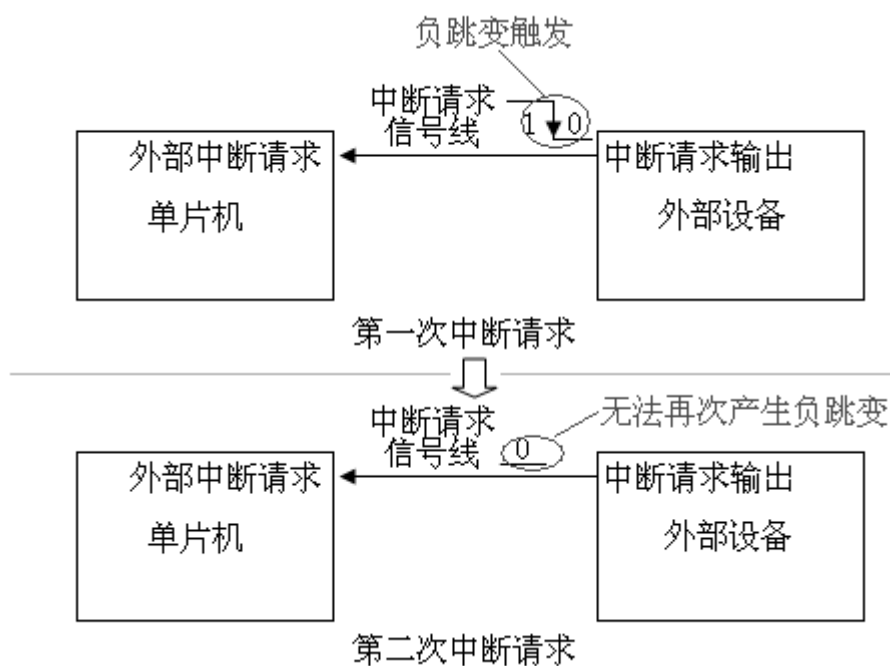


图 1.10.-3 外设只能产生一次中断请求信号示意图

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。

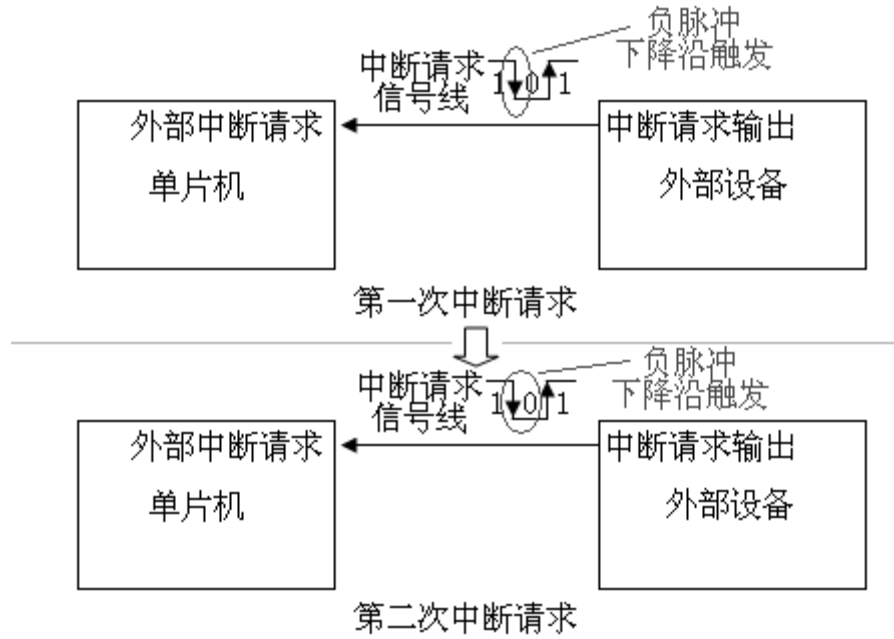


图 1.10.-4 外设可连续产生中断请求信号示意图一

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。

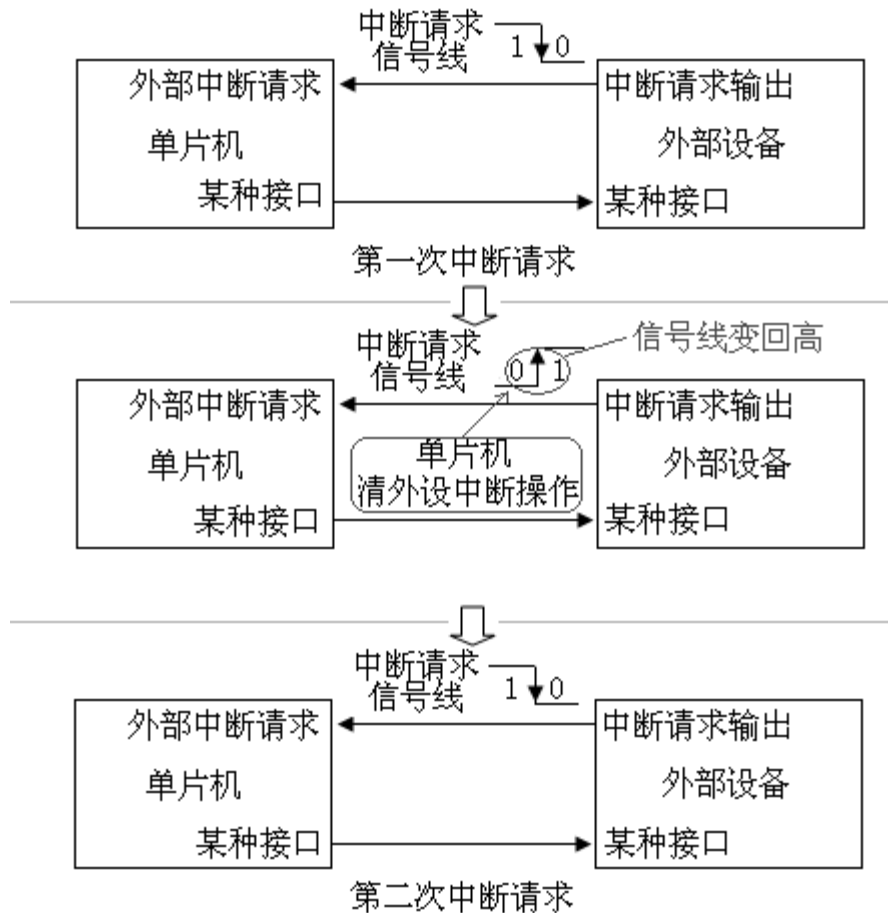


图 1.10.-5 外设可连续产生中断请求信号示意图二

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。

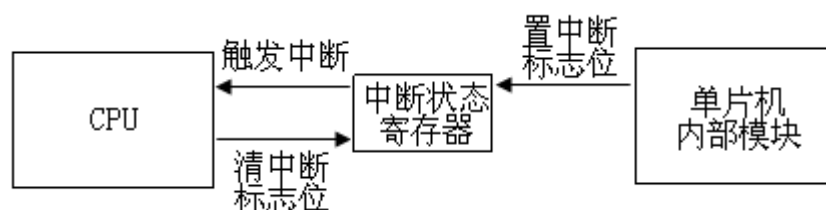


图 1.10.-6 内部中断触发示意图

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清除法，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主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。

