第三章　单片机高级特性

时至今日，单片机的技术已经发展到前所未有的地步，PC 流行大旗刚刚树起的九十年代，主频终于突破 100MHz，简称 586 的奔腾一代开始用软解压向人们结结巴巴的演示多媒体的未来，就是 INTEL 自己也为这一进步激动不已，从此电视广告中“Beng Beng Beng Beng”的旋律成为 INTEL 的象征。

让我们来看看当时让 INTEL 如此激动的奔腾电脑的摸样：

1996 年

100M 主频 Intel Pentuim CPU

16M 内存

1M 显存显卡

850M 硬盘

14 寸彩显

大概需要 8000~10000 人民币

再来看一看现在 iPhone 使用的三星 64xx 的 MCU（以某开发板为例）：

Samsung S3C6410，ARM1176JZF-S 内核，主频 533MHz/667MHz

128M Bytes DDR RAM

256M Bytes NAND Flash

2M Bytes NOR FLASH

100Mbps 以太网接口

USB HOST 接口

USB Device 接口

AC97 接口

双高速 SD 卡接口

双 LCD 接口

VGA 接口

TV OUT 接口

S-VIDEO 接口

双摄像头接口

2D/3D 硬件加速

带 800*480 的低成本液晶屏开发板成本大约为 300~400 人民币

只要简单对比就可以知道今天的高端单片机在性能方面已经远超当年的奔腾电脑，单片机要发展到这一步肯定不能拘泥在早期单片机技术的框架当中，需要不断引入一系列新技术，这些技术有可能是早期电脑才能采用“昂贵”技术，随着技术的不断进步才逐渐平民化为单片机所用，这一章让我们来一起了解单片机的这些高级技术。

本章的内容如果你看不明白并不要紧，你糊里糊涂的看就行了，知道有这么回事，等到有一天你面对这些技术时突然有恍然大悟的感觉时再回来与你的理想做对比。有告诉你一个秘密，这一章中的内容其实我自己也不大明白，就是许多专家也不完全明白。

3.1. Cache

首先得清楚什么 Cache，Cache 是英文中对高速缓存系统的称谓，Cache 的概念在硬件和软件中都存在。这里我借鉴别人的一个例子来解释 Cache 的作用：软件高速缓存的作用产生于人们使用数据不平均时，我们虽然常常拥有大量数据，但最经常使用的往往只有其中一小部分。如国标汉字不到 7000 个，但经常使用的只有 2000~3000 个，其中几百个又占了 50%以上的使用频率，如果将这几百个放到存取最快的地方，就可以用很小的代价大大提高工作速度。我们知道内存的存取速度比硬盘快得多，程序一起启动我们就将常用几百个字模装入内存指定区域，当使用这部分字的时候直接从内存取字，其余的才会去读硬盘。我们知道内存的读取速度为硬盘的数万倍，假设我们有一本书需要显示，预装几百个字模到内存指定区域的方法差不多将平均读取速度提高一倍，如果将预装的字模数增加到常用 2000~3000 个，读取速度甚至可以提高十倍。

这里我们要说的 Cache 是指一种用来加速存储器读写操作的硬件存储器，象买电脑时常说的一级高速缓存/二级高速缓存就是 CPU 内部的这种硬件存储器，和软件高速缓存比虽然是两种不同的方式，但其作用是一样的，都是为了提高读写速度。

可能有人会有这样的疑问，明明 RAM 已经是一种存取速度非常快的硬件，为什么还需要 Cache 呢，是的现在的 RAM 可以提供超过 100M 的读写速率，但这个速度同 CPU 的处理速度相比并不存在优势，甚至远小于 CPU 的处理速度，象 S3C6410 工作在 667MHz 主频下，就是一条需要 4 个周期的指令执行也只需要 6ns，而 RAM 的读写时间需要 10ns，显然 RAM 的速度无法满足 CPU 的高速处理要求。

如何解决 CPU 与 RAM 之间的这种速度差异问题？通常有下列方法：

一、在基本总线周期中插入等待，当 CPU 需要读写 RAM 数据的时候，先向 RAM 发送读写命令，再等待 RAM 处理好总线数据完成一些读写操作，这样做显然会浪费 CPU 的能力，就象我们设计了速度都可以达到 120 公里/小时的汽车和高速公路，可高速公路每隔十公里就设置一个收费站，这样再好的汽车也无法跑出快的速度来。

二、采用存取时间较快的 SRAM 或其它新型存储器材作存储器，这样虽然解决了 CPU 与存储器间速度不匹配的问题，但却大幅提升了系统成本。另外还有一个问题，大容量 RAM 作为外部器件，需要通过外部连线将其与 CPU 连接起来，这些外部连线因为分布电容等问题使得 RAM 与 CPU 之间的

最高传输速率有限制，如果对 PCB 布板要求过高不利于生产推广，这个问题同样可以用高速公路的例子来理解，汽车的速度可以继续提高，收费站也可以撤掉，但实际生活中高速公路不可能设计成笔直宽阔的大道，所以还是不能满足汽车速度的需求。

三、在慢速的 RAM 和快速 CPU 之间插入一速度较快、容量较小的 SRAM，起到缓冲作用，使 CPU 既可以以较快速度存取 RAM 中的数据，又不使系统成本上升过高，这就是 Cache 法。目前，一般采用这种方法，它是在不大增加成本的前提下，使 CPU 性能提升的一个非常有效的技术。

当然 Cache 的实现并不是简单的插入一块小容量高速存储器那么简单，是基于程序统计规律通过一系列复杂控制技术才得以实现，而且它并不是万能的，同样存在缺陷，后面我们会详细讲述这些细节。

先看一下 ARM 关于存储器的结构图。

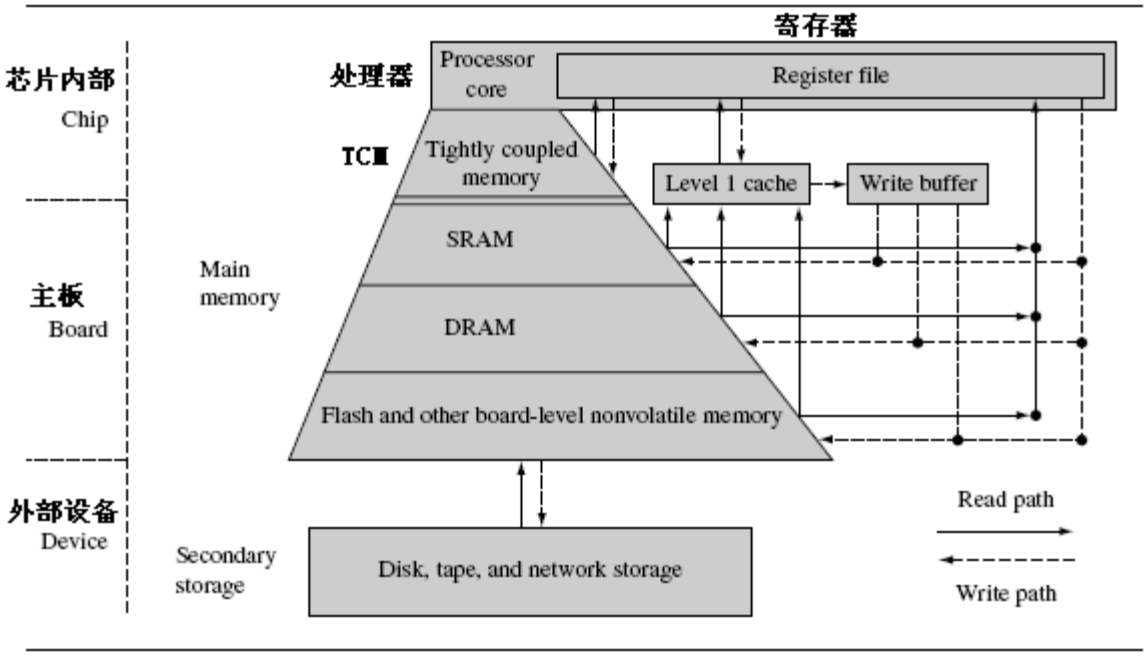

图 3.1.-1　ARM 存储器示意图

TCM（摘自 ARM 论坛）：

TCM 是一个固定大小的 RAM，紧密地耦合至处理器内核，提供与 cache 相当的性能，相比于 cache 的优点是，程序代码可以精确地控制什么函数或代码放在那儿(RAM 里)。当然 TCM 永远不会被踢出主存储器，因此，他会有一个被用户预设的性能，而不是象 cache 那样是统计特性的性能提高。

TCM 对于以下几种情况的代码是非常有用、也是需要的：可预见的实时处理（中断处理）、时间可预见（加密算法）、避免 cache 分析（加密算法）、或者只是要求高性能的代码（编解码功能）。随着 cache 大小的增加以及总线性能的规模，TCM 将会变得越来越不重要，但是他提供了一个让你权衡的机会。

那么，哪一个更好呢？他取决于你的应用。Cache 是一个通用目的的加速器，他会加速你的所

有代码,而不依赖于存储方式。TCM 只会加速你有意放入 TCM 的代码,其余的其他代码只能通过 cache 加速。Cache 是一个通用目的解决方案，TCM 在某些特殊情况下是非常有用的。假如你不认为需要 TCM 的话，那么你可能就不需要了，转而加大你的 cache，从而加速运行于内核上的所有软件代码。

可以看出 Cache 位于芯片内部，通过内部总线与 CPU 相连，图中自上而下的存储器离处理器越远读写速度就越慢，Cache 本质也是 SRAM，只是对其增加了一些特殊的读写控制方法。同样是 SRAM，片外的 SRAM 速度比片内要慢，这就是外部总线的影响。Cache 是不能独立当作存储器使用的，对于程序员来说，它并没有特定的地址可以进行访问，只是处理器提供了一些控制指令可以让程序员对 Cache 进行控制方法的设定，所以为了针对某些特殊应用芯片厂商会在芯片内部另外会放置一小段 SRAM，这段 SRAM 对于程序员来说就有特定的地址与之对应，程序可以当作普通 RAM 进行读写。

Cache 的工作原理

通常程序代码都是连续的，代码执行时都是一条接一条的连续执行，程序中跳转操作所占的比例并不高，即便是跳转指令，大多数时候跳转的距离都不会太远，加上指令地址的分布本来就是连续的，另外象程序中的循环体要重复执行多次，这样在一个较短的时间间隔内，由程序产生的地址往往集中在存储器地址空间的很小范围之内，因此对这些地址的访问就自然地具有时间上集中分布的倾向，对大量典型程序运行情况的统计分析结果也验证了这一点。

数据分布的这种集中倾向没有程序代码明显，程序中的数据读写操作虽然大多数时候也是处在相邻区域，但间距大过程序代码几率要高，不过数组的存储和访问还是会让存储器地址相对集中。

```
UINT32 i;
UINT32 data_buf1[1024];
UINT32 data_buf2[1024];

for(i=0;i<1024;i++)
{
  data_buf1[i]=i;
}
for(i=0;i<1024;i++)
{
  data_buf2[i]=data_buf1[i];
}
```

假定样例代码中的变量 i 和数组 data_buf1[]与 data_buf2[]分布是连续的，两段循环代码可以肯定是连续分布。

第一个循环：

```
for(i=0;i<1024;i++)              //需要读写 i
{
  data_buf1[i]=i;                //顺序写数组 data_buf1[]的每一个成员
}
```

很明显循环体的代码量非常小，执行这段代码完全满足代码地址在一小段区域之内的要求，但对数据的读写则有点不同，当 i 在 0 附近时，data_buf1[i]=i 的操作 RAM 地址间隔并不大，但当 i 逐渐增大情况就发生了变化，比如 i 为 1000 时，data_buf1[i]=i 的操作对 RAM 的操作跳转就会变得比较大，按假定条件 i 和写 data_buf1[1000]在 RAM 中的位置间隔有 4000 字节，虽然这 4000 个字节并不是特别大，但如果数组大小从 1024 变为 1024*1024 呢？间隔就会非常之大。

第二个循环：

```
for(i=0;i<1024;i++)              //需要读写 i
{
  data_buf2[i]=data_buf1[i];  //顺序读写数组 data_buf1[]和 data_buf2[]的每一个成员
}
```

和循环一相比情况数据在 RAM 的读写跳跃间隔会更大，每一次循环都要在两个数组之间切换，这样跳跃的间隔为数组的大小 4096 字节，变量 i 与数组 data_buf2[]之间的间隔全部超过 4096 字节。

Cache 的工作原理正是是基于程序访问的局部性来实现的，如果把较短时间间隔内的代码从外部 RAM 放到做为内部 Cache 的 SRAM 中执行，这段代码显然会因为不需要等待外部 RAM 的存取操作而获得更高的执行效率，但 Cache 的容量有限，只能放置少量的代码，还需要通过某种方法让整个程序都是在 Cache 中执行才有实际意义。Cache 对于程序代码的效果总体上看要优于数据。

Cache 的实现思路并不复杂，由处理器硬件不断地将与当前代码相关联的一小段后续代码从 RAM 中读到 Cache，然后再与 CPU 高速传送，从而达到速度匹配。CPU 对存储器进行数据请求（包含代码执行和数据读写）时，通常先访问 Cache。通过前面的分析我们知道由于局部性原理并不能保证所请求的数据百分之百地在 Cache 中，这里便存在一个命中率，即 CPU 在任一时刻从 Cache 中可靠获取数据的几率。

命中率越高，正确获取数据的可能性就越大，命中率和 Cache 的容量是成正比的。一般来说，Cache 的容量比 RAM 的容量小得多。从成本考虑，是 Cache 的容量越小越好，最好是不用，但 Cache，太小会使命中率太低；站在最大发挥 CPU 功效的角度，最好是能达到 100%的命中率，这样就希望 Cache 的容量尽可能的大，但过大不但会增加成本，因为命中率和容量之间的关系不是线性比例关系，当容量超过一定值后，命中率随容量的增加将会变得不明显。

只要 Cache 的空间与 RAM 空间在一定范围内保持适当比例的映射关系，是可以保证 Cache 的命中率达到一个比较高的比例。统计分析的结论告诉我们，当 Cache 与 RAM 的空间比例关系在 1：256 时，即 4kBytes Cache 映射 1Mbbytes RAM 既能得到较高的命中率，又不至于因为 Cache 导致成本上升太多。在这种情况下，命中率可以达到 90%以上，至于没有命中的数据，CPU 只好直接从内存

获取。获取的同时，也把它拷进 Cache，以备下次访问。

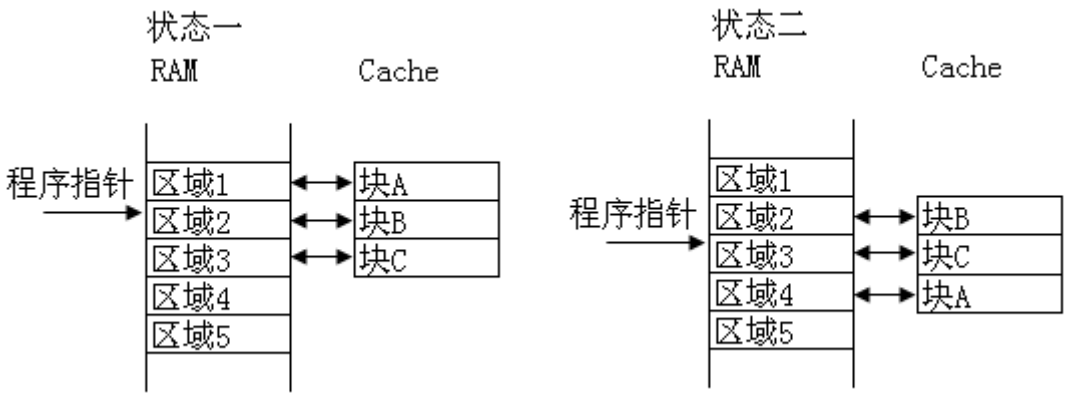

图 3.1.-2　Cache 映射示意图

图示为一种简单的 Cache 控制方式，CPU 从 RAM 存取数据（包括程序代码和程序数据）时均通过 Cache 进行，Cache 分成三块，块 B 对应当前程序指针所在的区域，块 A 和块 C 则分别对应其前后的相邻区域。通常程序不会产生大距离的跳转，所以程序大部分时候下一条指令所在位置都会位于这三个区域之中，当程序运行到程序指针会换区时，硬件会自动更新 Cache 的映射状态，这样就能继续维持程序指针始终指向 Cache 中间块。图示状态一到状态二程序指针从 RAM 的区域 2 换到区域 3，此时硬件自动将块 A 原本映射区域 1 的内容释放给区域 1，改为装载区域 4 的内容并建立映射关系，虽然这个更新过程同外部 RAM 同步数据需要一定时间，当前程序指针所指向的块自己有一定空间可以耗费程序一段时间以实现更新，当 CPU 需要区域 4 中的内容时，大部分时候 Cache 的更新已经完成。

这种简单的 Cache 控制方法在效果方面不是特别理想，对于 CPU 存取数据的地址跳转情况没有做出一个比较好的应对方法，对于一些特殊的程序模式命中率会比较低。

```
if(flag==1)
{
  func();
}
flag=0;
```

代码中的条件执行效果就会不怎么好，程序会依据 flag 的值选择是继续执行还是调用函数 func()，一般来说函数入口地址与当前所执行代码的位置间距都不会小，也就是说对于程序选择执行函数 func()的情况命中率很大机会不够理想，差的时候甚至是 0。

对于这种效果不理想的情况并非没有应对良策，正常编写的程序一定满足局部集中性的规律，

除非写出全是跳转的变态程序，如果将 Cache 的分块分得更为精细，每一块的空间尽可能的变小，所映射的区域不要求必须连续，而是通过某种方法预测知道后面执行代码的可能位置，提前将这些位置所在的区域建立映射，同样还是可以有效的保证高命中率。

象例中代码执行完 if(flag==1)语句后就有两种可能，一是调用函数 func()，二是执行 flag=0，什么方法可以让我们知道这两种可能的存在呢，现在我们看代码一下就可以看出来，如果能让 CPU 做到这一点自然也就可以实现，一种叫流水线的技术实现了我们的期望。这种技术一是可以避免 CPU 需要取指、译码、执行步骤串行进行产生的时间等待；二就是一定程度上可以解决程序跳转 Cache 命中率低的问题，在进行 if(flag==1)比较之后肯定会有一条条件转移指令，在执行比较之前流水线技术就可以知道当前比较指令之后一定有跳转指令，将两种可能地址的内容都用 Cache 的精细小块建立起映射关系，这样无论程序比较结果是哪一种都能保证后续的代码已经映射到 Cache 之中。

