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。

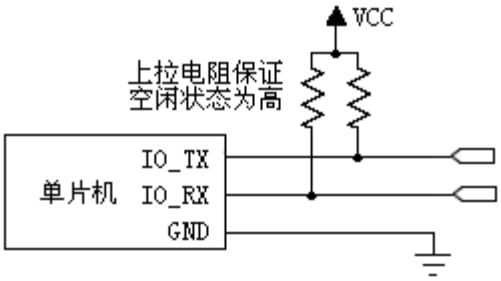

图 2.1.-1　IO 模拟 UART 示意图

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。

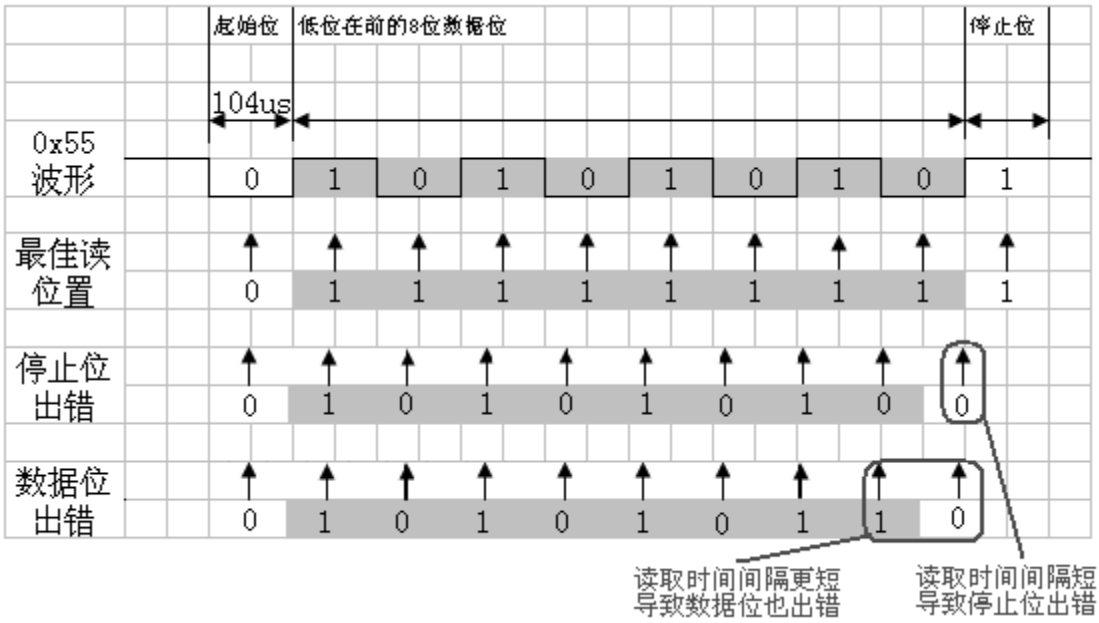

图 2.1.-2 UART 读数据位置示意图

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。

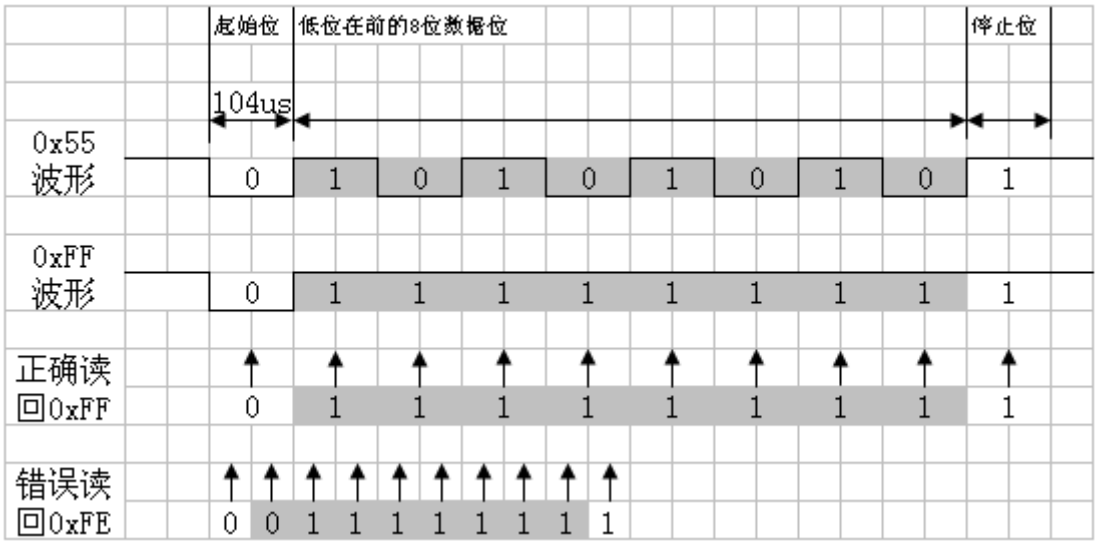

图 2.1.-3 UART 读数据出错示意图