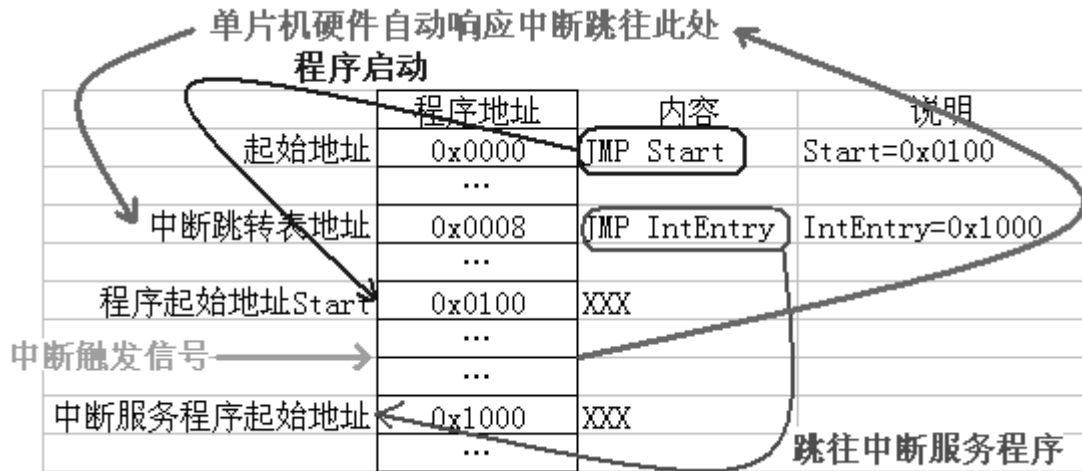


图 1.10.-7 中断响应示意图

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。

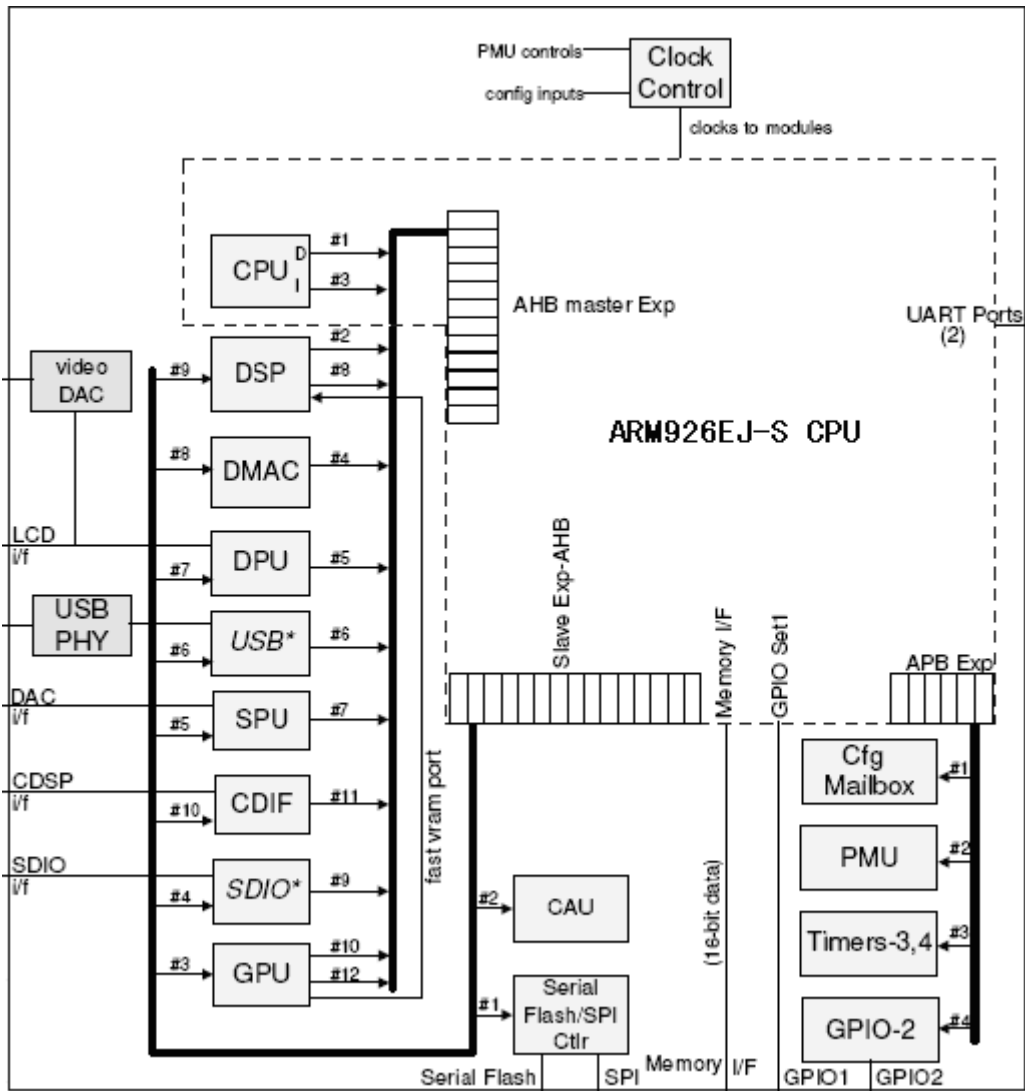


图 1.10.-8 ARM 内核单片机架构图

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。

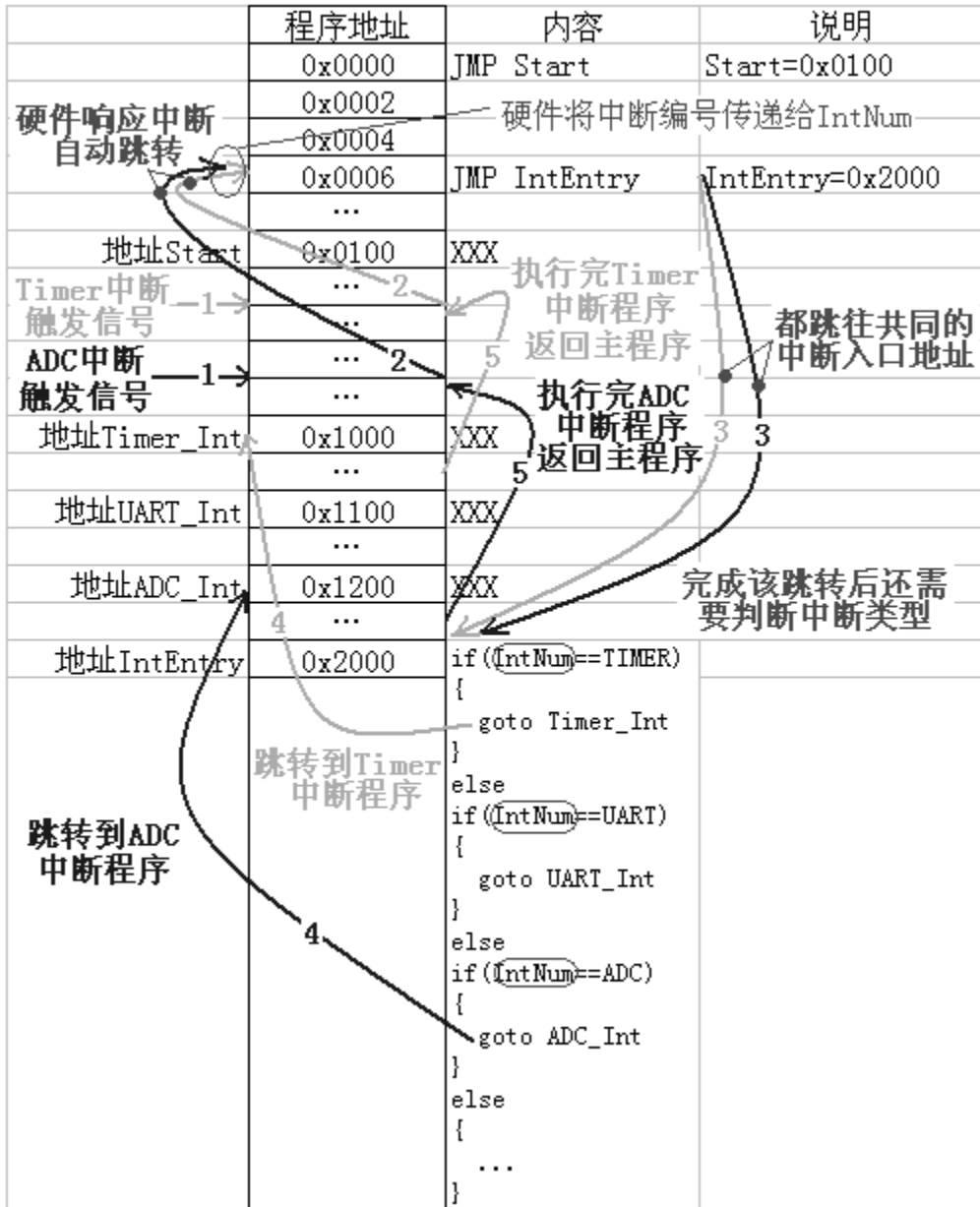


图 1.10.-9 公用中断入口中断响应流程图

	程序地址	内容	说明
	0x0000	JMP Start	Start=0x0100
	0x0002		
	0x0004		硬件已经自动判定中断类型
硬件响应中断 自动跳转	0x0006	JMP Timer_Int	Timer_Int=0x1000
	0x0008	JMP UART_Int	UART_Int=0x1100
硬件响应中断 自动跳转	0x000A	JMP ADC_Int	ADC_Int=0x1200
	...		
地址Start	2. 0x0100	XXX	Timer中断 返回主程序
Timer中断-1 触发信号	...		
ADC中断 触发信号-1	...		
地址Timer_Int	0x1000	XXX	完成ADC中断 返回主程序
	...		
地址UART_Int	0x1100	XXX	直接跳转到 Timer中断程序
	...		
地址ADC_Int	0x1200	XXX	直接跳转到 ADC中断程序
	...		