当然要真正做到这一步并不是我所说的那么简单，需要一系列的复杂技术才可以实现，这里是为了便于大家理解而做出的最简解释，如果想完全理解透有关 Cache 的实现技术，还需要大家自己查阅相关资料，接下来为大家介绍一些 Cache 的技术要点。

Cache 的基本结构

Cache 通常由相联存储器实现，相联存储器的每一个存储块都具有额外的存储信息，称为标签(Tag)。当访问相联存储器时，将地址和每一个标签同时进行比较，从而对标签相同的存储块进行访问。如果地址没有找到与之匹配的标签，则需要将原有的标签按一定规则丢弃一个，然后将其改映射到新地址。

Cache 的三种基本结构如下：

全相联 Cache

在全相联 Cache 中，存储的块与块之间，以及存储顺序或保存的存储器地址之间没有直接的关系。程序可以访问很多的子程序、堆栈和段，而它们是位于主存储器的不同部位上。

因此，Cache 保存着很多互不相关的数据块，Cache 必须对每个块和块自身的地址加以存储。当请求数据时，Cache 控制器要把请求地址同所有地址加以比较，进行确认。

这种 Cache 结构的主要优点是，它能够在给定的时间内去存储主存器中的不同的块，命中率高；缺点是每一次请求数据同 Cache 中的地址进行比较需要相当的时间，速度较慢。

直接映像 Cache

直接映像 Cache 不同于全相联 Cache，地址仅需比较一次。

在直接映像 Cache 中，由于每个主存储器的块在 Cache 中仅存在一个位置，因而把地址的比较次数减少为一次。其做法是，为 Cache 中的每个块位置分配一个索引字段，用 Tag 字段区分存放在

Cache 位置上的不同的块。

单路直接映像把主存储器分成若干页，主存储器的每一页与 Cache 存储器的大小相同，匹配的主存储器的偏移量可以直接映像为 Cache 偏移量。Cache 的 Tag 存储器(偏移量)保存着主存储器的页地址(页号)。

以上可以看出，直接映像 Cache 优于全相联 Cache，能进行快速查找，其缺点是当主存储器的组之间做频繁调用时，Cache 控制器必须做多次转换。

组相联 Cache

组相联 Cache 是介于全相联 Cache 和直接映像 Cache 之间的一种结构。这种类型的 Cache 使用了几组直接映像的块，对于某一个给定的索引号，可以允许有几个块位置，因而可以增加命中率和系统效率。

单纯从字面可能不容易理解这三种结构到底有何不同，用一个 8kBytes Cache 和 32Mbytes RAM 的情况来解释一下这三种结构（和实际情况可能有所不同)。

全相联将 8kBytes 的 Cache 分成 16Bytes 大小 512 小段，每段映射 32Mbytes RAM 中的一个地址，CPU 存取数据时从这 512 个 Cache 小段中查询是否命中。

直接映像将 32Mbytes RAM 按 8kBytes 的大小分成 4096 页，这样每页的大小与 Cache 一致，当 CPU 需要从 RAM 存取数据只要判断存取数据的地址是否在页面映射的区域之内就知道数据是否命中。

组相联将 8kBytes 的 Cache 分成 1kBytes 大小的 8 等份，现在 32Mbytes RAM 应该分成大小为 1kBytes 的 32768 页，CPU 存取数据时从这 8 个 Cache 分区中查询是否命中。

Cache 与 RAM 的数据一致性

在 CPU 与 RAM 之间增加了 Cache 之后，便存在数据在 Cache 和 RAM 中一致性的问题。

对 Cache 的读写有两种 2 种方式：

直写法(write through)

直写法是当 CPU 在写 Cache 的同时，写入 Cache 中的新内容也会随之更新 RAM 中对应的内容，这样 RAM 和 Cache 中的内容始终是一致的，因为需要写 RAM，所以速度会慢一点，但比没有 Cache 的情况要快，如果没有 Cache 由 CPU 直接写 RAM 需要 CPU 等待 RAM 写成功，会受到 RAM 读写速度的限制，但有了 Cache 不需要这个等待时间，写 RAM 的操作会在程序执行后续代码的同时由 Cache 控制器自动完成。

回写法(write back)

回写法和直写法不同在于当 CPU 在写 Cache 的时候并不同步更新 RAM 中对应的内容，只是在特

定位置出一个刚才所写位置已经被新内容改写的标志，这个标志位叫做脏位，直到 Cache 需要抛弃当前 RAM 位置改映射新 RAM 位置时才由 Cache 将新内容更新到 RAM 中去。

当一段程序频繁使用某些临时局部变量的时候，由于这些变量是临时的，所以根本不需要写进 RAM 中去，这样回写法效率就会非常高，不用写 RAM 就可以完成整段程序功能，对于全局变量同样也会有效果，如果一个频繁被改写的全局变量对于程序来说也不需要每次都写进新内容，只需要在抛弃 Cache 映射关系时更新最后的内容就可以。

这里将局部变量和全局变量分开说是因为全局变量因为 Cache 的使用引入了一个新问题，RAM 和与之对应的 Cache 存在内容不一致的可能，当程序没有抛弃当前的 Cache 映射关系时，程序所修改的变量实际上只修改 Cache 中的内容，RAM 里面的内容保持不变，两者不相一致。这个不一致会不会对应用产生不良影响呢？答案是肯定的。

一种情况就会导致错误，当 MCU 的硬件可以不通过 CPU 与 RAM 交换数据时，错误就会产生，象我们通过 DMA 将 RAM 指定位置的内容传送到其它位置或者外围接口时，DMA 取的数据是 RAM 中的内容，而程序更改的是 Cache 里面的内容，这样 DMA 传送的数据并不是程序最新的结果。用例子解释会更清楚一些，程序将某个全局变量从 0 开始往上累加，DMA 则将这个变量传递给 UART 输出到电脑显示，这里程序累加的是 Cache 里面的值，RAM 中始终会保持 0，导致结果是电脑收到的始终是 0，直到抛弃当前的 Cache 映射关系才改为输出新的值。

这是 MCU 中的某种设备由总线绕过 CPU 操作直接读 RAM 而产生的数据不一致错误，另外一种情况刚好相反，是设备使用总线绕过 CPU 操作直接写 RAM 而导致程序处理的数据不是最新数据，将上面的情况反过来从 PC 向 MCU 发送数据，UART 收到的数据通过 DMA 直接存入 RAM，MCU 显示自己接收到的数据不能保证为最新状态。

注：后面关于问题调试与分析的章节中有与此相关的实例。

Cache 的分级

目前处理器发展趋势是 CPU 主频越做越快，系统架构越做越先进，但主存 RAM 的结构和存取时间改进相对偏慢。因此如何将 CPU 的高速特性展现出来，Cache 技术就成为不二选择，但芯片面积和成本等因素的限制不能满足 Cache 做得足够大的愿望，所以 Cache 的设计提出了分级的概念。

微处理器性能由如下几种因素估算：

性能=k(f×1/CPI－(1－H)×N)

式中 k 为比例常数，f 为工作频率，CPI 为执行每条指令需要的周期数，H 为 Cache 的命中率，N 为存储周期数。

要想提高处理器的性能，就应该提高工作频率，减少执行每条指令需要的周期数，提高 Cache 的命中率。减少 CPI 值可通过同时分发多条指令和采用乱序控制的方法实现，采用转移预测和增加 Cache 容量则可以提高 H 值，减少存储周期数 N 通常是采用高速总线接口和不分块 Cache 技术。

以前为了提高处理器的性能，主要采用提高工作频率和指令并行度这类直接方法，开始效果是非常明显，但随着改进的深入瓶颈也随之出现，也就是说靠提高工作频率和指令并行度对效率的提升效果不再明显，于是改进方向转向了提高 Cache 的命中率，正是这样的背景下设计出无阻塞 Cache 分级结构。

Cache 分级结构的主要优势在于，对于一个典型的一级缓存系统的 80％的内存申请都发生在 CPU 内部，只有 20％的内存申请是与外部内存打交道。而这 20％的外部内存申请中的 80％又与二级缓存打交道。因此，只有 4％的内存申请定向到 RAM 中。Cache 分级结构的不足在于高速缓存组数目受限，需要占用线路板空间和一些支持逻辑电路，会使成本增加所以目前采用 Cache 分级结构的 MCU 还比较少。

I-Cache 和 D-Cache

从数据集中性的分析中我们知道虽然 CPU 的指令和数据都满足集中性规则，但指令比数据会更符合这一规则，所以在集中性方面指令和数据虽然符合统一基本规则，但各自又都具有相对独立的特征。

基于这个规律有些芯片公司将 Cache 设计成 I-Cache（指令 Cache）与 D-Cache（数据 Cache）两种。这种双路高速缓存结构减少了争用高速缓存所造成的冲突，改进了处理器效能，可以让数据访问和指令调用在同一时钟周期内进行。另外对于程序通常数据和指令在内存中的位置是以数据或者指令的方式归类成块分步，采用 I-Cache 和 D-Cache 可以减少因为数据和指令的位置不同而导致的 Cache 更新操作。

```
if(flag==1)
{
  func();
}
flag=0;
```

这里读写 flag 是在数据段中进行，函数 func()和 flag=0 的指令是在代码段中，采用 I-Cache 和 D-Cache 分离方式就不会出现数据段和代码段间隔大而产生的 Cache 更新操作，对提高 Cache 效率有着明显的作用。

PC 的 Cache 技术

PC 中 Cache 的发展是以 80386 为界。

现在计算机系统中都采用高速 DRAM（动态 RAM）芯片作为主存储器。早期的 CPU 速度比较慢，CPU 与内存间的数据交换过程中，CPU 不需要进行额外的等待，以早期的 8MHz 的 286 为例，其时钟周期为 125ns，而 DRAM 的存取时间一般为 60~100ns，因此 CPU 与主存交换数据无须等待。这种情

况称为零等待状态，所以CPU与内存直接打交道是完全不影响速度的。

近年来CPU的时钟频率的发展速度远远超过DRAM，几年内CPU的时钟周期从100ns加速到几个ns，而DRAM经历了FPM、EDO、SDRAM几个发展阶段，速度只不过从几十ns提高到10ns左右，DRAM和CPU之间的速度差，使得CPU在存储器读写总线周期中必须插入等待周期；由于CPU与内存的频繁交换数据，这极大地影响了整个系统的性能，这使得存储器的存取速度已成为整个系统的瓶颈。

当然采用高速的静态RAM（SRAM）可以作为主存储器与CPU速度匹配，问题是SRAM结构复杂，不仅体积大而且价格昂贵。因此除了大力加快DRAM的存取速度之外，当前解决这个问题的最佳方案是采用Cache技术。Cache即高速缓冲存储器，它是位于CPU和DRAM主存之间的规模小的速度快的存储器，通常由SRAM组成。

Cache的工作原理是保存CPU最常用数据，当Cache中保存着CPU要读写的数据时，CPU直接访问Cache。由于Cache的速度与CPU相当，CPU就能在零等待状态下迅速地实现数据存取，只有在Cache中不含有CPU所需的数据时CPU才去访问主存。Cache在CPU的读取期间依照优化命中原则淘汰和更新数据，可以把Cache看成是主存与CPU之间的缓冲适配器，借助于Cache，可以高效地完成DRAM内存和CPU之间的速度匹配。

386以前的芯片一般都没有Cache，对后来的486以及奔腾级甚至更高级芯片，已把Cache集成到芯片内部，称为片内Cache。片内Cache的容量相对较小，可以存储CPU最常用的指令和数据。别看容量小,片内Cache灵活方便,对系统效率有相当的提高,如果在BIOS中关掉CPU的内部Cache，会让系统性能下降一半甚至更多。

但是片内Cache容量有限，在CPU内集成大量的SRAM会极大的降低CPU的成品率，增加CPU的成本。在这种情况下，采取的措施是在CPU芯片片内Cache与DRAM间再加Cache，称为片外二级Cache(Secondary Cache)。片外二级Cache实际上是CPU与主存之间的真正缓冲。由于主板DRAM的响应时间远低于CPU的速度，如果没有片外二级Cache，就不可能达到CPU的理想速度。片外二级Cache的容量通常比片内Cache大一个数量级以上。

主板上的片外Cache工作在CPU的外频下，与CPU主频速度通常相差几倍。为了进一步提高系统性能，在CPU片内Cache和主板Cache之间加入真正的二级Cache，这就是片内二级Cache。它通常以CPU主频的半速或全速工作，容量一般为128K～512K，新的至强处理器则达到2M以上。

全速的二级Cache可以极大地加速大型密集性程序的运行速度,带有同速的Cache的Pentium II至强、 Pentium Pro系列处理器是大型服务器的首选CPU。但集成高密度的二级Cache同样会加大CPU的成本，所以这一类的处理器都是价格昂贵的产品。去掉二级Cache的处理器性能虽然有不少下降，但价格可以降得很多，市场上的赛扬处理器就是一个很好的例子。

3.2. 总线

总线的专业解释是一种描述电子信号传输线路的结构形式，是一类信号线的集合，是子系统间

传输信息的公共通道。通过总线能使整个系统内各部件之间的信息进行传输、交换、共享和逻辑控制等功能。在单片机系统中，它是 CPU、内存、输入、输出设备传递信息的公用通道，MCU 的各个模块通过内部总线相连接，外部设备则通过相应的接口电路再与总线相连。

总线英文叫作“BUS”，对应中文意思为“公交车”，这是一个形象的比喻。为了更形象你可以将总线理解成一座独木桥，和常见的独木桥不同的是这个独木桥中间有许多分支，每条分支都连接一座房子，房子里面可能只住一个人，也可能住着许多人。如果一个人想从一座房子到另外一座房子里面去，就要出来经过独木桥才可以到达，但独木桥同一时刻只能走一个人，一个人出来之前就应该先看看桥上有没有人在走，没有人才可以出发。

总线分类的方式有很多，下面是几种最常见的分类方法。

按连接方式分：

按连接方式可分为内部总线和外部总线。内部总线和外部总线在功能上可以完全相同，没有本质的区别，只是在速度、抗干扰能力等指标性能方面会存在不同。内部总线在芯片内部连接各功能模块，因为总线不出芯片，所以可以工作到相对更高的速度，外部总线从通过引脚从芯片内部引出来，加上接口驱动电路后就可以连写接口功能一致的外部设备，因为外部引线会受到物理电气特性的限制，最高速度相对会慢一些。

内部总线的最高速度不是一定会高过外部总线，有一些芯片内部的工作主频比较低，这时外围的接口驱动电路最高限制速度就有可能高过总线可设置到的最高速度，此时外部总线的最高速度可以和内部总线相同。

单片机因为 CPU 只是芯片的一部分，所以除了在内部有内部总线连接各个功能模块，也会有不少单片机支持外部总线方式，这样可以让用户自行决定是否使用某些可选功能模块，外部总线加上接口驱动电路就可以和同样功能其它设备进行通讯。PC 的总线主要以外部总线方式在主板上体现，这是由 PC 的构架决定的，CPU 主要是完成运算处理功能，需要通过总线和外部交换数据。

按功能分：

最常见的是从功能上分为地址总线（address bus）、数据总线（data bus）和控制总线（control bus）。在有的系统中，数据总线和地址总线可以在地址锁存器控制下被共享，采用的是地址和数据复用方式，这种方式在一些简单的 MCU 中较为常见。

地址总线顾名思义是用来传送地址的，实际应用中最常见的是 CPU 地址总线来选用存储器的存储地址。地址总线的位数往往决定了存储器存储空间的大小，所支持的最大空间大小为 2 的总线位数次幂，象 8 位/16 位/32 位的地址总线对应的其最大可存储空间为 256/64k/4G Bytes。地址总线越宽芯片设计需要的体积越大，为了降低实现成本，一些简单的 MCU 会采用 8 位地址总线，但 8 位地址总线所支持的寻址空间只有 256Bytes，这些 MCU 会采用 PAGE/BANK 之类的技术来增大芯片的寻址空间（可参阅章节“单片机 PAGE/BANK 概念”）。

数据总线用于传送数据信息，它又有单向传输和双向传输数据总线之分，双向传输数据总线通常采用双向三态形式的总线。数据总线的位数通常与 MCU 的字长一致，8 位的 MCU 字长 8 位，其数据总线宽度也是 8 位。在实际工作中，数据总线上传送的并不一定是完全意义上的数据，象 CPU 取指令操作就是先将地址总线设为指令所在地址，然后通过数据总线将具体指令取回，此时数据总线传送的是指令而不是数据。

控制总线用于传送控制信号和时序信号。象 MCU 对外部存储器 SDRAM 进行读写操作就要先通过控制总线发出读/写信号、片选信号和读入中断响应信号等。控制总线一般是双向的，其传送方向由具体控制信号而定，其位数也要根据系统的实际控制需要而定。

另外也有分成系统总线和非系统总线的分法，不过实际应用中很少使用这种分法。

按传输方式分：

按照数据传输的方式划分，总线可以被分为串行总线和并行总线。理论上并行传输方式要优于串行传输方式，传输速率会远远高过串行方式，但其成本上会有所增加。这一点在以外部接口方式出现时更为明显，因为外部接口为了保证连接的可靠，就需要采用性能良好的接头，在接头插槽接触点位置，需要镀金等技术保证接触良好。另外并行方式接口过多的连接点会导致连接的可靠性下降，因为接头的每一个接触点出故障的几率是等同的，随着触点增多可靠性自然就会低下来。

常见的串行总线有 SPI、I2C、USB、UART、CAN、SIO 等；并行总线则有 PCI、LPT、CSI、TFT、IDE 等。

按时钟方式分：

按照时钟信号是否独立，可以分为同步总线和异步总线。同步总线的时钟信号独立于数据，也就是说要用一根单独的线来作为时钟信号线；而异步总线没有独立的时钟信号，通常是在信号中约定一个同步触发信号，这个同步触发信号被当做时间基点，然后总线上的各个模块用自己内部的时钟信号得出总线控制时序的时间轴。