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。

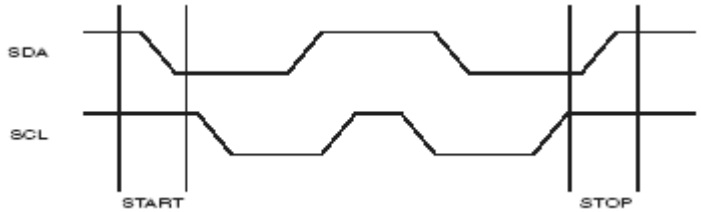

图 2.1.-4　I2C 接口 START 和 STOP 信号示意图

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。

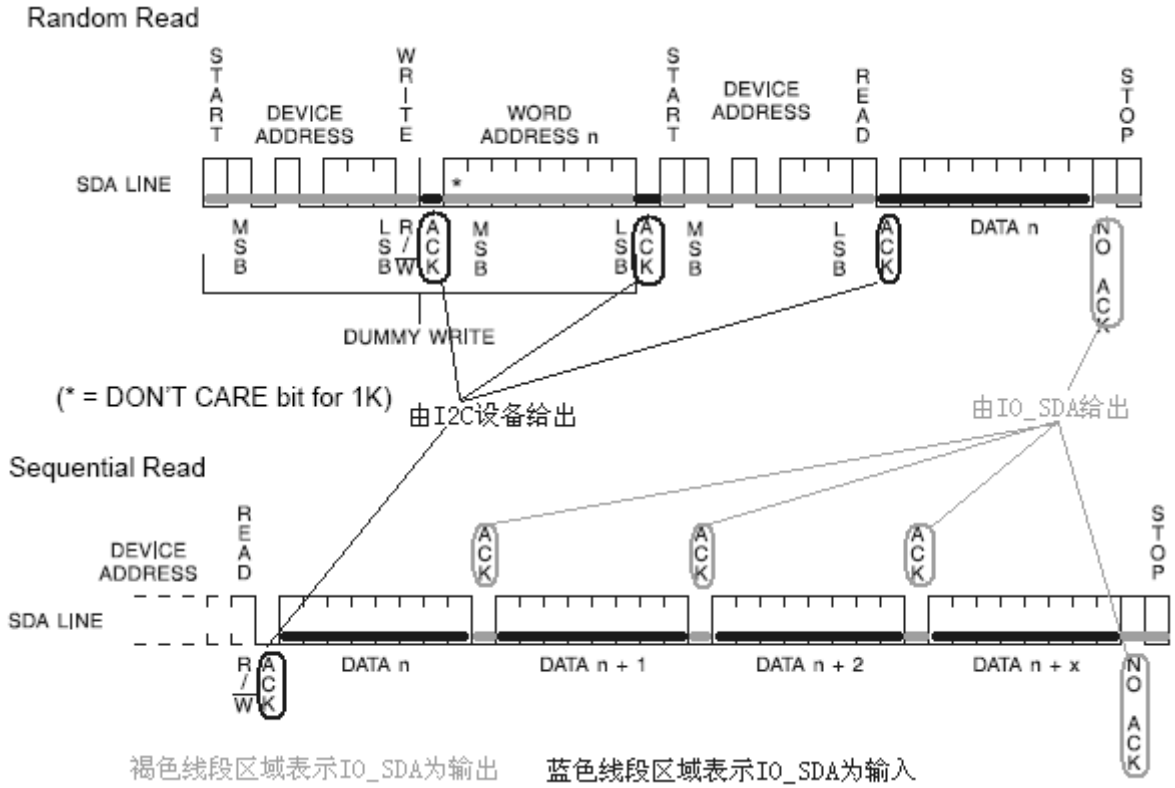

图 2.1.-5　I2C 时序图

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。

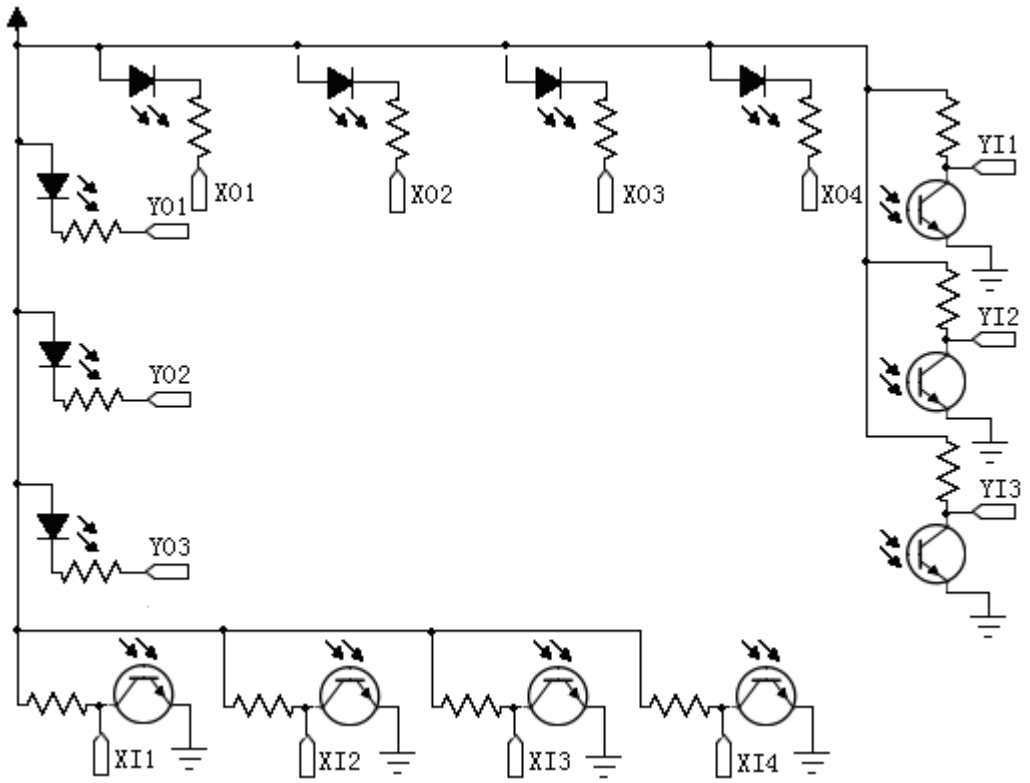