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

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。

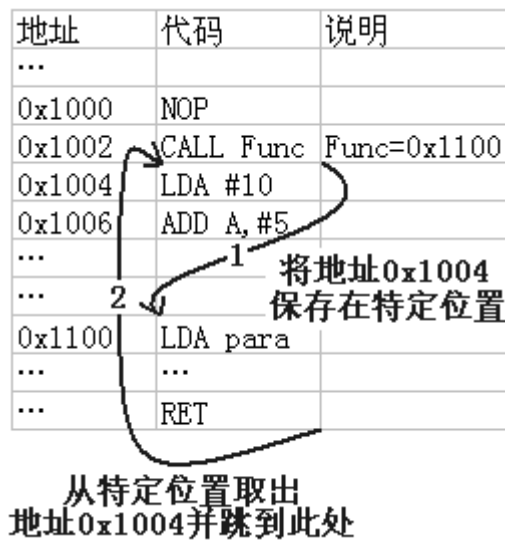


图 1.11.-1 函数执行示意图

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。

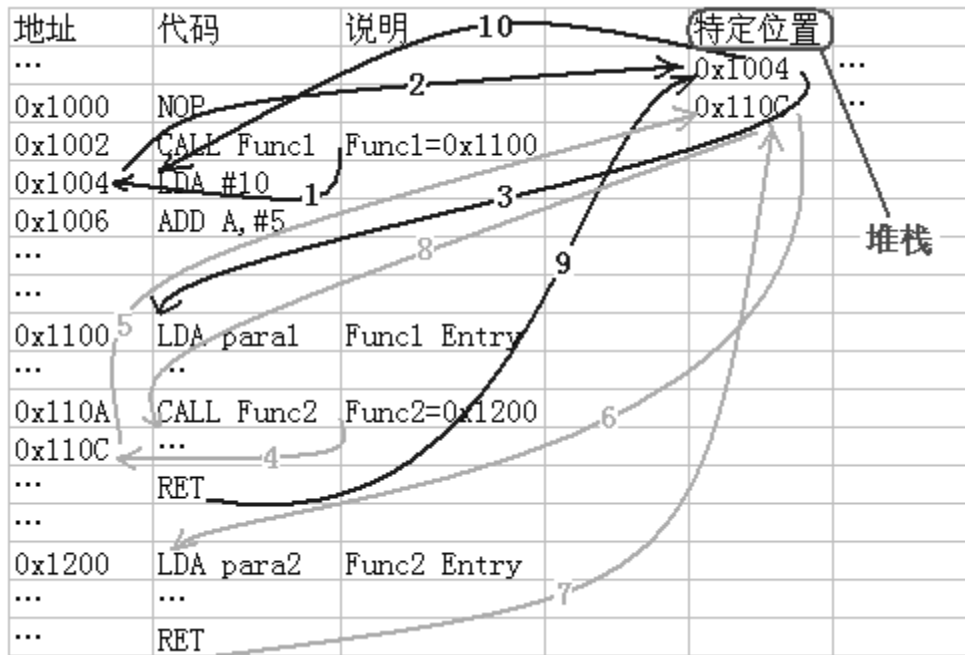


图 1.11.-2 调用函数堆栈存取示意图

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。

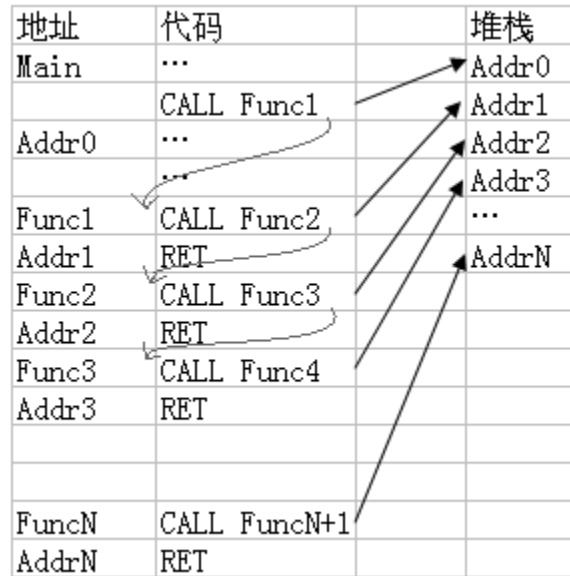


图 1.11.-5 函数堆栈溢出示意图

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。

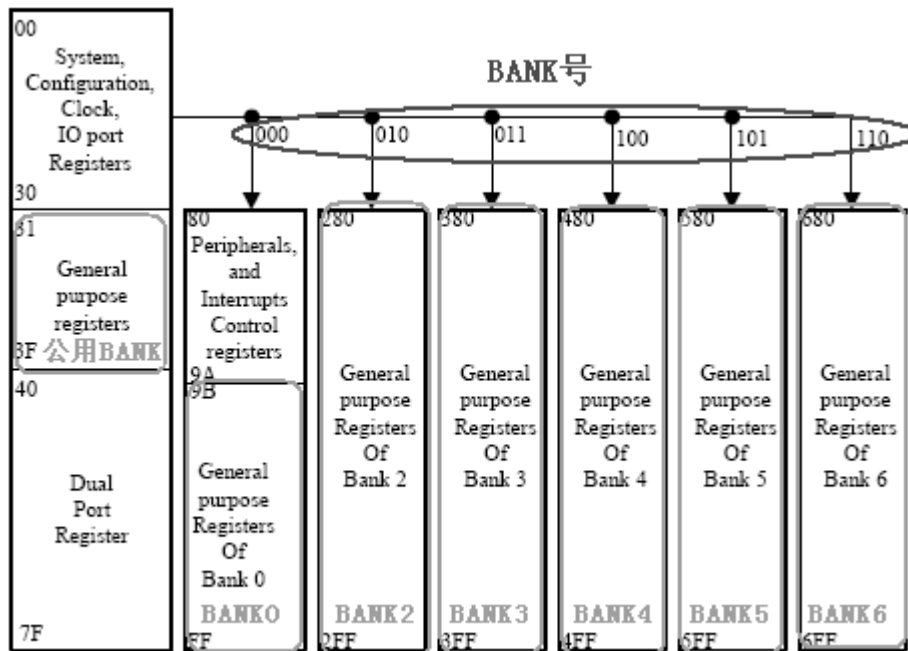


Fig. 6-2 of Data Memory (RAM) Configuration

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 0x00~0xFF，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。

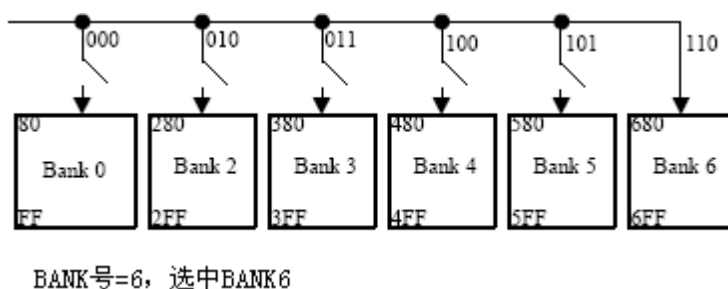


图 1.12.-4 BANK 选择示意图

- 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，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 and 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 and 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陷，如果产品生产出来后发现有问题，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问题都是不好的，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，然后开膛破肚重新烧写程序，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，不使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孔，然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，再人工将元件放上去，经过回流焊设备就完成了焊接；波峰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。