因为同步总线有专门的时钟信号线，所以同步总线进行通讯时每一步都由该时钟信号线进行同步，所以同步总线的通讯时序上任意时刻各模块间的同步性是一致的。异步总线由于没有时钟信号线，各个模块的内部时钟不可能做到绝对一致，相互之间会存在误差，这个误差会在控制时序的时间轴上进行累加，累加到一定程度就有可能出错，所以异步方式每隔一段时间都需要重新同步。

总线技术指标：

评价总线的主要技术指标是总线的带宽（即传输速率）、数据位的宽度（位宽）、工作频率和传输数据的可靠性、稳定性等。

总线的带宽指的是单位时间内总线上传送的数据量，即每秒可以传送最大数据传输率。总线的位宽指的是总线能同时传送的二进制数据的位数，或数据总线的位数，即 16 位、32 位等总线宽度的概念；总线的位宽越宽，数据传输速率越大，总线的带宽就越宽。总线的工作时钟频率以 MHz 为

单位，它与传输的介质、信号的幅度大小和传输距离有关。在同样硬件条件下，我们采用差分信号传输时的频率常常会比单边信号高得多，这是因为差分信号的的幅度只有单边信号的一半而已。

MCU 有最高工作频率限制，所以对于任何 MCU 其内部总线的带宽也是有限度的，有的 MCU 内部可能包含许多功能模块，大多数时候只会用到一部分功能模块，为了降低芯片成本芯片厂商在设计总线的时候会尽量降低最高工作频率，这种情况如果将所有模块都设置到极限工作状态，就会出现总线速率跟不上的情况，从外部看是 MCU 的整体性能急剧下降。

总线信号复用方式：

依据前面对总线的定义可知总线的基本作用就是用来传输信号，为了各模块的信息能及时有效的被传送，就需要避免各模块彼此间的信号相互干扰和物理空间上过于拥挤，解决此问题最好的办法是采用多路复用技术。所谓多路复用就是指多个用户共享公用信道的一种机制，目前最常见的主要有时分复用、频分复用和码分复用等。

时分复用（TDMA）是将信道按时间加以分割成多个时间段，不同来源的信号会要求在不同的时间段内得到响应，彼此信号的传输时间在时间坐标轴上是不会重叠。

频分复用（FDMA）就是把信道的可用频带划分成若干互不交叠的频段，每路信号经过频率调制后的频谱占用其中的一个频段，以此来实现多路不同频率的信号在同一信道中传输。而当接收端接收到信号后将采用适当的带通滤波器和频率解调器等来恢复原来的信号。

码分复用（CDMA）是所被传输的信号都会有各自特定的标识码或地址码，接收端将会根据不同的标识码或地址码来区分公共信道上的传输信息，只有标识码或地址码完全一致的情况下传输信息才会被接收。

总线并不是一项陌生的技术，单片机诞生的那一天它就一直存在于单片机之中，只不过在很长的一段时间内单片机都只采用总线技术最简单的部分，这一阶段对工程师来说只要知道总线的存在就行，完全不需要了解其中的技术细节。直到 32bits 的 MCU 成为市场的一大支柱，总线技术从幕后跃居台前，实现技术也从简单变得复杂，编写底层驱动程序的时候也逐渐需要工程师去了解总线的工作方式，甚至是设定总线的控制模式。

接下看一看一个 16 位 MCU 和另外一个 32 位 MCU 的总线连接示意图，16 位 MCU 单片机总线实现可以说最为简单，而 32 位 MCU 则相对要复杂一些。

16 位 MCU 总线示例：

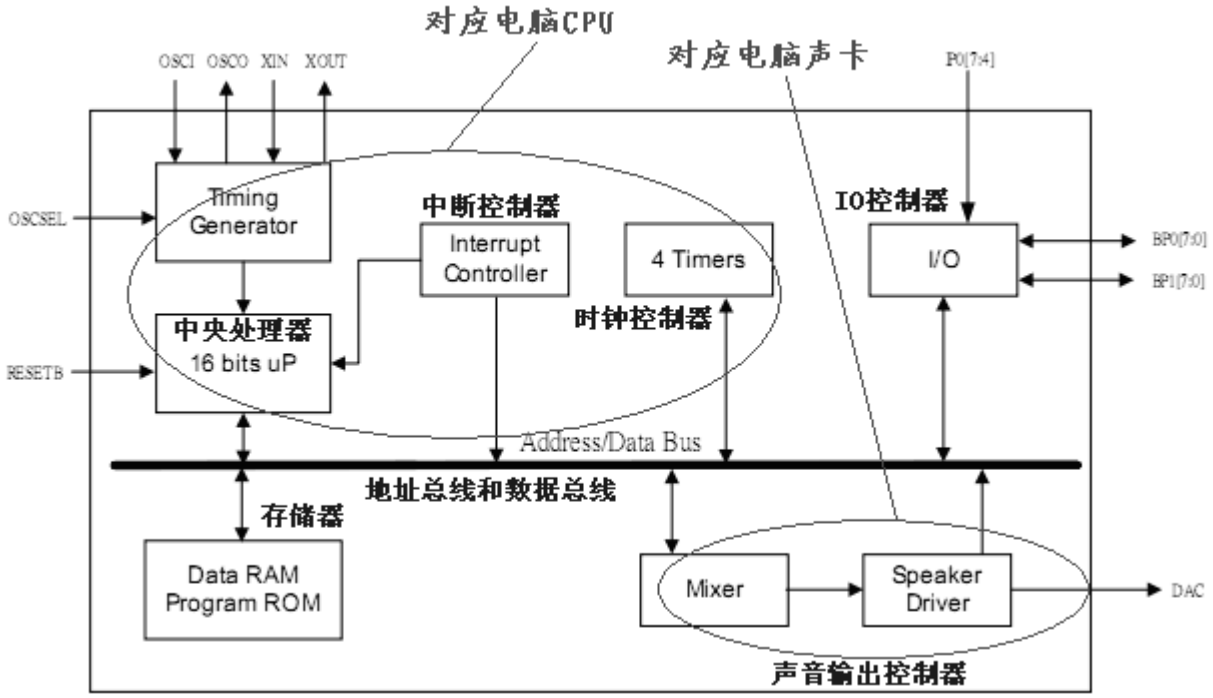

图 3.2.-1　16 位 MCU 样例总线示意图

该 16 位 MCU 所用的总线比较简单，中央处理器通（CPU）过总线与存储器（RAM 和 ROM）、中断控制器、时钟控制器、IO 控制器和声音输出控制器（SPU）相连，其中中央处理器和声音输出控制器可以独立访问存储器 RAM 和 ROM，中断控制器、时钟控制器、IO 控制器只能由中央处理器进行控制访问。

不同模块间的访问通过总线实现，存储器、中断控制器、时钟控制器、IO 控制器和声音输出控制器各自都有自己的空间地址段，不同模块间的地址段相互独立且不重复，图中的 MCU 提供 24 位地址总线，所以其支持范围为 0x000000~0xFFFFFF 的 16MBytes 寻址空间，显然该 MCU 的应用程序不会有 16Mbytes 这么大，之所以支持这么大的范围是为了给声音输出控制器提供足够的声音数据。

来看一下 MCU 的地址空间映射关系：

0x000000　0x007FFF　内部 32kBytes 的 RAM 空间

0x008000　0x008FFF　系统配制寄存器空间

0x009000　0x009FFF　中断控制寄存器空间

0x00A000　0x00AFFF　时钟控制寄存器空间

0x00B000　0x00BFFF　IO 控制寄存器空间

…

0x010000　0x1FFFFF　内部 ROM 空间（接近 2MBytes）

0x200000　0xFFFFFF　外部可扩展存储器空间（ROM 或 RAM）

当中央处理器执行程序时，先通过地址总线设定相应地址从存储器中得到指令和数据，然后执行相应指令，如果指令的操作对象是其它控制寄存器时，基本流程和存储器中读写数据过程相同，

只是需要设定不同的地址。

如果指令和数据共用同一套地址总线，取指令和操作数需要对地址总线设定相应地址，如果操作对象为 RAM 变量或寄存器则还需要对地址总线设定另外的地址以读取或存储相应数据，这样一条指令就有可能需要对地址总线进行多次设定，中央处理器每做一步操作至少需要一个系统时钟周期，这样就难以得到高的代码执行效率。针对这种问题有的 MCU 地址总线有两套，一套专门用于取指令操作，另外一套则用于数据读写，这样做理论上可以让代码执行效率提高将近一倍，但内部结构和成本会略有上升。

声音输出控制器也能独立从存储器空间读取数据，具体方法是在声音输出控制器提供一系列的寄存器供中央处理器进行设置，这些寄存器包含有声音输出开始和结束地址等信息，当程序设置好这些寄存器后声音输出控制器就开始自主工作，按规定的间隔从存储器读取数据并通过 DAC 输出，这样就不需要中央处理器再进行干预就能将指定的声音输出。

可是中央处理器会占用总线，那声音输出控制器如何通过总线得到声音数据呢？答案很简单，总线并不是时刻都被中央处理器占用，如果不是 RISC 结构的 MCU，至少在指令被执行的那一个系统时钟周期内中央处理器是不会占用总线的，这样在任意一条指令执行过程中都至少存在一个系统时钟周期中央处理器是不会占用总线，声音输出控制器就可以在这个时间间隙内使用总线完成数据读取。

那如果 MCU 是 RISC 结构怎么办？这个问题就涉及到了总线管理一些更复杂的技术，这里我们先不进行讨论，留到后面 32 位 MCU 例子中再做解释。这个例子的总线控制方法相对比较简单，能够主动申请总线操作的只有中央处理器和声音输出控制器，所以只是类似时分复用的简单方法就实现了总线的管理。

32 位 MCU 总线示例：

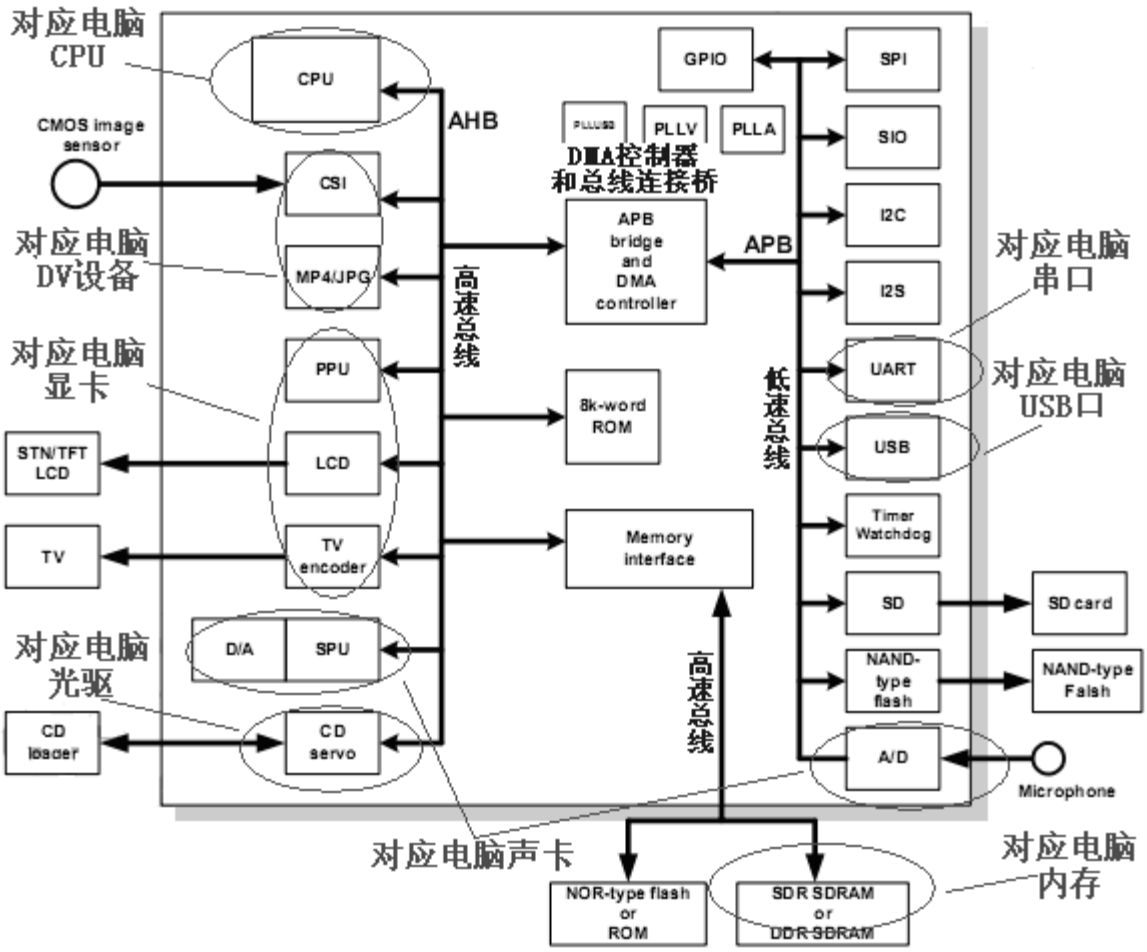

图 3.2.-2　32 位 MCU 样例总线示意图

与 16 位的 MCU 相比该 32 位 MCU 内部的总线连接明显要复杂，首先是总线有了高速和低速之分，其次多了总线控制器（总线连接桥），另外还多了和总线控制器融合在一起的 DMA 控制器。DMA 控制器我们在后面一节中会详细讲述，这里只要知道 DMA 可以独立于 CPU 之外自主完成数据传输功能。

前面 16 位 MCU 只有中央处理器和声音输出控制器会主动去申请总线传送数据，这里 32 位的 MCU 会申请总线操作的内部模块远多于 16 位 MCU 的两种。下列操作会主动申请总线，为简化说明这里将存储器全当做 RAM，实际上所有读 RAM 操作对于读 ROM 一样可行。

●CPU 总线双向读写 RAM 操作，完成 CPU 取指令、读写数据等操作。

●CSI 总线单向写 RAM 操作，将摄像头的数据自动填入相应的 RAM buffer。

●PPU 总线双向读写 RAM 操作，将各个虚拟屏的数据自动读出合成到输出 RAM buffer。

●LCD 总线单向读 RAM 操作，自动读取 PPU 输出的数据并通过 LCD 接口输出液晶屏。

●TVE 总线单向读 RAM 操作，自动读取 PPU 输出的数据并通过 TV 接口输出到电视

●SPU 总线单向读 RAM 操作，自动读取 RAM 中的声音数据并通过 Audio 接口输出。

●DMA 申请总线 RAM 与 RAM 间传送操作，自动完成 RAM 到 RAM 间的数据块传送。

●DMA 申请总线 RAM 与接口模块间传送操作，可以自动完成 SPI、SIO、I2C、I2S、UART、USB、SD、NAND 这些接口模块的输入输出操作，另外可以自动完成 CD、ADC 接口模块的输入操作。

既然可以主动申请总线操作的模块源远大于两种，继续采用类似 16 位 MCU 的总线管理方法肯定行不通，所以在这个 MCU 中出现了用于总线管理的 DMA 控制器和总线连接桥。需要留意的是 DMA 控制器并不只是单单管理前面所列出的 DMA 申请类别，对于 CSI、PPU、LCD、TVE、SPU 这几个模块它们内部隐藏有自己专用的 DMA 通道，不可以被其它模块使用，这些私有 DMA 通道同样也需要经过这个 DMA 控制器进行管理。后面两种 DMA 申请是几个公用 DMA 通道供所列出的模块共享，当某个模块需要使用 DMA 时，需要从这公用 DMA 通道中申请一个空闲通道，如果没有空闲通道则申请失败，然后等待别的模块释放通道或者停止一个正在使用的通道供其使用。

象 CSI、PPU、LCD、TVE、SPU 这几个模块需要传送的数据量都相当大，而 CPU 也是随时需要使用总线，这样相互间同时申请总线的几率自然就会高，于是总线申请冲突产生，从原理上将这种冲突是无法避免的，总线控制器就是用于总线调度管理，以消除这些冲突。通常总线控制器不允许某个模块独自长时间占用总线，会轮询产生了总线申请的各个模块，然后让这些模块分时使用总线，如果模块需要连续传送数据，总线控制器会将这一过程分割成许多小段，每段传送一小块数据，这样就可以让所有产生总线申请的模块都通过总线完成数据传送。

即便是总线控制器采用轮巡方法让各个模块分时公享总线，也还存在问题，如果是高速运行的程序在时间上偶然产生一个细小的延迟，用户可能很难察觉到，但对于 PPU、SPU 输出的图像和声音可能就不一样，也许中间数据传送的一个小小延迟会让图像和声音出现噪点和杂音。这样就要求对于这些模块同时对总线产生的申请应该由总线控制器制定出优先次序，同时产生的申请先响应优先级高的模块，将冲突导致的时间延迟尽量加到用户不容易察觉的申请类型中。

通常 MCU 是依据各模块对数据实时性依赖的程序在内部建立各模块对总线申请的优先级表，少数功能强大的 MCU 会允许程序员对优先级进行配置，对于这类 MCU 需要程序员对模块功能和总线管理非常熟悉，一般建议采用 MCU 的默认模式。

很显然，分时轮巡加优先级表的方法解决了 16 位 MCU 例中简单的总线控制方法对于 RISC 结构 MCU 不适用的矛盾，当然实际中的总线管理并不是我所说的那么简单，还需要许多复杂的技术才能实现，比如当延迟产生后如何让申请总线的模块等待延迟结束、如何让总线传输和模块的操作同步等等，这里就不一一细述。

再来看一下总线带宽的极限情况，假定该 32 位 MCU 最高工作频率 100MHz，总线位宽为 32 位，其 PPU 支持 4 层虚拟屏，如果我们将这 4 层虚拟屏全部打开、屏幕为 VGA（640*480）、16bits 颜色、同时输出到 LCD 和 TV、打开摄像头（VGA 模式），看看每秒输出 30 帧的时候显示功能会占用总线多少资源。