图 2.2.-1　红外矩阵示意图

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。

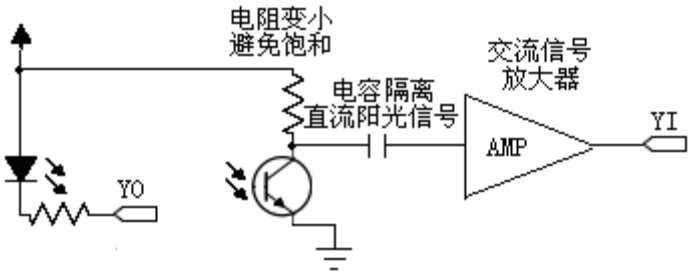

图 2.2.-2　红外信号处理示意图

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

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

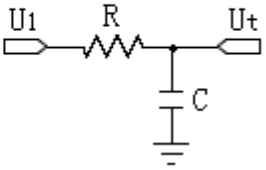

图 2.4.-1 电阻电容充放电示意图

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))。

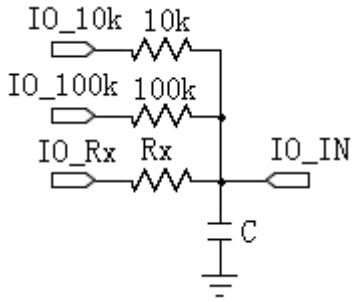

图 2.4.-2 电容效应测电阻示意图

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？

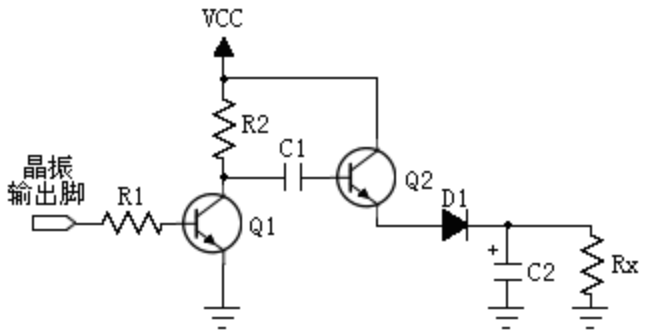

图 2.5.-1　晶振控制电源示意图

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。

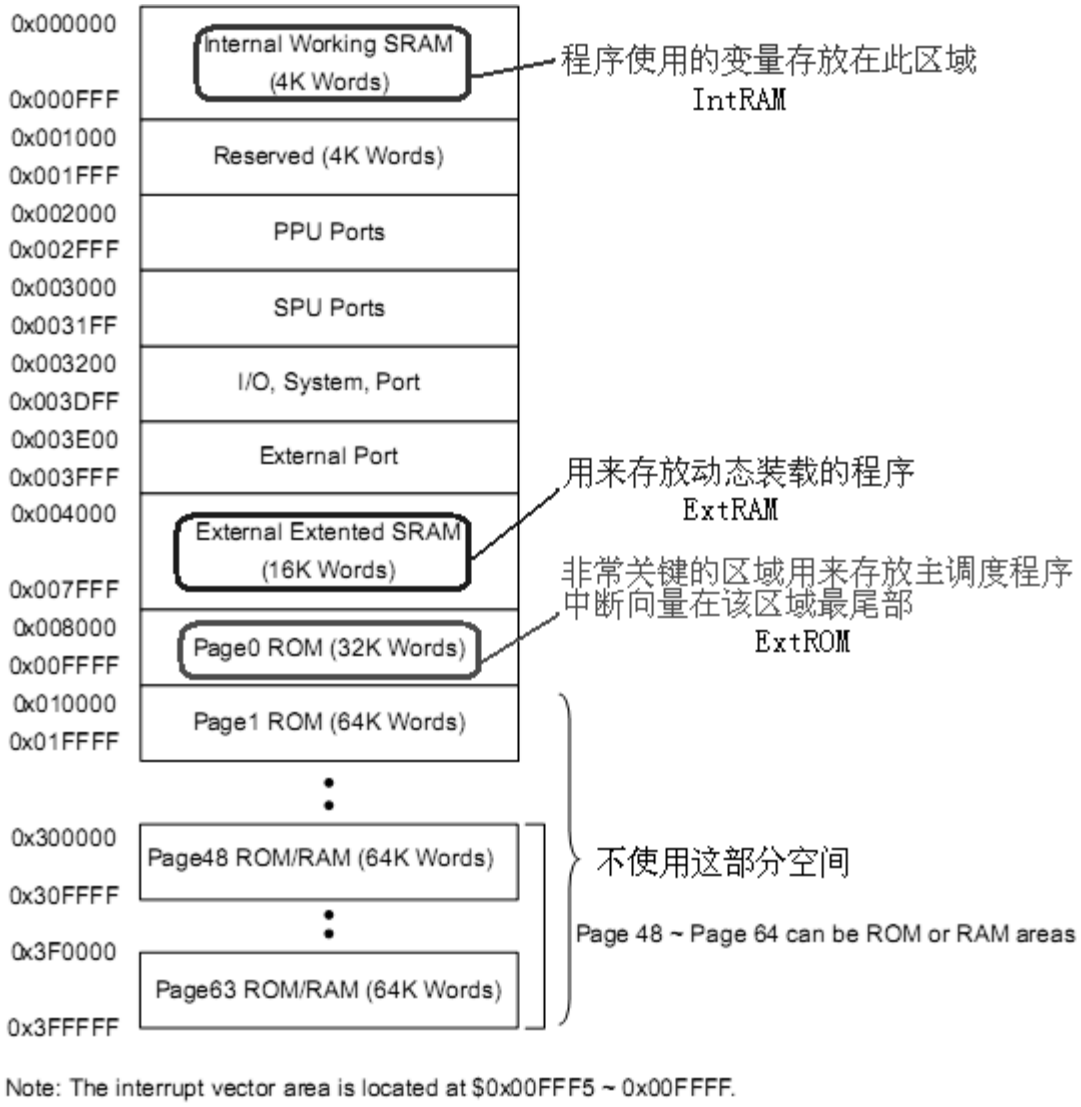

图 2.9.-1　样例 MCU 内存分布示意图

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。

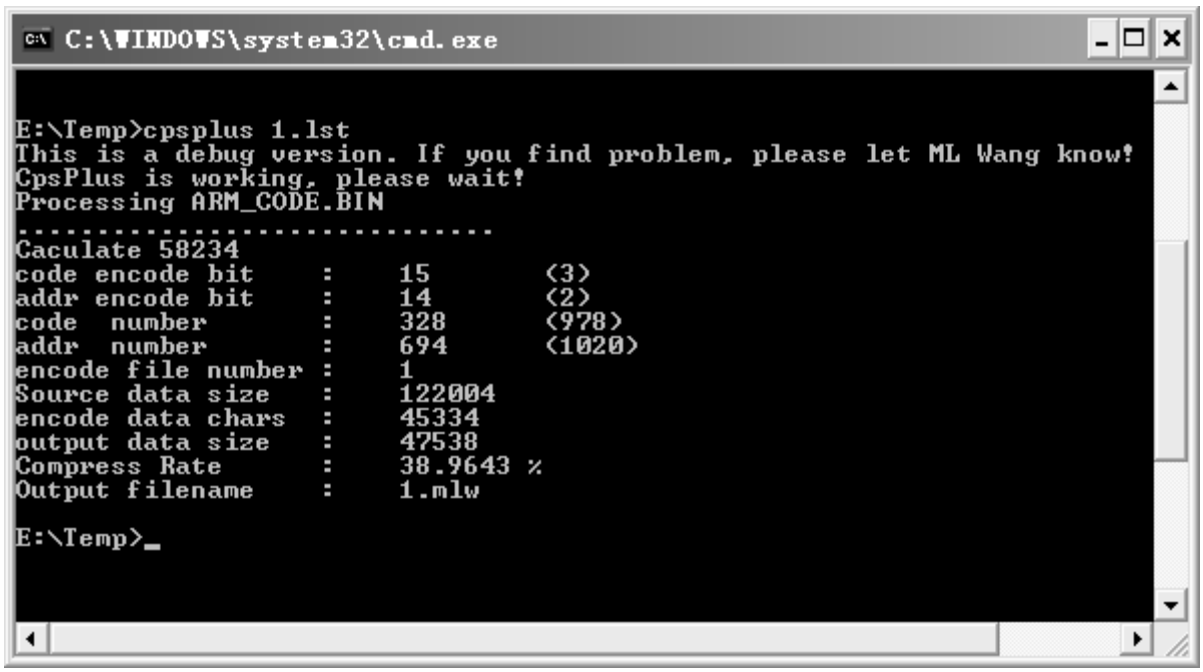

图 2.10.-2 电脑模拟压缩结果图

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。

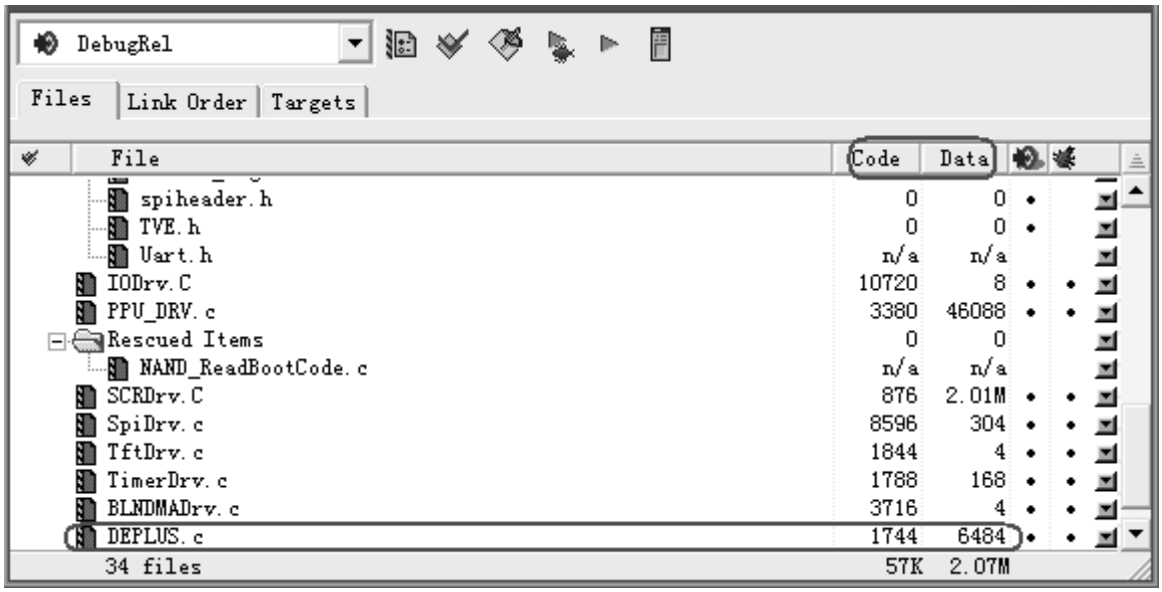

图 2.10.-3　解压程序耗用 ARM 资源示意图

●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。

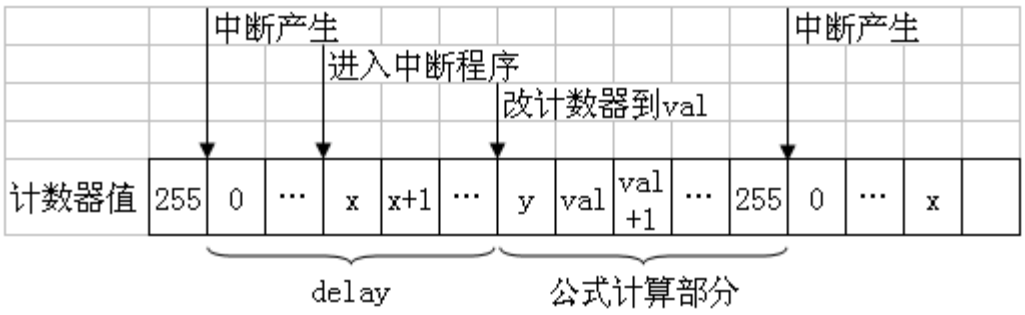

图 2.12.-1　定时中断校正示意图

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。