每层虚拟屏需要 640*480*2=614400Bytes。

4 层虚拟屏、LCD/TV 输出和摄像头输入共需要 614400*7=4300800Bytes。

每秒 30 帧则需要 4300800*30=129024000Bytes（约为 129MBytes）。

MCU 最高工作频率 100MHz，总线位宽为 32 位，带宽=最高工作频率*总线位宽/8=400MBytes。但是实际应用中虽然 MCU 内核为 32 位，为降低成本往往外部只接一片 SDRAM，此时总线访问外部 SDRAM 实际上是 16 位模式，所以带宽还需要除以 2 为 400MBytes/2=200MBytes。另外总线控制器在不同模块间进行管理切换需要时间，需要通过总线向 SDRAM 发送一些控制指令等，所以实际上有效带宽大概只有 200MBytes*70%=140MBytes。

这个数值和 129MBytes 已经相差不大，也就是说此时显示功能几乎耗尽了 MCU 的总线带宽，已经难以为程序提供正常运行所需带宽。实际测试结果也验证了这一分析结果：当 MCU 工作在前面状态下摄像头的图像在屏幕上回显会出现许多小彗星一样的飞点，如果减少一个虚拟屏或者将 LCD/TV 一个关闭输出，飞点消失。飞点消失后将主频减半，飞点重新出现，而且更加严重。

总线的通信协议

总线除了从电气层面定义接口方式外，通信协议也是非常重要的，任何一种总线都会有通信协议与之对应，相关文档白皮书则会从最基本的模型开始介绍总线的实现方式。就单片机应用来说，了解总线协议并不需要过多探究七层协议之类的理论，重要的是了解总线传送数据的时序，知道总线是如何传递数据就完全满足应用需求。

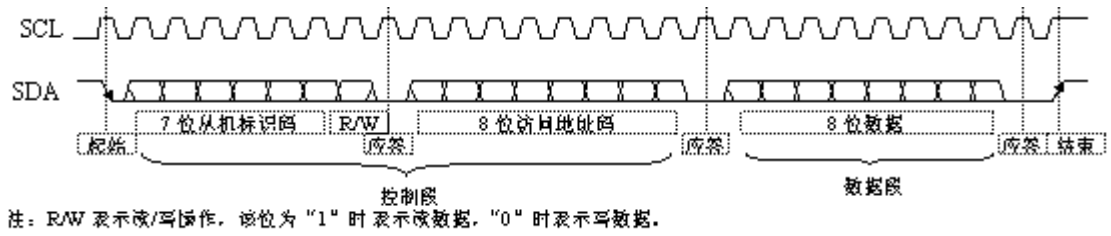

图 3.2.-3　I2C 总线时序图

图示是常见的 I2C 总线上传输的一字节数据的数据帧，其总线形式是由数据线 SDA 和时钟 SCL 构成的双线制串行总线，并联在总线上的各个设备既然可作为发送器（主机）也可作为接收器（从机）。帧数据中除了控制码（包括从机标识码和访问地址码）与数据码外还包括起始信号、结束信号和应答信号。

起始信号：SCL 为高电平时 SDA 由高电平向低电平跳变。

控制码：也叫地址码，用来选择操作目标，只有内部地址码与之相同的从设备才会继续响应后面数据。

数据码：是主机向从机发送的具体的有用的数据。

应答信号：接收方收到 8bits 数据后，向发送方发出低电平做为确认信号。

结束信号：SCL 为高电平时，SDA 由低电平向高电平跳变表示数据帧传输结束。

按协议规定在 SDA 和 SCL 上应该接 4.7k 上拉电阻，这一要求和应答信号和结束信号的定义是一致的。应答信号为低是接收方收到数据后主动将 SDA 拉低，如果没有响应上拉电阻会将 SDA 拉成高。结束信号表示总线操作结束，于是总线会被释放掉，所以定为从底变为高不会与总线释放变为高产生冲突。

3.3. DMA

DMA 是 Direct Memory Access（存储器直接访问）的缩写，它是一种高速的数据传输操作，允许在外部设备和存储器之间直接读写数据，既不通过 CPU，也不需要 CPU 干预。数据传输过程的操作管理通过 DMA 控制器实现，当其进行数据传输时，只需要 CPU 在数据传输开始和结束时对 DMA 控制器进行相关设定，然后 DMA 控制器自行完成指定的数据传输工作，在传输过程中 CPU 可以解放出来进行其它工作。

这样设计实质就是将某些数据搬运工作由硬件完成，不需要消耗 CPU 的软件资源，能这样设计是因为实际应用程序 CPU 大部分时间并不占用总线，在这部分时间段内 MCU 实际上是可以通过总线来间隔传输数据，所以可以将 DMA 控制器与 CPU 对总线的操作设计为并行状态，两者相互交替使用总线，这样就可以将 CPU 没有占用总线的那段空余时间利用起来，使整个系统的效率大为提高。

就这样单纯的说效率可大大提高并没有说服力，相信不少人还是会疑惑效率到底提高在什么地方呢？现在手机大都带有照相功能，也可以摄录一些视频短片，只要手机工作到照相机模式，就会将摄像头的实时画面显示在屏幕上，加入现在你是开发手机的工程师，对于这项功能你会怎样实现？

如果没有 DMA 功能，只能是编写程序从摄像头（CMOS Sensor）将实时画面的图像数据取回来，然后将这些数据通过 LCD 显示出来，图像数据从 CMOS Sensor 搬运到 LCD 的工作需要由程序来完成，假定我们每次搬运一个点的颜色数据，就算是完成 QVGA/30 帧这样的效果也需要一秒传输 320*240*30=2304000 个点。

那完成一个点的数据搬运需要 CPU 做多少事情呢？最少需要下面步骤：

a.依据当前点位置判断是否向 CMOS Sensor 给出行场同步脉冲信号

b.向 CMOS Sensor 给出时钟信号

c.读当前点的颜色数据

d.依据当前点位置判断向 CMOS Sensor 给出行场同步脉冲信号

e.向 CMOS Sensor 给出时钟信号

f.写当前点颜色数据到 LCD

g.更新下一点的位置继续循环

就算每一步平均需要两条指令一个点会耗费 14 条指令，完成实时图像数据的搬运每秒需要执行 2304000*14=32256000 条指令，实际情况比这个数会更多，无疑占用了太多的 CPU 资源。

有了 DMA 情况会截然不同，这每秒 32256000 条用来搬运数据的指令可以全部省掉，这类手机为了支持 CMOS Sensor 和 LCD，芯片会提供相应专用接口，接口可以自动完成同步、时钟信号的处理，同时将输入数据写进指定位置或者从指定位置读出并输出。可能在 MCU 的内部结构图或数据手册上并没有明确这些数据的传送是 DMA 完成，从实质上讲这类操作都可以归纳为 DMA 操作，只是表现形式不同。

现在只需要程序通过 CPU 设定好 CMOS Sensor 和 LCD 的工作参数，好让摄像头和屏幕工作起来，这些参数一定包含有设定数据空间的操作，象 CMOS Sensor 和 LCD 需要设定数据缓冲区的起始地址、图像的宽和高以及图像的颜色深度等信息。有了这些设定，当 CMOS Sensor 开始工作就会由硬件自动将数据填入所设定的数据缓冲区地址，LCD 对应数据缓冲区的数据则会由硬件自动读出并输出给液晶屏，只要两者参数相互适应且数据缓冲区地址相同，CMOS Sensor 的实时画面就可以不受 CPU 干预自动在屏幕显示出来。

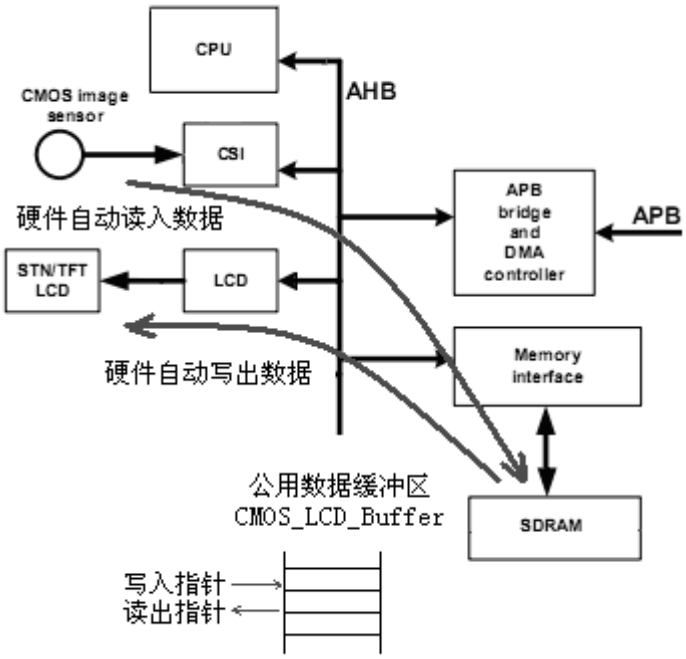

图 3.3.-1　DMA 操作示意图

在讲 Cache 时曾提 DMA 传输数据不经由 CPU，从图示可以看出从 CMOS Sensor 进来的数据直接由 DMA 通过 AHB 高速总线传进 SDRAM，确实不需要经过 CPU 进行传递，LCD 显示也是同样情况，DMA 控制器通过 AHB 高速总线将 SDRAM 中的数据直接传递给 LCD 显示。

那 DMA 是不是可以完成任意方式的数据搬运操作呢？答案是否定的，DMA 控制器不可能设计得非常复杂，基本上都是设定好起始地址和所需搬运数据的长度和方式就可以自动开始进行传输，每完成一次传输硬件会自动将地址递增或递减，这样 DMA 的传输过程实际上就只适用地址连续的数据块传输，间隔传输则无法实现，更不用说想做到随机地址传输。

对于例子中的 CMOS Sensor 和 LCD 的 DMA，只能由这两个模块专用，因此在设定上相当简单，只需要设定传输地址和传输数据大小，不用管传输数据宽度等其它设定。但通用的 DMA 不能设计得过于死板，会允许用户自己设定传输数据位宽、传输模式等，这样同一个通用 DMA 通道可以依据实际情况进行配置，以期得到最佳效果。

比如现在想让 RAM 之间的 DMA 数据传输速度最快，就可以将 DMA 的传输数据位宽和传输数据块长度设最大。如果设置成传输数据位宽 32bits、传输模式为每次传 32 次块传输，那总线每让 DMA 占用一次就可以传送 4*32=128 字节；要是设置成传输数据位宽 8bits、传输模式为每次传 1Bytes 单点传输，总线每让 DMA 占用一次则只能传送 1*1=1 字节。显然前一种设置速度会更快，既然前一种速度要快，而 DMA 的设计目的也是更快传输数据，为什么还可以出现后一种设置呢？

前一种设置速度虽然有优势，但每一次传输实际上传送了 128 字节，这样 DMA 传输的长度必须是 128 的整数倍，否则设定会出错，而后一种可以设置任意传输长度。另外 DMA 的块传输意思是 DMA 占用总线后要完成规定次数传输才释放总线，前一种设置每次 DMA 占用总线时间都是后一种的 32 倍，这样如果有其它模块需要使用总线传输数据前一种设置就需要等更长的时间才有机会申请到总线，要是这个模块对数据的实时性要求很高就会有不良影响。

通常 DMA 会包括这些组成功能：设定传输数据地址、设定传输数据数量、设定 DMA 的控制 / 状态逻辑、DMA 总线请求触发、DMA 数据缓冲、管理 DMA 中断。对 DMA 的设计并没有固定模式，只是要求通过简单设定就能完成地址连续的数据块传输，所以不同的 MCU 设计 DMA 往往会依据自己芯片的应用方向而各具特色。

前面例子中图像输入输出的数据格式没有统一标准，象 LCD 和 CMOS Sensor 有的支持 YUV 格式，有的支持 RGB 格式。YUV 格式显示效果要好， RGB 格式但对于程序员直观，可以知道图像中任意点的颜色，如果程序员希望处理这些数据，YUV 格式需要编写程序转换成 RGB 格式才容易理解。对于 LCD 和 CMOS Sensor，厂家为了降低成本，可能只支持 YUV 和 RGB 格式中的一种，虽然 MCU 大都同时支持 YUV 和 RGB 格式，但如果产品选用的 CMOS Sensor 只支持 YUV 格式而 LCD 只支持 RGB 格式同样存在问题，也需要编写程序转换数据格式才能正确显示。可是软件转换需要逐点转换，转换公式是一个 3*3 的矩阵计算，需要相当多的 CPU 指令才可以完成，所以软件转换的效果不怎么好。

台湾一些芯片设计公司发现了视频图像数据传输处理方面这一特殊需求，于是他们在设计一些芯片 DMA 功能的时候特意加上了相应处理，象针对电视游戏机市场的一些 MCU 他们让 DMA 完成数据传输的同时支持格式转换，因为 YUV 和 RGB 格式转换的公式是恒定的，用程序转换耗费 CPU 资源多是需要将所有的点都按转换公式计算一遍，如果在芯片内部设计有专门的计算电路就可以省去软件

计算矩阵所耗费的时间，这样这些 MCU 在进行 DMA 数据传输的时候可以由程序员选择是否同时由硬件进行数据转换，解决了软件转换效率低下的问题。

武侠影视剧中常看到大侠在空中飞来飞去，对于这种场景喜欢问为什么的人会疑惑，这不是违背了牛顿三大定律么？是否违背牛三定律我不管，影视剧嘛能带给观众良好的视听享受就行，没必要和科学较真，但我们可以了解这些场景到底是怎么拍出来的。飞来飞去大家都清楚，用细钢丝绳将演员吊起来拉来拉去，钢丝绳细，距离远一点摄像机就不会拍出来。但一些在悬崖之类的危险地方飞来飞去难道也是吊几条钢丝绳在悬崖上？肯定不是，要是那样做多危险，大都是让演员先吊在摄影棚里拍，然后拍后面的背景，再把两个场景合成在一起。

如何把两个场景合成在一起好象是一件挺奇妙的事情，娱乐新闻里面常会看到这样的场景，演员吊在一面蓝色（或其它颜色）的背景墙之前做各种动作，但最后出来的影视作品演员就变成了在悬崖、沙漠中，着实让人有点不可思议。其实刚才我所说的那些台湾芯片公司在设计 DMA 附加功能时也能实现场景合成这么奇妙的工作，我们知道，颜色是由三基色构成，任何颜色都可以由 RGB 三原色组合而成，这些芯片可以由程序员定义一种颜色，在该颜色三基色附近的颜色当成透明色，当 DMA 传输的时候就会把这些颜色数据过滤掉。比如 24bits 色为了滤掉蓝色背景，我们就可以定义红色和绿色分量小于 10、蓝色分量大于 245 的颜色为透明色，只要背景的蓝够蓝，用该 DMA 传输出来的数据就能将背景墙过滤掉。

要想用好 DMA，就要熟悉总线的运作方式，DMA 和总线两项技术是相互相成，总线提供可以进行数据快速传递的前提条件，DMA 则是总线实现数据快速传递的具体方式。

3.4. 存储器管理

单片机上电后 PC 指针会自动指向一个固定地址，通常这个地址为 0，然后从该地址开始执行具体程序。简单的单片机的地址分配都很简单，MCU 自身内部所带的 ROM/RAM 会占用固定的地址空间，如果外部可以扩展也一般只支持单片存储器扩展，所扩展的地址空间也都是从特定位置开始，大小不超过外部扩展存储器容量的空间。

当然也可以通过一些特殊方法实现外部多片存储器的扩展，比如用不同 IO 口去选择外部不同存储器的 CS 脚，以实现对扩展接口的地址和数据总线的共享。但这么做需要软件做出相应处理，这些存储器就 MCU 来说是相同的地址空间，类似于一些小单片机为了增大存储空间所采用的 PAGE、BANK 方法。

简单的单片机程序都是在 ROM 中直接运行，程序所需的 ROM 和 RAM 空间一般不会太大，所以这些简单的单片机大都提供出容量一定的存储空间给用户使用，如果空间不够就只能选择另外的单片机进行开发。随着 32 位单片机和嵌入式系统的兴起，不同产品程序所需的 ROM 和 RAM 空间可能相去甚远，小的程序只需要几十 k 的空间就够用，而大的可能需要几十 M 甚至上百 M 才够用，这样 32 位的单片机为了有更好的适用性，改成了自身所带存储器空间非常小，主要由外部存储器扩展的方

式。这样做的好处是有利于降低成本，可以由用户自行决定所需存储器的大小，避免出现成本过高或者空间浪费的情况。

为了支持大容量的存储空间，这些单片机往往可以支持多片外部存储器扩展，和简单单片机不同之处在于这些单片机外部有多条用于存储器扩展的 CS 脚，每一条 CS 脚都映射有一段空间，这样单片机硬件自己就可以通过这些 CS 脚的选择自动访问外部不同的存储器，不需要简单单片机中用程序改变外部存储器的选择。