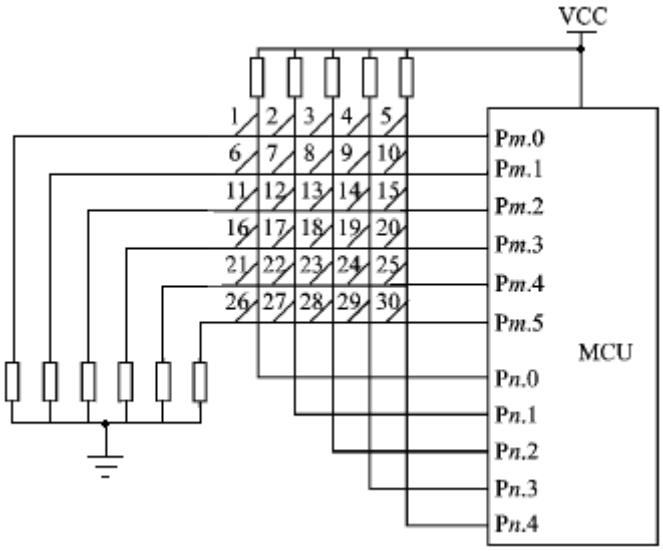

图 2.15.-1　单片机键盘矩阵示意图

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。

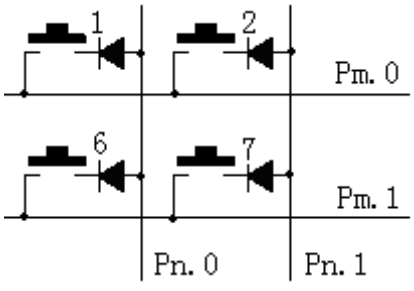

图 2.15.-2　键盘矩阵二极管保护示意图

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。

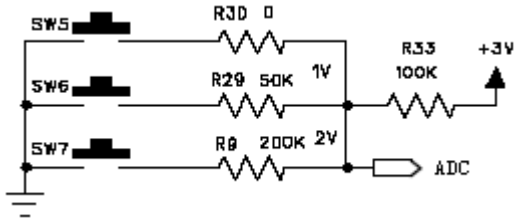

图 2.15.-3　电阻分压键盘示意图

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。