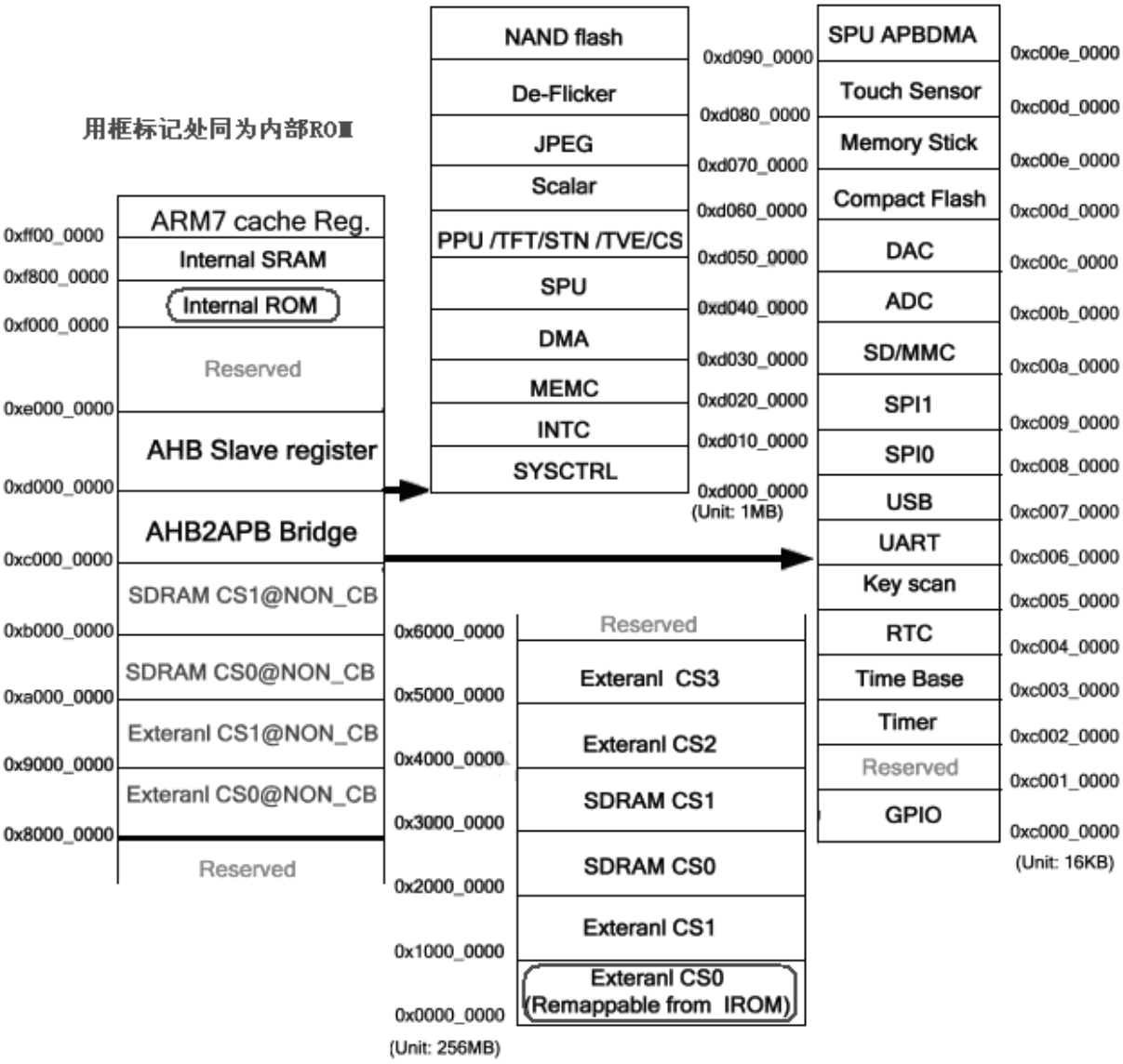

图 3.4.-1 MCU 存储器空间地址表

图示中的 MCU 最多可以扩展 6 片外部存储器，从图看该 MCU 支持的地址总线为 32 位，实际上其地址总线的最高位用来做其它用途，实际上有效地址总线为 31 位，也就是说只有 bit31 不同的两个地址实际上是同一个地址，所以在图中可以看到每条 CS 对应高低两段空间，象 SDRAM CS0 就

对应 0x20000000 和 0xA0000000 两段空间，这两段空间实际上是同一片外部 SDRAM。当程序访问 0x20000000 时，硬件就会自动访问由 SDRAM CS0 选择的外部 SDRAM，不需要程序再做其它任何控制。

对于外部存储器的扩展并没有连接次序或数目的限制，SDRAM 可以从 SDRAM CS0 和 SDRAM CS1 中自由选择组合，只扩展一片可以选 SDRAM CS0，同样也可以选 SDRAM CS1，只需工程师自己知道所对应的地址空间。至于外部扩展存储器的空间大小和总片数也是只要不超过图中的限制就行，象 SDRAM CS0 扩展一片 16MBytes 的 SDRAM，而 SDRAM CS1 却扩展一片 32MBytes 的 SDRAM 都是可以的，只是从 0x20000000 起 16MBytes 空间才有效，从 0x30000000 起是 32MBytes 有效。

既然这类 MCU 可支持的存储器空间急剧增加，空间大小分布又不是必须满足等大小和连续的规则，如果还要求程序员了解系统的存储器空间分配和具体硬件实现后再去写程序显然不太方便，所以最好 MCU 自身能提供出一整套完备的存储器管理的方法，设定好以后由硬件自动完成相关管理，程序员只需知道他可以使用的存储器空间大小就可以编写应用程序。

当然 MCU 不提供硬件对存储器空间管理功能也还是可以编写程序的，来看看这样对程序员编程会造成多大的麻烦。我以图示中 MCU 设计一种极端情况，六个 CS 分别扩展 1/2/4/8/16/32MBytes 的存储器，这种情况得到的实际有效存储器空间如下：

按 1/2/4/8/16/32MBytes 顺序扩展	按 32/16/8/4/2/1MBytes 顺序扩展
0x50000000~0x51FFFFFF	0x50000000~0x500FFFFF
0x40000000~0x40FFFFFF	0x40000000~0x401FFFFF
0x30000000~0x307FFFFF	0x30000000~0x303FFFFF
0x20000000~0x203FFFFF	0x20000000~0x207FFFFF
0x10000000~0x101FFFFF	0x10000000~0x10FFFFFF
0x00000000~0x000FFFFF	0x00000000~0x01FFFFFF

图 3.4.-2　外部存储器扩展表

对于这种有效地址的分布情况，程序员需要时刻防止自己的程序进入到无效空间，一旦外部扩展方式改变，需要重新编写整个程序。MCU 提供的硬件存储器空间管理功能可以让这些烦恼一扫而空，即便是外部扩展方式改变，也只需在系统底层做少量修改，完全可以做到不更改应用程序。

MMU（Memory Management Unit）

MMP 字面意思为存储器管理单元，其主要功能就是解决前面所提出的应用程序和实际地址空间如何协调的问题。先提出两个名词：物理地址和虚拟地址（也可将虚拟地址称为逻辑地址）。物理地址就是在硬件层面看存储器所处的空间位置，物理地址等于访问存储器的地址总线上的地址内容，前面表中的地址均为物理地址。虚拟地址（逻辑地址）则是站在应用程序层面看的存储器的空间位置，它可以等同物理地址，也可以与之不同。

MPU 的做法是将其所能支持的物理地址空间分成许多小块，然后以这些小块为单位将其实际物理地址映射成另外一个地址。比如物理起始地址为 0x00000000 的 64kBytes 空间，MPU 就可以将其映射到起始地址为 0x10000000 或 0x20080000 的位置。CPU 程序需要访问存储器时候，先不将需要访问的地址写入地址总线，而是交给 MPU 的一个转换器，这个转换器先进行虚拟地址到物理地址的逆转换，然后依照逆转换出来的实际物理地址访问存储器。比如 MPU 将物理地址为 0x00000000 的 64kBytes 空间映射成逻辑地址 0x20080000，当 CPU 访问地址 0x20080000 开始的 16kBytes 空间时，实际访问的物理地址并不是 0x2008XXXX，而是 0x0000XXXX。

需要留意的这个地址转换是以某个大小为单位的块为基本单位，MPU 不可能做到将所有的地址单个一一对应，芯片设计人员希望 MPU 的内部转换器越小越好，这样可以节省芯片空间。所以通常块大小都大于 4kBytes，而且允许用户可以自行选择设定块的大小，最终可以将所有的物理地址空间都映射到一个连续的虚拟地址空间。

按 1/2/4/8/16/32MBytes 顺序扩展	虚拟地址（逻辑地址）
0x50000000~0x51FFFFFF	0x01F00000~0x01FFFFFF
0x40000000~0x40FFFFFF	0x00F00000~0x00FFFFFF
0x30000000~0x307FFFFF	0x00700000~0x00EFFFFF
0x20000000~0x203FFFFF	0x00300000~0x006FFFFF
0x10000000~0x101FFFFF	0x00100000~0x002FFFFF
0x00000000~0x000FFFFF	0x00000000~0x000FFFFF

图 3.4.-3　外部存储器虚拟地址映射表

在系统启动的时候先运行固化在系统板上的启动配置程序，完成对转换表的相关设定，启动起来后再装载应用程序就可以按虚拟地址访问存储器，如果系统存储器硬件有改变，也只需更改这部分启动配置代码，完全可以做到应用程序不做任何修改。

Remap

嵌入式系统和通用单片机的程序运行方式不一样,通用单片机的程序大都是在 ROM 中直接执行，虽然有少数应用会将程序放在 RAM 中执行，但都是特殊实现方式，通用单片机的设计思想就是让程序在 ROM 中直接执行。而嵌入式不同，几乎所有的应用程序都是放在 RAM 中执行的，这就存在一个问题，单片机断电后 RAM 里面的内容不会被保存，所以嵌入式系统需要解决断电后可重新运行程序这一问题。

要想保存 RAM 中的内容不丢失，只有在断电后继续保持 RAM 供电，方法无非就是用备用电池，这样做对于第一次上电和取走电池的情况无效，所以用电池的方法行不通。于是 MCU 采用了另外一种方法，在芯片内部自带一小片内部 FALSH，另外还有一小片内部 SRAM，上电时这片内部 ROM 的地址会映射到 0x00000000 位置，看前面的图用方框标示的 IROM 就是它，在地址映射图中有对应到地

址 0x00000000 位置,它自己的真实物理地址是在 0xF0000000。芯片上电后会从固定地址 0x00000000 执行程序，这样就会执行 IROM 中的程序。

不过内部 ROM 中的程序编写存在一些特殊限制，通常这段代码（常被称为 bootloader）是由汇编代码写成，为什么要用汇编是因为如果用 C 无法保证代码中不使用 RAM 变量，而在刚上电的时候物理地址和虚拟地址之间的映射关系还没有建立，只能是通过绝对物理地址来访问存储器。而且外部扩展的存储器不是接上去就可以使用的，还需要对接口进行一些设定才能可靠访问。用汇编则可以保证不使用 RAM 变量，如果非用 RAM 变量不可也可以直接在内部的 SRAM 中按绝对物理地址直接定义。

这段汇编代码需要完成系统时钟、中断等资源的初始化，另外还需要配置好外部扩展存储器的接口参数，到这一步系统可以使用外部扩展的存储器，不过还需要按绝对物理地址进行访问。接下来的工作是将真正的程序装载进来，程序的位置和大小等信息事先需要约定好，汇编代码按照约定将真正的程序装载到 SDRAM 的指定物理地址。为了加快程序的装载速度这里会要求打开 Cache，如果不开速度可能相差数倍。

装载完程序启动代码的工作基本完成，接下来应该开始执行所装载的程序，只需要把程序指针指向所装载程序的入口地址即可。但现在所装载的程序物理地址可能并不等于程序层面所需的虚拟地址，所以跳转前还需要做一个重要操作，就是 Remap，这步操作其实很简单，将某个寄存器的一个控制位置上就可以，当然之前已经设定好 MMU 的映射表，此后 MMU 就开始工作。这里对程序还有一个特殊要求，因为在 Remap 之前程序是以物理地址基准，Remap 之后变成虚拟地址，实际上程序已经转到另外的物理空间去，所以需要保证新的逻辑地址中后一条指令依然相同。这个特殊要求可以这样小技巧实现，启动程序和所装载的程序都按一定格式进行编写，比如按照后面的固定格式定义从地址 0x00000000 开始的代码内容与结构，这样启动程序和所装载的程序开始一段代码的格式是一样的，每个位置对应的都是相同信息，所以 Remap 之后就能得到正确的跳转指令。

```
.org 0x0
ROMbase:
  b  Reset_Handler
  ldr  pc,=mmUndefinedInstructionEntry
  ldr  pc,=mmSWIEntry
  ldr  pc,=mmInstructionAbortEntry
  ldr  pc,=mmDataAbortEntry
  nop
  ldr  pc,=mmInterruptEntry
  ldr  pc,=mmFastInterruptEntry
.org 0x40
__mmUndefinedInstructionEntry:
```

```
  .long  mmUndefinedInstructionEntry
__mmSWIEntry:
  .long  mmSWIEntry
__mmInstructionAbortEntry:
  .long  mmInstructionAbortEntry
__mmDataAbortEntry:
  .long  mmDataAbortEntry
__mmInterruptEntry:
  .long  mmInterruptEntry
__mmFastInterruptEntry:
  .long mmFastInterruptEntry
.globl  Reset_Handler
.type  Reset_Handler, function
Reset_Handler:
  bl  _kmc_asm_init     /*初始化系统*/
  bl  _init_cache_mmu   /*开Cache*/
  bl  _copy_text_data   /*装载程序*/
  bl  _fill_bss         /*初始化变量区*/
  ldr  pc,=start       /*跳转到程序入口*/
```

不是所有MCU 都有内部FLASH，有的MCU 会省掉内部 FLASH，直接用外部扩展的FLASH 来启动，这种设计存在某些FLASH 不可以使用的可能，因为芯片上电后扩展接口工作在默认状态，如果选用的FLASH 不支持该工作设定就可能不能使用，不过基本上所有FLASH 的工作接口方式都相同，所以一般不存在问题。前面图示例子也可以不用内部FLASH 启动，该MCU 有一条管脚选择内部ROM 还是外部ROM，所以图中可以看到IROM 在0x00000000 位置和一组扩展CS 位置重叠，实际应用中为二选一，并不矛盾。

MPU（Memory Protection Unit）

MPU 字面意思为存储器保护单元，嵌入式系统的程序需要一个基础框架来支撑，就好比电脑的Windows 程序需要在Windows 支撑一样，这个基础框架就是操作系统，应用程序是不允许对系统所在的位置进行读写操作的，否则可能导致系统崩溃，这就需要对系统所在空间进行保护，MPU 就是提供这类功能。

保护可以通过软件来实现，但这样做需要对软件的编写提出额外的保护要求，而且所写的软件如果出错保护就无法实现，所以软件保护不是一个方便可靠的方法。MPU 所提供的是硬件保护，系

统由专门的营建来检测和限制系统资源的访问，当任何程序去访问任意地址之前，MPU 会依照所制定的访问权限控制规则检查当前程序是否有权限进行访问。

如果没有相应权限而程序试图进行访问操作，MPU 会产生一个异常信号，该信号会触发处理器执行异常中断处理程序，这就是嵌入式系统常常遇到发生异常中断的一个主要原因，程序试图访问一个它没有权限的地址。当然 MPU 的保护对于此类错误并不能避免或者从错误中恢复，只能告诉调试的程序员当前发生了超越权限的访问，要完全解决错误还需要程序员自己查找出超越权限代码的位置。

来看一个带 MPU 的 MCU 的存储器映射例子，该芯片内部自带 256kBytes 的 RAM，地址范围为 0x00000~0x3FFFF，外部可扩展的的存储器为 0x10000000~0x12000000 的 32MBytes 空间。

该例子具备这些特点：

●系统小于 64kBytes，向量表、异常处理程序位于此空间内，系统软件空间用户模式下的程序不可访问，以免应用程序破坏系统。

●有一个不超过 64kBytes 的共享系统空间，用于存放系统提供的通用库以及用户任务间的数据传递。

●最多支持 3 个独立功能的用户任务，每个任务占用的空间最多 32kBytes，任务间完全独立，不可以相互访问。

功能	访问级别	起始地址	大小	区域
外部扩展空间	系统	0x10000000	32MBytes	4
受保护的系统	系统	0x00000000	4GByte	1
共享的系统	用户	0x00010000	64kBytes	2
用户任务 1	用户	0x00020000	32kBytes	3
用户任务 2	用户	0x00028000	32kBytes	3
用户任务 3	用户	0x00030000	32kBytes	3

图 3.4.-4 MCU 存储器空间权限分配表

区域表示空间处于访问控制权限表中的位置，不同 MPU 映射关系会不同，例子中的 MPU 总共支持 16 级权限控制区域供程序员设定。

管理员	用户	区域编号
不可访问	不可访问	0
读/写	不可访问	1

读/写	只读	2
读/写	读/写	3
不可预知	不可预知	4
只读	不可访问	5
只读	只读	6
不可预知	不可预知	7
不可预知	不可预知	8~15

图 3.4.-5 MCU 存储器访问权限表

系统软件空间只允许管理员进行读写，用户编写的应用程序是不可进行任何读写操作；系统共享空间因为要给应用程序提供库函数，所以应用程序具有读权限，但是不可以写，如果任务间需要传递数据必须调用相应系统函数才能完成；三个用户任务各自拥有一段空间，这段空间对于任务自己和管理员是完全敞开的，可以进行读写操作。外部扩展空间定义成不可预知是现在不知道外部所接存储器的类型，如果是 SDRAM，读写操作都是允许的，如果是 NOR FLASH 显然不可以进行写操作，所以只能定义成不可预知。

如何设定存储器的保护不同 MCU 在数据手册的 MPU 部分会有详细描述，主要是通过设定一系列特殊寄存器来实现。通过这些寄存器可以依照特定的地址、空间大小转换规则建立一张映射表，然后 MPU 依照映射表进行保护。如果用户不需要 MPU 功能，可以选择关闭 MPU 功能。

3.5. 嵌入式与操作系统

单片机发展到今天，嵌入式和操作系统已经成为单片机产品开发的一大主流趋势，掌握嵌入式系统的相关知识已经成为一名电子工程师的必备素质，这一节我们一起来谈谈嵌入式操作系统，嵌入式样操作系统可以说是目前单片机技术在软件方面所发展到的最高层次，不是三言两语就能说清楚的，所以这一节中我只是简单的讲一点我个人对嵌入式系统的理解，如果你想深入了解嵌入式核心，那还得要你自己去多找一些关于嵌入式的资料进行钻研。

什么是嵌入式

什么是嵌入式？看似简单的一个个问题，实际却没几个人能回答得清晰透彻，即便是一些在嵌入式领域有着丰富经验的人也只是知道好像就是那么一种单片机方面的开发应用，到底具体指什么也不能很肯定。我自己实际上也认为嵌入式是一个含糊的概念，它渗透在电子产品开发之中，与传

统常规单片机开发并没有严格明晰的界限。

IEEE（国际电气和电子工程师协会）对嵌入式系统的定义是：Devices Used to Control，Monitor or Assist the Operation of Equipment，Machinery or Plants，意思为用于控制、监视或者辅助操作机器和设备的装置。这个定义相对比较抽象，从字面意思理解所有电子控制设备都属于嵌入式，哪怕就是一个振荡器控制LED闪烁的电路都是，这种理解显然过于广泛。

目前国内普遍认同的嵌入式系统定义为：以应用为中心，以计算机技术为基础，软硬件可裁剪，适应应用系统对功能、可靠性、成本、体积、功耗等严格要求的专用计算机系统。

嵌入式系统（Embedded System），一般由嵌入式微处理器、外围硬件设备、嵌入式操作系统以及用户的应用程序等四个部分组成，用于实现对其他设备的控制、监视或管理等功能。和传统单片机电子产品相比，嵌入式样更能体现功能细分的时代潮流特性。