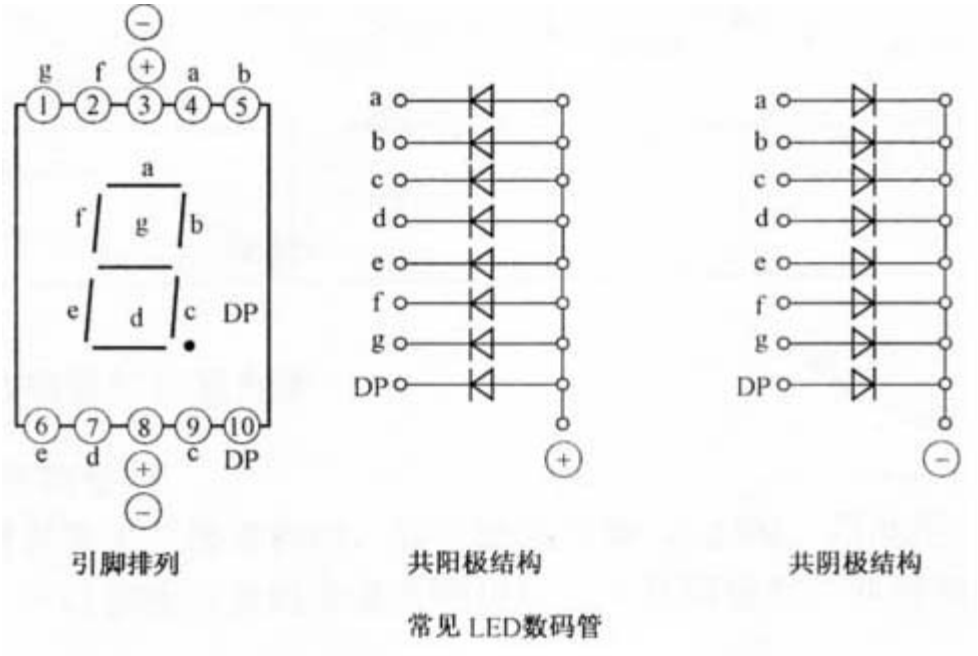

图 2.16.-1　数码管示意图

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。

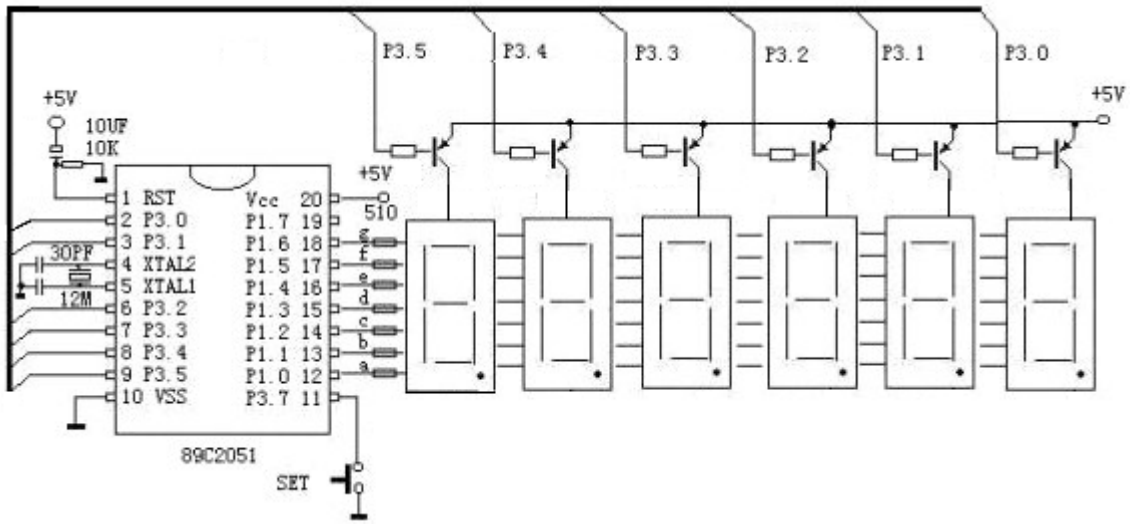

图 2.16.-2　数码管利用视觉暂留应用示意图

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。

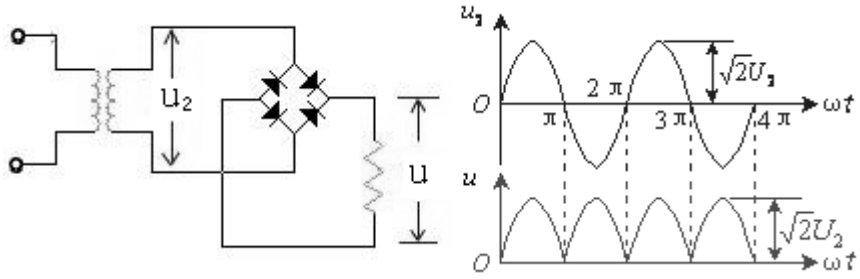

图 2.19.-1　交直流整流示意图

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。