传统单片机产品是工程师在特定硬件平台下需要完成所有的代码工作，即便是芯片厂商提供有代码样例，也只是适用于与之相适应的硬件，如果更换硬件则需要重新编写代码。不同的厂商芯片代码编写没有统一规程，每家都是按自己的意愿设计样例代码，各种接口的驱动代码也是同样境况。如果想把一个基于东家MCU的产品程序移植到西家MCU上，基本上可以移植的只有程序流程和框架，原来的代码只有参考意义。

传统单片机厂商所为了便于工程师迅速开始编写程序所提供的无功能样例，样例包含有中断、主循环等关键组成单元，工程师只要在上面填写自己的代码就可以正式开始程序编写，可以省掉学习如何搭建程序架构的过程。

传统单片机程序构架样例：

```
#include <HT46R01C.H>
#pragma vector isr_4 @ 0x4
#pragma vector isr_8 @ 0x8
#pragma vector isr_c @ 0xc
//中断服务函数定义，如果需要中断在对应函数内添加相应代码
void isr_4(){} //external ISR
void isr_8(){} //timer/event0
void isr_c(){} //ADC convert
//----------------------------------------
//安全初始化，可以去掉此函数自己初始化系统
//----------------------------------------
void safeguard_init()
{
  _wdts = 0x00;
  _intc0 = 0x00;
```

```
  _tmr0c = 0x00;
  _tmr1c = 0x00;
  _ctrl0 = 0x00;
  _adcr = 0x00;
}
//------------------------------------------
//主程序
//------------------------------------------
void main()
{
  safeguard_init();
  while(1)
  {
    //添加用户代码
  }
}
```

有了这样的例子工程师就可以在此基础上编写自己的代码，即便改变芯片型号只要是同公司同一系列的芯片，也都很容易用照葫芦画瓢的方式建立自己的新项目工程。

嵌入式系统实际上就是起类似样例框架代码作用，是一些专业公司针对某些硬件平台设计出基本程序框架，框架本身不包含实际的具体的功能实现，但要求尽可能的支持硬件平台的所有功能，也就是说其它工程师在此框架基础之上通过接口驱动函数可以使用硬件平台的所有功能。既然嵌入式系统由专业公司提供，所提供的功能自然相当强大，除了支持硬件所具备的各项特性外，还在软件方面做出了许多功能扩展，比如多任务控制管理等。

嵌入式系统要做到支持全部硬件功能，势必需要一定数量的代码方可实现，对于任意一个实际产品可能只是用到硬件平台的部分功能，这样会造成程序空间的浪费，所以嵌入式系统除了功能支持外还需要支持功能的可选择，对于不需要的功能，可以通过规定的方法从系统中将相应代码移除掉，这就是嵌入式系统的可裁剪性。象 UC/OS 裁剪出最小内核只需要 2kBytes 的空间，而如果加上对硬件平台的功能支持、TCP/IP 等通讯协议的实现则需要几百 kBytes，如果是自己设计一个最简支持多任务控制的操作系统模型，一两百 kBytes 都可以实现。

提示：可以将嵌入式系统理解为带操作系统的单片机产品

嵌入式操作系统大都支持多任务，所以将嵌入式的操作系统称为多任务操作系统也是可以的。

如果用电脑来打比方的话，传统的单片机程序就是早期的 MSDOS，单任务，所有的系统资源都可归当前任务所有，嵌入式系统则是现在的 Windows，多任务，系统资源需要经由 Windows 来统一进行调度。

嵌入式系统因为设计思想大体与 Windows 相同，为了便于程序员理解系统以及进行程序开发，所以在驱动程序设计方面也是尽量与电脑方式一致，当程序员针对嵌入式系统进行开发或者移植工作时，会感觉到许多地方程序的规则、结构和风格和电脑非常相似。WinCE 是微软针对便携式电子产品推出的嵌入式操作系统，如果将 WinCE 放在一个有键盘和屏幕显示的电子产品上，会让使用者觉得这完全就是一个装了 Windows 的小电脑，甚至电脑上的程序只要做少量修改用 WinCE 的开发工具编译后就能直接在上面运行，有兴趣的朋友可以用采用 WinCE 的多普达等智能手机看看。

嵌入式系统为了便于扩展和平台移植，通常都是使用分层的方法设计系统，图示就是 eCos 的大体构架图，该图在 eCos 官方示意图的基础根据实际情况做出了一点修改，官方示意图左边的“设备驱动程序”并不与“目标硬件平台”相邻，这里改成了同时与“目标硬件平台”和“硬件抽象层”相邻。（相邻表示相互之间可以进行通讯）

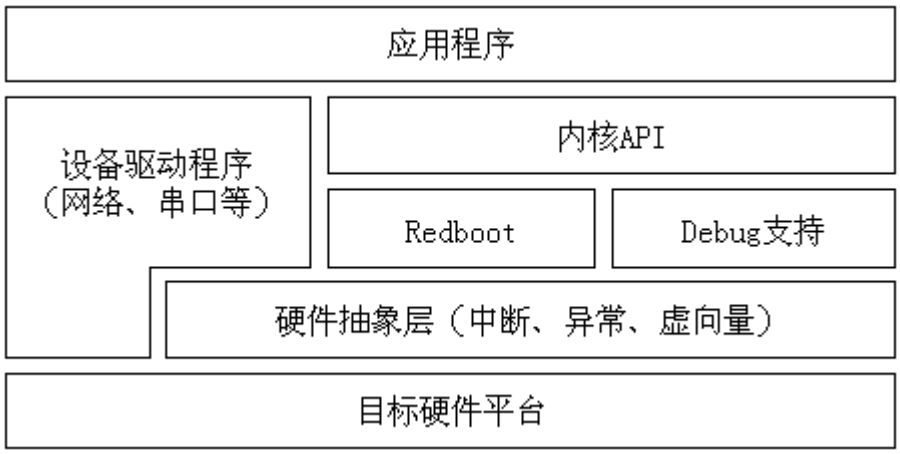

图 3.5.-1　eCos 操作系统构架示意图

如果严格遵循嵌入式操作系统的要求，应用程序应尽量避免直接操作硬件，而是要求按照统一规则编写驱动函数。来看一下 eCos 所提供的标准驱动函数是什么样子。

cyg_io_lookup(const char *name, cyg_io_handle_t *handle)

lookup 函数用来在设备表中查找 name 参数指定的设备，并在参数 handle 中返回句柄指针，如果没有找到指定设备，函数返回设备未找到的错误信息。

name 通常为“/dev/serial0”这种形式。

cyg_io_write(cyg_io_handle_t handle,void *buf,cyg_uint32 *len)

cyg_io_read(cyg_io_handle_t handle,void *buf,cyg_uint32 *len)

write 和 read 函数是对句柄所对应的设备进行写或读操作，buf 和 len 分别是数据缓冲区和数据长度，如果操作失败函数返回相应错误信息。

cyg_io_set_config(cyg_io_handle_t handle,cyg_uint32 key,void *buf,cyg_uint32 *len)

cyg_io_get_config(cyg_io_handle_t handle,cyg_uint32 key,void *buf,cyg_uint32 *len)

set 和 get config 函数对句柄对应的设备进行配置，其中 key 表示配置的具体类型，另外两个指针参数存放配置的具体参数。

如果对 VC 熟悉就会发现这几个函数和 VC 里面的 CreateFile()、ReadFile()和 WriteFile()去控制串口等外设的方式非常相似，都是通过设备名得到一个句柄，然后对句柄进行相应读写操作。

假定现在我们用 eCos 的系统函数控制硬件串口，在底层系统串口名被定义为“/dev/serial0”，该名称在底层系统是独立唯一的，与之对应有一系列的底层函数，这些底层函数可以直接完成对串口硬件的任意操作，但对于应用程序来说这些底层函数是不公开的，所以应用程序不能直接调用底层函数，只能通过前面的 eCos 标准接口函数来间接调用。

eCos 在内部建立有几张表，表应该包含下列内容（和实际有区别）：

设备名	/dev/serial0	/dev/serial1	/dev/spi	/dev/i2c	……
实际设备	硬件串口一	硬件串口二	硬件 SPI 口	硬件 I2C 口	……

图 3.5.-2 eCos 设备名关系示意表

key 值	函数名	功能
0	uart_set_bandrate()	设置波特率
1	uart_set_length()	设置数据位长度
2	uart_set_check_bit()	设置校验位方式
3	uart_set_stop_bit()	设置停止位方式
…	…	…

图 3.5.-3 eCos 串口底层函数示意图

当应用程序调用 eCos 所带的标准接口函数时，会先从设备名表中找出设备名“/dev/serial0”对应的设备为硬件串口一，然后依据 key 值知道所需要进行的具体操作，参数则通过接口函数中的指针传递进来，只需要应用程序和底层函数都按统一格式传递参数就可保证参数正确。

将这些底层函数融合到操作系统的过程叫做驱动程序编写，任何一个操作系统推出时本身都只会支持某几种型号的 MCU，如果想要操作系统可以在新的 MCU 上运行就需要编写针对新 MCU 的驱动程序，常见做法是以一个操作系统支持的 MCU 为蓝本，在其基础之上实现对新 MCU 的支持，这个工作被称为操作系统平台移植。

操作系统平台移植不是一项简单的工作，这一过程需要严格遵循操作系统制定的各项规则，参与移植工作的人需要不但需要对操作系统了解透彻，还要对新的 MCU 的各种硬件特性了如指掌，所以平台移植是一件费时费力的事情，少则三五个月，稍微慢点就可能需要一年半载，对于大多数公司来说这无法接受。

正是基于这个原因前面 eCos 系统的构架图我做了修改，目的是满足公司产品开发时间紧迫的要求，如果应用程序不需要调用其它软件公司针对操作系统提供的第三方软件，底层驱动就不一定必须完全按照操作系统的规则进行编写。

第三方软件是软件公司针对某一类功能提供的专业软件包，也就是适用于应用层的上层驱动。为了通用，第三方软件所调用的接口函数必须完全满足操作系统所制定的规则。比如现在有一家第三方公司针对 eCos 编写了上层图形显示的 API，用户通过其提供的 API 可以非常方便的显示各种图形，第三方公司并不知道图形显示的具体方式，只是将处理好的数据通过 cyg_io_write()交给用 cyg_io_lookup()得到的显示设备句柄，如果第三方公司的 API 需要知道显示设备所能支持的数据格式、显示宽高等信息可以通过 cyg_io_get_config()得到。要支持这家公司的 API 显示驱动程序就必须按照 eCos 的要求编写，显示设备可以是“/dev/lcd”，也可以是“/dev/tv”，但都必须封装成 eCos 的标准形式。

不是所有的应用程序都需要使用此类第三方软件，这时可以采用一种简化方法。因为操作系统对任务调度、中断管理这些操作主要是以软件为主，硬件只提供少量最基本的支持，所以这部分移植工作量相对要简单一些，我们只是将这部分代码按操作系统要求进行移植，对于设备驱动程序不进行移植。设备驱动程序还是按照传统的方法编写代码，提供下面的驱动函数。

lcd_init()

lcd_set_config()

lcd_get_con()

lcd_write_data()

应用程序需要显示的时候直接调用这些驱动函数，不过编写这些驱动函数时要留意防止不同任务同时调用同一个驱动函数，在函数内部需要有保护代码。这种方式可以将开发时间大为缩短，不足是不满足操作系统的通用性原则，象第三方公司提供的 API 就无法使用。

总体说掌握嵌入式操作系统还是一件较有难度的事情，单凭一两篇文章就想弄清楚操作系统是不现实的，理解操作系统细节唯一的方法是阅读理解其提供的源代码，有一定基础知识之后自己再尝试移植一个简单的系统，并写出一两个符合操作系统要求的驱动程序，当完这些事情都你就会发现嵌入式操作系统已经嵌入到你的头脑之中。

嵌入式误区之不死机

嵌入式系统一词在国内流行开的时间并不长，刚开始并不叫嵌入式，我记得九十年代末还习惯被称为RTOS（实时多任务操作系统）。自我第一次接触到RTOS，给我印象最深刻的是其宣称的高可靠性不死机的超强特性，培训中无论是 RTOS 软件公司的市场和技术人员还是本公司的前辈都一再强调这一点，使得刚参加工作的我虽然有些不理解但不敢质疑。

记得当时讲解如何实现不死机这一特性主要是依靠RTOS 公司的专业性和 RTOS 的多任务机制得以实现。

首先 RTOS 是由专业软件公司开发完成的，这样的公司技术人员具有丰富的经验，所以写出来的程序可靠性高，出错的几率比较小，而我们这样的技术人员尤其是刚参加工作的，经验欠缺，所写的程序自然容易出错，这一点想想确实如此，没有可质疑的地方。

其次一个 RTOS 在推出之前经过了严格的测试，进一步降低了出错的几率，一般公司写的程序都没有经过如此严格的测试，这一点好象也是那么回事。

最后一点是不用 RTOS 写的程序为单任务模式，通常程序需要完成键盘扫描、显示和功能控制等操作，这些功能模块是串联模式，一旦有一个因异常导致死循环，整个循环就会一同死掉，如果是多任务可以将这些功能分别放在不同任务当中，即使有死循环产生其它任务还能继续工作，不至于死掉，好象说得也有道理。

解释完负责培训的人员还会列举使用 RTOS 的例子：美国的航天飞机控制程序必须采用RTOS，美国军方的一些控制设备也要求使用RTOS，意思就是世界上使用技术最先进、对产品质量要求最高的地方都是要用RTOS。到这个时候就是再多的半信半疑也只能是咽到肚子里去，总不能跑到美国去求证真伪。

我的记忆可能不准确，就算没有记错也有可能是当时培训的人员为了推荐产品自己有一些夸张，或者是他自己理解不够准确，所以我前面的叙述并不能肯定 RTOS 不死机的说法真的存在。这里我从网络中摘录了一些关于 RTOS 的概念陈述，发现大都同样有提到不死机的特性，看来这一说话确实存在。

1.实时多任务操作系统(RTOS)

(1)更加面向硬件系统，而不是操作者

嵌入式系统处理器一般都是独立工作的，没有人的直接参与；即使参与，也没有大量的文字信息输出，这是和桌面计算机有所不同的。因此 RTOS 着重面向的是硬件，而不是具有完整的人机界面。

*(2)**实时性***

单片机系统的监测、控制、通信等工作都要求实时性，一旦出现有关情况，CPU 能够及时响应，刻不容缓。为此，一个实用的RTOS 都应具有完善的中断响应机制，保证中断响应潜伏时间足够短。

(3)多任务

半导体技术的发展和应用复杂性的增长促使CPU的处理能力越来越高，当今的一片16位或32位单片机，在运算速度、寻址能力等方面可以相当于8位单片机的几十片之和。在这样强大的处理器上运行应用程序，必然不是整块，而是根据所要实现的若干方面功能，划分为数个任务，这样有利于软件的开发和维护。

因此单片机系统中采用的RTOS必然是支持多任务的，并能够根据各个任务的轻重缓急，合理地在它们之间分配CPU和各种资源的占用时间。

(4)不同的典型外设驱动支持

单片机的典型片内外设为定时器、A/D、PWM、D/A、串行口、LCD/LED接口,CAN-bus、IC-bus等。根据处理器类型的不同，RTOS在出厂时一般附带若干上面硬件接口的驱动程度，而网卡等片外设备的驱动程序，以及其它一些高级驱动函数，如兼容DOS的文件系统、TCP/IP协议等，则需要另行选购。以RTOS为基础和接口标准，可以设计出大量的库函数驱动模块，并根据实际需要选择或裁剪。

*(5)**高可靠性***

*一般计算机的操作系统出现问题，例如**死机**，除数据丢失等外，不会有太大的问题；而单片机系统一般都是和工业控制、交通工具、医用器械等机电系统密切相关，不适当的输出甚至不及时的输出都可能会带来财产损失和安全问题。因此嵌入式系统中的RTOS要求高可靠性，发行之前必须经过严格的测试。这是一个耗费时间和精力的过程，也是RTOS价格普遍高于一般操作系统的原因之一。*

2.RTOS是一个内核

典型的单片机程序在程序指针复位后，首先进行堆栈、中断、中断向量、定时器、串行口等接口设置、初始化数据存储区和显示内容，然后就来到了一个监测、等待或空循环，在这个循环中，CPU可以监视外设、响应中断或用户输入。

这段主程序可以看作是一个内核，内核负责系统的初始化和开放、调度其它任务，相当于C语言中的主函数。

RTOS就是这样的一个标准内核，包括了各种片上外设初始化和数据结构的格式化，不必、也不推荐用户再对硬件设备和资源进行直接操作，所有的硬件设置和资源访问都要通过RTOS核心。硬件这样屏蔽起来以后，用户不必清楚硬件系统的每一个细节就可以进行开发，这样就减少了开发前的学习量。

一般来说，对硬件的直接访问越少，系统的可靠性越高。RTOS是一个经过测试的内核，与一般用户自行编写的主程序内核相比，更规范，效率和可靠性更高。对于一个精通单片机硬件系统和编程的“老手”而言，通过RTOS对系统进行管理可能不如直接访问更直观、自由度大，但是通过RTOS管理能够排除人为疏忽因素，提高软件可靠性。

*另外，**高效率**地进行多任务支持是RTOS设计从始至终的一条主线，采用RTOS管理系统可以统*

一协调各个任务，优化 CPU 时间和系统资源的分配，使之不空闲、不拥塞。针对某种具体应用，精细推敲的应用程序不采用 RTOS 可能比采用 RTOS 能达到更高的效率；但是对于大多数一般用户和新手而言，采用 RTOS 是可以提高资源利用率的，尤其是在片上资源不断增长、产品可靠性和进入市场时间更重要的今天。

3.RTOS 是一个平台

RTOS 建立在单片机硬件系统之上，用户的一切开发工作都进行于其上，因此它可以称作是一个平台。采用 RTOS 的用户不必花大量时间学习硬件，和直接开发相比起点更高。

RTOS 还是一个标准化的平台，它定义了每个应用任务和内核的接口，也促进了应用程序的标准化。应用程序标准化后便于软件的存档、交流、修改和扩展，为嵌入式软件开发的工程化创造了条件、减少开发管理工作量。嵌入式软件标准化推广到社会后，可以促进软件开发的分工，减少重复劳动，近来出现的建立于 RTOS 上的文件和通信协议库函数产品等就是实例。

RTOS 对于开发单位和开发者个人来说也是一种提高。引入 RTOS 的开发单位，相当于引入了一套行业中广泛采用的嵌入式系统应用程序开发标准，使开发管理更简易、有效。基于 RTOS 和 C 语言的开发，具有良好的可继承性，在应用程序、处理器升级以及更换处理器类型时，现存的软件大部分可以不经修改地移植过来。

对于开发人员来说，则相当于在程序设计中采用一种标准化的思维方式，提高知识创造的效率；同时因为具有类似的思路，可以更快地理解同行其它人员的创造成果。

4.RTOS 产生并得到迅速发展的原因

单片机处理器能力的提高和应用程序功能的复杂化、精确化，迫使应用程序划分为多个重要性不同的任务，在各任务间优化地分配 CPU 时间和系统资源，同时还要保证实时性。靠用户自己编写一个实现上述功能的内核一般是不现实的，而这种需求又是普遍的。在这种形势之下，由专业人员编写的、满足大多数用户需要的高性能 RTOS 内核就是一种必然结果了。

对程序实时性和可靠性要求的提高也是 RTOS 发展的一个原因。此外，单片机系统软件开发日趋工程化，产品进入市场时间不断缩短，也迫使管理人员寻找一种有利于程序继承性、标准化、多人并行开发的管理方式。从长远的意义上来讲，RTOS 的推广能够带来嵌入式软件工业更有效、更专业化的分工，减少社会重复劳动、提高劳动生产率。

5.RTOS 的基本特征

(1)任务

任务(Task)是 RTOS 中最重要的操作对象，每个任务在 RTOS 的调用下由 CPU 分时执行。激活的或当前任务是 CPU 正在执行的任务，休眠的任务是在存储器中保留其执行的上下文背景、一旦切换为当前任务即可从上次执行的末尾继续执行的任务。任务的调度目前主要有时间分片式(TimeSlicing)、轮流查询式(Round-Robin)和优先抢占式(Preemptive)三种，不同的 RTOS 可能支持

其中的一种或几种，其中优先抢占式对实时性的支持最好。

(2)任务切换

RTOS 管理下的系统 CPU 和系统资源的时间是同时分配给不同任务的，这样看起来就象许多任务在同时执行，但实际上每个时刻只有一个任务在执行，也就是当前任务。任务的切换有两种原因。当一个任务正常地结束操作时，它就把 CPU 控制权交给 RTOS，RTOS 则检查任务队列中的所有任务，判断下面那个任务的优先级最高，需要先执行。另一种情况是在一个任务执行时，一个优先级更高的任务发生了中断，这时 RTOS 就将当前任务的上下文保存起来，切换到中断任务。RTOS 经常性地整理任务队列，删除结束的任务，增加新的要执行任务，并将其按照优先级从大到小的顺序排列起来，这样可以合理地在各个任务之间分配系统资源。

(3)消息和邮箱

消息(Message)和邮箱(Mailbox)是 RTOS 中任务之间数据传递的载体和渠道，一个任务可以有多个邮箱。通过邮箱，各个任务之间可以异步地传递信息，没有占用 CPU 时间的查询和等待。当 RTOS 包含片上总线接口驱动功能时，各个单片机之间的通信也通过邮箱的方式来进行，用户并不需要了解更深的关于硬件的内容。

(4)旗语

旗语(Semaphore)相当于一种标志(Flag)，通过预置,一个事件的发生可以改变旗语。一个任务可以通过监测旗语的变化来决定其行动，在监测旗语变化的时候不消耗 CPU 时间，旗语对任务的触发是由 RTOS 来完成的。通过使用旗语，一个任务在等待事件变化的时候就可以不必不断查询，而把 CPU 时间出让给其它任务。

(5)存储区分配

RTOS 对系统存储区进行统一分配，分配的方式可以是动态的或静态的，每个任务在需要存储区时都要向 RTOS 内核申请。RTOS 通过使用存储分配类核心对象管理数据存储器，在动态分配时能够防止存储区的零碎化。

(6)中断和资源管理

*RTOS 提供了一种通用的设计用于中断管理，有效率而灵活，这样可以实现最小的中断潜伏时间和最大的中断响应度。RTOS 内核中的资源对象类则实现了对系统实体资源或虚拟资源的独占式访问，一个任务可以取得对资源的唯一访问权，其它任务在资源释放以前无法访问，这样可以避免资源冲突。设计完善的 RTOS 具有检查可能导致**系统死锁**的资源调用设计。*

阅读完这段陈述除了知道 RTOS 强调自己的高可靠性不死机外，还表明它实际上就是嵌入式操作系统，只是表述方法不同而已。

那 RTOS 到底有没有具备不死机的超强特性呢？这种说法在我看来是错误的，至少可以说不够严谨。可以说无论是硬件还是软件，目前还不存在完全解决了死机问题的方法，所有产品都只是尽可能的降低死机的几率，不可能将死机的几率降到零。

死机的原因五花八门，可能是硬件的，也可能是软件的，如果是硬件原因导致，单纯依靠软件

是无法解决的，而 RTOS 只是软件方面的产品，凭这一点就可以说 RTOS 不死机言过其实。在软件层面，RTOS 只是提供了一个程序框架，并不包含有实际功能的应用程序，要用到产品中就需要工程师在此框架基础上编写相应应用程序，就算 RTOS 自我非常完善，可应用程序的内容还是由应用工程师决定，如果应用程序出错 RTOS 同样无能为力。

虽然 RTOS 可以采用某些方式对系统自身进行保护，但程序运行起来后始终会出现 CPU 完全由应用程序控制的状态，这个时候 RTOS 如果没有硬件特殊功能（MPU）的支持，在有问题的应用程序面前同样如同一只任人宰割的羔羊。

```
UINT32 *p,i;
i=0x00000000;                  //i 进行这样一段复杂的运算是为了避免编译器优化
i=i+0x00001234;
i=i+0x56780000;
i=i&0x0000C1C1;                //到这里 i 的实际结果等于 0
p=(UINT32 *)i;                 //所以 p 这里也指向地址 0
for(i=0;i<0x00100000;i++)      //将从地址 0 开始的 4MBytes 空间清 0
{
  *p=0x00000000;
  p++;
}
```

通常 OS 都位于存储器从地址 0 开始的一段区域，如果执行这段代码会将 OS 所在区域清 0，也就是 OS 自身被应用程序破坏掉，试想 RTOS 面对这样的代码何以保证高可靠性？当然这样的代码是不允许存在的，但可以说明一个问题，应用程序在使用指针时如果指针出错，就存在将整个 RTOS 摧毁的可能。

对于编写应用程序的工程师，如果是在 RTOS 上进行编程，就应用程序本身来说，出错的几率和不使用 RTOS 是相同的，程序的质量主要靠工程师的素质来把握，和 RTOS 关系不大。因为 RTOS 多少对程序编写存在一些限制，所以对于工程师实际上会增加一些额外的负担，需要了解 RTOS 的相关知识，而且程序调试会因为 RTOS 的存在要麻烦一些。

所以 RTOS 高可靠性、不死机的说法是不正确的。实际上 RTOS 是适合逻辑流程相对复杂、而且需要同时处理多项事情的程序，如果是单任务方式，程序就需要非常多的判断、跳转操作，即便是经验丰富的工程师也可能会被过多的逻辑流程控制把自己绕晕，有了 RTOS 可以将不同的事情处理分离到独立的任务当中，任务之间通过 RTOS 传递交换数据，逻辑流程自然就明晰起来。

我们不用 RTOS 也可以实现 RTOS 类似的工作，比如我们可以在将每一个事项的处理分成许多小段程序，然后利用定时中断来依次执行每个事项的分段程序。这样做需要中断程序具备管理功能，以保证每次中断正确调用相应程序分段，这里的中断程序相当于一个功能最简的 RTOS。虽然中断程

序只是实现简单的管理功能，实际要做好并不容易。现有的 RTOS 正是解决了这个问题，将任务的管理工作很稳定的实现，高可靠性指的是这一点。

RTOS 不但在任务管理方面可靠性高，所提供的功能也是相当强大，几乎考虑了所有的用户需求，另外模块式的程序结构可以让用户自由进行功能裁剪，让用户制定出适合自己且经济高效的系统，应用层标准化接口更加有利技术的分工合作，使得软件公司开发基于 RTOS 的标准功能库成为现实。

所以不要因为我反对了 RTOS 的某一个宣传说法有所夸大就全面否定 RTOS，它的优点是符合技术发展的潮流趋势，现在不少产品都不能全靠自己的力量完成，适当引用其它公司的现有技术可以让产品开发周期和质量都得到提升。RTOS 确实是一个好东西，就好比 WINDOWS，和 MSDOS 相比虽然复杂许多，而且需要功能更强的硬件支持，但能给用户带来完全不一样的感受。

使用 RTOS，就如同站在巨人的肩膀上看世界，只是爬上巨人的肩膀需要多花一些力气。

嵌入式效率

嵌入式系统一贯宣称自己的实时、高效，在我看来这一点也是不正确的，最高效率是由硬件决定的，一旦硬件平台确定，任何软件都无法突破其最高效率。

也许有人会说软件高手编写的程序效率会比普通软件人员写的效率会高，嵌入式操作系统都是软件高手来完成的，说效率高很正常啊。这种说法采用的是概念转移的伪证法，将一个硬件平台所能达到的软件最高效率问题转换成不同程序员编写的代码效率高低。我们很容易将这种说法辩驳倒，让同一个写操作系统的程序员用两种方法来完成产品代码的编写，一种是程序员在操作系统之上编写，另外一种是程序员直接针对硬件编写，显然后一种方法得到的程序效率要高。

嵌入式操作系统是软件，软件的运行就要消耗 CPU 资源，同一个 CPU 其能提供的资源是恒定的，既然嵌入式系统运行耗费了部分资源，对于应用程序来说可用的资源就要减少，所以能达到的最大效率也自然要相应降低。另外操作系统为了实现自己的管理功能，需要打开 TIMER 中断为整个操作系统提供同步时钟信号，这些中断程序可能对产品实际功能并无多大作用，但要占用 CPU 运行时间，如果不用操作系统可以关闭掉。所以同样功能的软件，带操作系统的效率肯定要低过带操作系统的版本。

同样道理，硬件平台对于事件的最快响应时间也是恒定的，嵌入式样操作系统在快速实时性方面实际上并不理想，因为操作系统的构架方式对事件的响应会明显慢过硬件所支持的速度。因为中断是操作系统直接管理的资源，操作系统在中断向量表中放置的并不是中断程序服务程序的地质，而是操作系统的中断管理函数入口地址，当硬件中断信号产生后，CPU 不是直接执行相应中断服务程序，而是先执行操作系统的总中断管理函数，在该函数中再由软件决定何时执行中断服务程序。这种模式使得操作系统对中断的响应效率大打折扣。操作系统为了通用，所提供的接口都是 C 语言形式，C 语言一般来说代码效率没有汇编高，所以操作系统编程语言的选择又使得系统对中断的响应速度更慢了一些。

所以单纯从代码执行效率来看，嵌入式系统并没有高效的特点，这个方面反而是嵌入式系统的不足。但如果从另外一个角度来理解，也还是可以承认这种说法。现在硬件的速度越来越快，硬件速度的提升弥补了嵌入式系统运行效率的不足。嵌入式系统对多任务的支持，可以让原本复杂的逻辑流程变简单，加上基操作系统的大量第三方标准软件的出现，许多工作都可以直接使用别人的现有成果，在整个产品的开发角度来说无疑是更加高效的方法。

第四章　单片机 C 语言

终于结束了晦涩枯燥的第三章，我自己也长吁了一口气，现在我真的是非常同情那些教专业基础课或者工程数学的老师，这里真诚的说一声：“辛苦你们了”。

这一章要轻松不少，相信就算是刚走出校门的雏鸟，多少都有一定的 C 语言基础，大学好象都要过一个计算机等级考试，所以 C 语言自然是逃不了。你不要指望我给你讲述 C 语言原理和指令这类基础知识，我更不会给你讲述 C++那些面对对象编程的高级编程方法，这一章讲述的内容都是 C 语言在单片机上应用会遇到的一些有意思的现象，让你知道 C 在单片机上是怎么工作的。

当然也会告诉你一些 C 的经验技巧，这些对提升你的单片机程序能力还是有一定作用的。

4.1. 单片机 C 语言

早期单片机编程是没有 C 语言支持的，都是汇编甚至是二进制的机器码，随着电脑技术的突飞猛进，单片机编程不再安于汇编的一亩三分地，也向着 C 语言的方向进发。理论上讲单片机实现 C 语言编程不存在丝毫问题，毕竟和电脑是同根生，于是一批专业或非专业、有着利益目的或无利益目的的工程师开始了这方面的努力。

和电脑最大的不同是单片机种类繁多，不象电脑只有那么几种芯片，而且电脑 CPU 的发展遵循着一定的规则，不同 CPU 要求做到指令兼容，单片机做这样的要求显然不现实，厂商不可能接受都遵循制定标准设定 MCU 的要求。虽然单片机种类繁多，但大部分单片机还是会采用通用构架进行设计，毕竟遵循一定标准可以不用厂商自己去完成指令系统、编译工具等繁琐工作，所以市面上流行的单片机内核其实并不多，不少八位的单片机都采用 51 内核，高端的 MCU 内核更是集中在 ARM/MIPS…这几种当中。

厂商设计的 MCU 通常都会沿用某一种构架，也就是厂商产品目录中的 xx 系列，这样做厂商可以节省开发成本，一套编译器可以为一个甚至多个系列的 MCU 所用，这样新设计 MCU 或编译器有问题也可以在日后进行改进，如果弄成一种 MCU 就对应一套编译器的方式，神仙也会疯掉。厂商为了占领更多的市场，自然就会依据市场需求针对 MCU 推出 C 的编译器，不过这种做法所退出的 C 编译器质量局限于厂商自己技术能力，通常说这类编译器可以用，但不要期望有着很高的效率。如果是流行面广的内核，会有另外一种方式，就是专业的软件公司针对这种内核的指令系统开发 C 编译器，象 KEIL C 就是一例，这种软件公司在编译方面经验丰富，所以他们做出来的编译器效率方面相当不错，只要是他们编译器支持的内核，就很容易让编译器支持。软件公司推出的 C 编译器虽然好，但要钱，有免费的版本可限制多多，技术世界从来不缺少活雷锋，GCC 这样的组织让免费获取 C 编

译器成为了现实，不过这类组织所支持的对象只能是内核为 ARM/MIPS…的高端通用 MCU。

想要做好单片机的 C 编译器则必须具备这两个条件：一是熟悉 MCU 的硬件资源和指令系统，二是熟悉 C 语言，两者缺一不可，否则是做不出一个优质高效的单片机 C 编译器的。编译器的工作就是将用 C 写的代码按一定规则转换成汇编指令，这样程序员面对的是接近自然语言的 C 代码，对程序的结构控制、含义理解等会容易不少。由于转换操作依赖编译器，虽然一个编译器需要经过大量测试才会推出，但测试无法涵盖所有的编程可能，这样编译器并不能保证可靠性为百分百，一旦有错误产生，调试会麻烦许多，毕竟错误不是程序员而是编译器产生的，在 C 语言层面会让错误弄得一头雾水，当然程序员对 C 和汇编都很熟悉的话还是可以通过查看汇编代码的方式来查找编译器错误。

同电脑的 C 相比单片机的 C 编程存在自己的特点：电脑用 C 写程序奉行的是硬件无关的原则，程序员只要了解 C 的语法就可以，就是深入到驱动程序层面也只需要了解驱动程序的接口就够，单片机则不然，C 只是让程序员面对的代码不再是汇编格式，程序编写依然还是要了解硬件特性，只是将原本由汇编写的硬件控制代码改成了 C 的语法格式；为了最大程度的利用单片机的各种指令，单片机的 C 编译器同电脑的 C 编译器相比可能会有多不同，比如对某些 C 语法做出修改，象 KEIL C 对 51 系列的单片机就多出了位变量的定义和操作的语法；单片机结构要比电脑简单，所提供的资源也要少许多，希望能支持 C 编程也主要是为了让程序结构简单明晰，所以 C 的控制流程语法就已经够用，并不需要象电脑一样在标准 C 的基础上继续类似 C++各种改进。但也不是绝对，现在各种嵌入式平台也还是尽力向电脑方向靠拢，这样做我的理解是众人拾柴火焰高，嵌入式已经发展到了需要许多人合作才能实现的阶段。

单片机用 C 编程便捷性无疑是大为提高，可当用 C 实现对单片机支持后新问题出现，这就是目前我们国家的现状是做单片机的大多是电子信息类的专业出生，在学习阶段以控制方面的知识为主，C 只是做为辅助课程出现，更不用说软件工程之类的课程，这样当这部分人由学生转入工作时容易写出汇编 C 代码，就是语法是 C 而程序风格和思路是汇编，在计算机专业出身的人眼里看为垃圾代码，如果本身是计算机专业出身去写单片机 C 程序又会面临电子专业基础不足的问题。这一章我会告诉习惯汇编的单片机程序员一些 C 的经验和技巧，相信通过这章的学习会让你对 C 的单片机编程有更深的认识。

4.2. for()和 while()循环

不同程序员都有自己的编程风格，让一个程序员用代码实现死循环，C 出身的程序员习惯会用 for(;;)和 while(1)，而汇编出身的程序员则习惯用 goto loop_label，风格上的差异都可以接受，但如果深入到代码效率层面就会发现不同的循环方式效率也会存在差别。

用 C 语言写出循环 100 次的九种不同实现方式：

```
void MCU_CTest(void)
```

```
{
  unsigned long i;
  unsigned long vTemp;
  //------loop mode 1------
  //------loop mode 2------
  //------loop mode 3------
  //------loop mode 4------
  //------loop mode 5------
  //------loop mode 6------
  //------loop mode 7------
  //------loop mode 8------
  //------loop mode 9------
  //------end flag------
  vTemp=0;
}
```