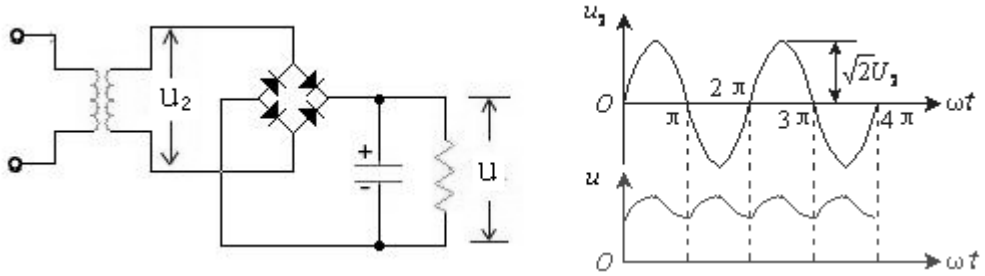

图 2.19.-2　带滤波电容交直流整流示意图

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。

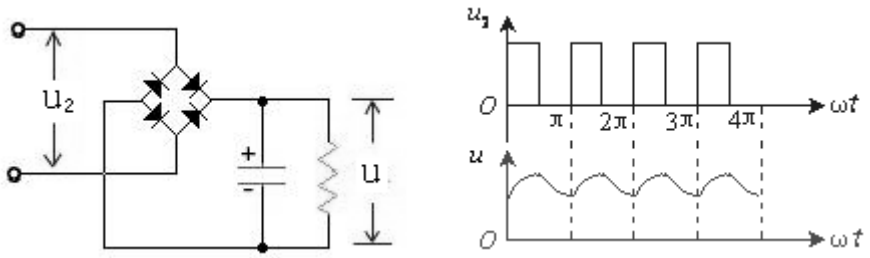

图 2.19.-3　方波整流示意图

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

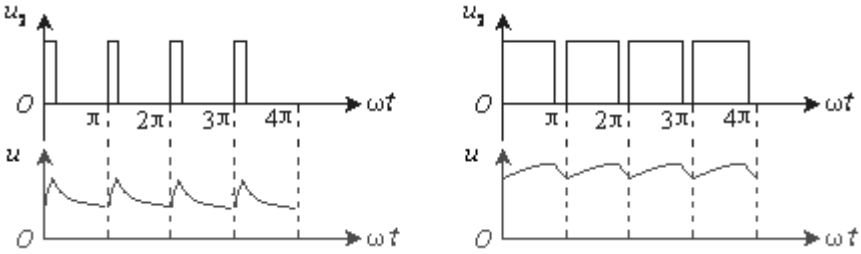

图 2.19.-4　PWM 效果示意图

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。