方式一 C 代码：

```
  vTemp=0;
  for(i=0;i<100;i++)
  {
    vTemp=vTemp+i;
  }
```

方式一汇编代码说明：

```
  0x1dc0:  MOV    R1,#0        ;vTemp=0，第一种循环
  0x1dc4:  MOV    R0,#0        ;i=0，循环次数清零
  0x1dc8:  CMP    R0,#0x64     ;将 i 和 100 比较
  0x1dcc:  BCS    0x1de4       ;i 大于等于 100 则跳到地址 0x1de4，结束循环
  0x1dd0:  B      0x1ddc       ;跳转到地址 0x1ddc 好进行 vTemp=vTemp+i
  0x1dd4:  ADD    R0,R0,#0x1   ;i++
  0x1dd8:  B      0x1dc8       ;跳转到地址 0x1dc8 好判断 i 是否小于 100
  0x1ddc:  ADD    R1,R1,R0     ;vTemp=vTemp+i
  0x1de0:  B      0x1dd4       ;跳转到地址 0x1dd4
  0x1de4:  MOV    R1,#0        ;vTemp=0，第二种循环开始
```

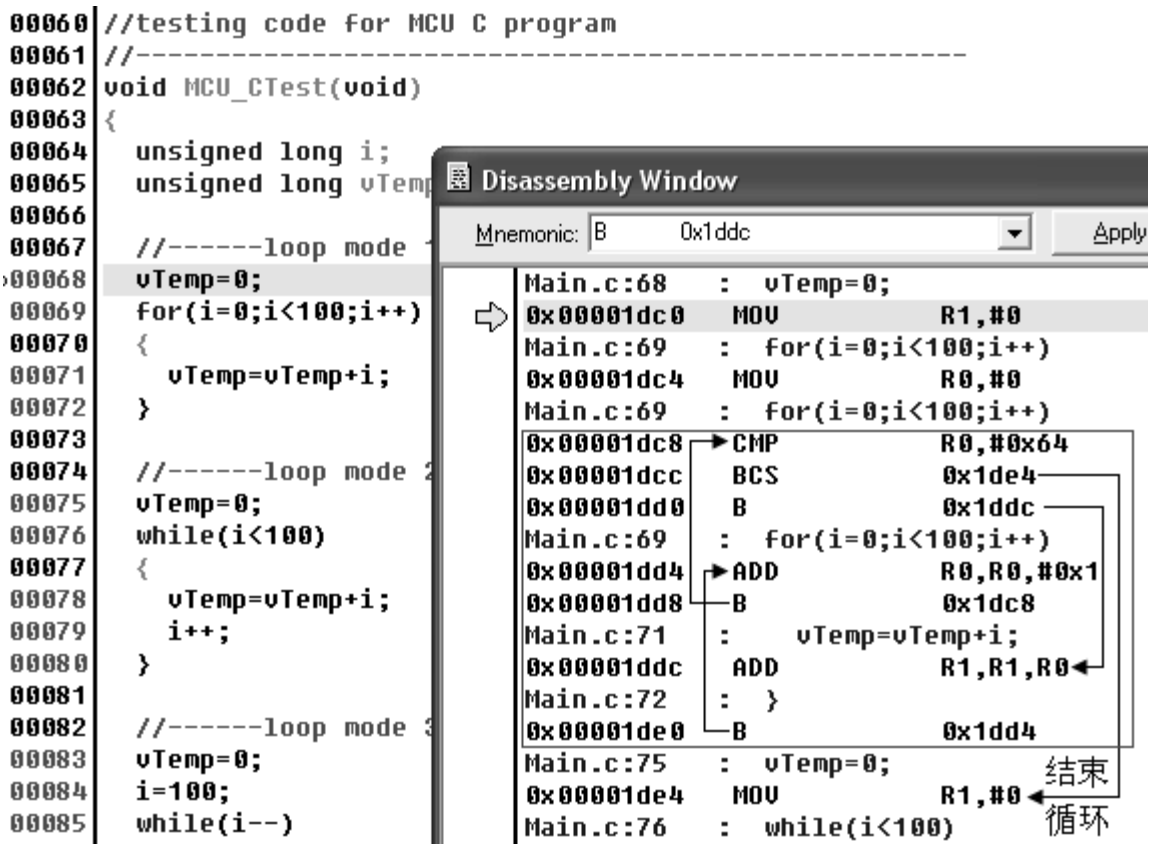

图 4.21.-1 ARM 汇编结果示意图一

方式一汇编代码执行流程：

```
0x1dc0:  MOV    R1,#0        ;vTemp=0，第一种循环
0x1dc4:  MOV    R0,#0        ;i=0，循环次数清零
0x1dc8:  CMP    R0,#0x64     ;将 i 和 100 比较，此时 i 为 0，第一次循环
0x1dcc:  BCS    0x1de4       ;比较结果不用跳转
0x1dd0:  B      0x1ddc       ;跳转到地址 0x1ddc
0x1ddc:  ADD    R1,R1,R0     ;vTemp=vTemp+i
0x1de0:  B      0x1dd4       ;跳转到地址 0x1dd4
0x1dd4:  ADD    R0,R0,#0x1   ;i++
0x1dd8:  B      0x1dc8       ;跳转到地址 0x1dc8
0x1dc8:  CMP    R0,#0x64     ;将 i 和 100 比较，此时 i 为 1，第二次循环
0x1dcc:  BCS    0x1de4       ;比较结果不用跳转
0x1dd0:  B      0x1ddc       ;跳转到地址 0x1ddc
0x1ddc:  ADD    R1,R1,R0     ;vTemp=vTemp+i
0x1de0:  B      0x1dd4       ;跳转到地址 0x1dd4
0x1dd4:  ADD    R0,R0,#0x1   ;i++
```

```
0x1dd8:  B      0x1dc8      ;跳转到地址 0x1dc8
0x1dc8:  CMP    R0,#0x64    ;将 i 和 100 比较，此时 i 为 2，第三次循环
……
```

方式一执行一次循环需要 7 条指令，其中真正跳转 3 次。

方式二 C 代码：

```
vTemp=0;
i=0;
while(i<100)
{
  vTemp=vTemp+i;
  i++;
}
```

方式二汇编代码说明：

```
0x1de4:  MOV    R1,#0       ;vTemp=0，第二种循环
0x1de8:  MOV    R0,#0       ;i=0，循环次数清零
0x1dec:  NOP                ;空操作
0x1df0:  CMP    R0,#0x64    ;将 i 和 100 比较
0x1df4:  BCS    0x1e04      ;i 大于等于 100 则跳到地址 0x1de4，结束循环
0x1df8:  ADD    R1,R1,R0    ;vTemp=vTemp+i
0x1dfc:  ADD    R0,R0,#0x1  ;i++
0x1e00:  B      0x1df0      ;跳转到地址 0x1df0
0x1e04:  MOV    R1,#0       ;vTemp=0，第三种循环开始
```

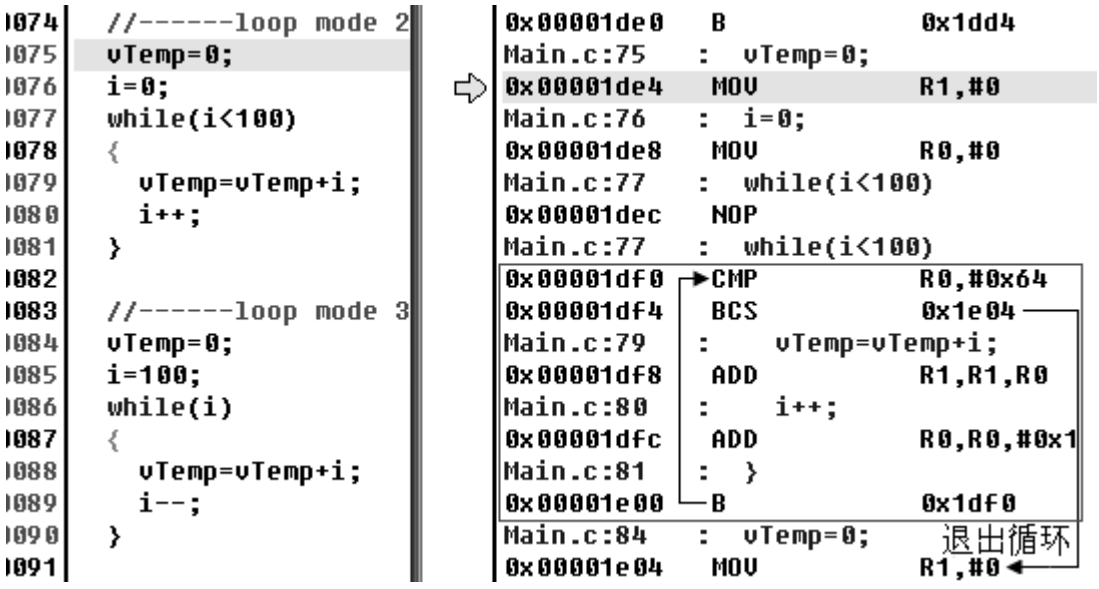

图 4.2.-2 ARM 汇编结果示意图二

方式二汇编代码执行流程比方式一要简单，执行阴影部分循环只需要 5 条指令，其中真正跳转 1 次。

为节约篇幅其它方式只将汇编代码整理出来，不贴图加以说明。

方式三 C 代码：

```
vTemp=0;
i=0;
while((i++)<100)
{
  vTemp=vTemp+i;
}
```

方式三汇编代码说明：

```
0x1e04:  MOV   R1,#0        ;vTemp=0，第三种循环
0x1e08:  MOV   R0,#0        ;i=0，循环次数清零
0x1e0c:  NOP                ;空操作
0x1e10:  MOV   R2,R0        ;将 i 的值用中间变量保存起来
0x1e14:  ADD   R0,R0,0x1    ;i++
0x1e18:  CMP   R2,#0x64     ;未加之前的 i 和 100 进行比较
0x1e1c:  BCS   0x1e28       ;大于等于 100 跳转到地址 0x1e28，结束循环
0x1e20:  ADD   R1,R1,R0     ;vTemp=vTemp+i
0x1e24:  B     0x1e10       ;跳转到地址 0x1e10
0x1e28:  MOV   R1,#0        ;vTemp=0，第四种循环开始
```

方式三汇编代码执行阴影部分循环需要 6 条指令，其中真正跳转 1 次。

方式四 C 代码：

```
vTemp=0;
i=0;
do
{
  vTemp=vTemp+i;
}while((i++)<100);
```

方式四汇编代码说明：

```
0x1e28:  MOV   R1,#0          ;vTemp=0，第四种循环
0x1e2c:  MOV   R0,#0          ;i=0，循环次数清零
0x1e30:  NOP                  ;空操作
0x1e34:  ADD   R1,R1,R0       ;vTemp=vTemp+i
0x1e38:  MOV   R2,R0          ;将 i 未加之前的值保存起来
0x1e3c:  ADD   R0,R0,#0x1     ;i++
0x1e40:  CMP   R2,#0x64       ;将未加之前的 i 和 100 进行比较
0x1e44:  BCC   0x1e34         ;小于 100 跳转到地址 0x1e34
0x1e48:  MOV   R1,#0          ;vTemp=0，第五种循环开始
```

方式四汇编代码执行阴影部分循环需要 5 条指令，其中真正跳转 1 次。

方式五 C 代码：

```
vTemp=0;
i=0;
do
{
  vTemp=vTemp+i;
  i++;
}while(i<100);
```

方式五汇编代码说明：

```
0x1e48:  MOV   R1,#0          ;vTemp=0，第五种循环
0x1e4c:  MOV   R0,#0          ;i=0，循环次数清零
0x1e50:  NOP                  ;空操作
0x1e54:  ADD   R1,R1,R0       ;vTemp=vTemp+i
0x1e58:  ADD   R0,R0,#0x1     ;i++
0x1e5c:  CMP   R0,#0x64       ;将未加之前的 i 和 100 进行比较
0x1e60:  BCC   0x1e54         ;小于 100 跳转到地址 0x1e54
0x1e64:  MOV   R1,#0          ;vTemp=0，第五种循环开始
```

方式五汇编代码执行阴影部分循环需要 4 条指令，其中真正跳转 1 次。

方式六 C 代码：

```
vTemp=0;
i=100;
while(i--)
{
```

```
    vTemp=vTemp+i;
  }
```

方式六汇编代码说明：

```
  0x1e64:  MOV   R1,#0          ;vTemp=0，第六种循环
  0x1e68:  MOV   R0,#0x64       ;i=100，循环次数设置为 100
  0x1e6c:  NOP                  ;空操作
  0x1e70:  SUB   R2,R0,#0x1     ;i--，减的结果放到到中间变量中
  0x1e74:  MOV   R0,R2          ;i--，将减得的结果取回
  0x1e78:  CMN   R2,#0x1        ;将 i 和 1 进行比较
  0x1e7c:  BEQ   0x1e88         ;等于跳转到地址 0x1e88，结束循环
  0x1e80:  ADD   R1,R1,R0       ;vTemp=vTemp+i
  0x1e84:  B     0x1e70         ;跳转到地址 0x1e70
  0x1e88:  MOV   R1,#0          ;vTemp=0，第七种循环开始
```

方式六汇编代码执行阴影部分循环需要 6 条指令，其中真正跳转 1 次。

方式七 C 代码：

```
  vTemp=0;
  i=100;
  while(i)
  {
    vTemp=vTemp+i;
    i—;
  }
```

方式七汇编代码说明：

```
  0x1e88:  MOV   R1,#0          ;vTemp=0，第七种循环
  0x1e8c:  MOV   R0,#0x64       ;i=100，循环次数设置为 100
  0x1e90:  NOP                  ;空操作
  0x1e94:  CMP   R0,#0x0        ;将 i 和比较
  0x1e98:  BEQ   0x1ea8         ;比较结果相等跳转到地址 0x1ea8，结束循环
  0x1e9c:  ADD   R1,R1,R0       ;vTemp=vTemp+i
  0x1ea0:  SUB   R0,R0,#0x1     ;i--
  0x1ea4:  B     0x1e94         ;跳转到地址 0x1e94
  0x1ea8:  MOV   R1,#0          ;vTemp=0，第八种循环开始
```

方式七汇编代码执行阴影部分循环需要 5 条指令，其中真正跳转 1 次。

方式八 C 代码：

```
vTemp=0;
i=100;
do
{
  vTemp=vTemp+i;
}while(i--);
```

方式八汇编代码说明：

```
0x1ea8:  MOV   R1,#0        ;vTemp=0，第八种循环
0x1eac:  MOV   R0,#0x64     ;i=100，循环次数设置为 100
0x1eb0:  NOP                ;空操作
0x1eb4:  ADD   R1,R1,R0     ;vTemp=vTemp+i
0x1eb8:  SUB   R2,R0,#0x1   ;i--，减的结果放到到中间变量中
0x1ebc:  MOV   R0,R2        ;i--，将减得的结果取回
0x1ec0:  CMN   R2,#0x1      ;将 i 和 1 比较
0x1ec4:  BNE   0x1eb4       ;不相等跳转到地址 0x1eb4
0x1ec8:  MOV   R1,#0        ;vTemp=0，第九种循环开始
```

方式八汇编代码执行阴影部分循环需要 5 条指令，其中真正跳转 1 次。

方式九 C 代码：

```
vTemp=0;
i=100;
do
{
  vTemp=vTemp+i;
  i--;
}while(i);
```

方式九汇编代码说明：

```
0x1ec8:  MOV   R1,#0        ;vTemp=0，第九种循环
0x1ecc:  MOV   R0,#0x64     ;i=100，循环次数设置为 100
0x1ed0:  NOP                ;空操作
0x1ed4:  ADD   R1,R1,R0     ;vTemp=vTemp+i
0x1ed8:  SUB   R0,R0,#0x1   ;i--
0x1edc:  CMP   R0,#0        ;将 i 和 0 进行比较
0x1ee0:  BNE   0x1ed4       ;比较结果不相等跳转到地址 0x1ed4
```

```
0x1ee4:  MOV   R1,#0          ;vTemp=0，所有循环结束
```

方式九汇编代码执行阴影部分循环需要 4 条指令，其中真正跳转 1 次。

对比九种不同循环方式的汇编代码，可以看出方式一的汇编指令最多，执行一次循环所耗费的时间最长，方式五和方式九的汇编指令最少，执行一次循环所耗费的时间也最短。这些循环方式效率满足 do while() > while() > for()的规律，而在实际中这三种 C 语言循环控制方式使用频率刚好相反，do while() < while() < for()。刚接触 C 语言编程的新人，最喜欢用的就是 for()方式，殊不知这种方式效率最低，所以如果是用 C 语言进行单片机编程，进入到熟练阶段后一定注意循环方式对代码效率的影响。

有兴趣的朋友可以仔细对比这几种方式的汇编代码，从中你可以找出编译器对 C 代码进行编译的规律：基本上按照 C 语言的顺序对应编译，对 i++/ i—这类先用再加的特殊语句是先将 i 的值取出来放到中间变量里面，然后将 i 进行加减处理，循环体中的 i 用中间变量进行替代。

4.3. 循环里的 i++与 i—

循环里面 i++和 i—的使用效果上也会有一些区别，上一节中我们列举了九种 C 语言的循环实现方式，其中方式五和方式九的效率最高，这两种方式一个采用的是 i++，另一个采用 i—，从产生的汇编指令看两者有着相同的效率，好象没有什么区别？别着急，接下来我让你相信确实有区别，而且明白区别是什么原因造成的。

方式五：

```
vTemp=0;
i=0;
do
{
  vTemp=vTemp+i;
  i++;
}while(i<100);
```

循环部分汇编代码

```
0x1e54:  ADD   R1,R1,R0      ;vTemp=vTemp+i
0x1e58:  ADD   R0,R0,#0x1    ;i++
0x1e5c:  CMP   R0,#0x64      ;将未加之前的 i 和 100 进行比较
0x1e60:  BCC   0x1e54        ;小于 100 跳转到地址 0x1e54
```

方式九：

```
vTemp=0;
```

```
i=100;
do
{
  vTemp=vTemp+i;
  i—;
}while(i);
```

循环部分汇编代码

```
0x1ed4:  ADD   R1,R1,R0      ;vTemp=vTemp+i
0x1ed8:  SUB   R0,R0,#0x1    ;i--
0x1edc:  CMP   R0,#0         ;将 i 和 0 进行比较 *这条指令可以不要
0x1ee0:  BNE   0x1ed4        ;比较结果不相等跳转到地址 0x1ed4
```

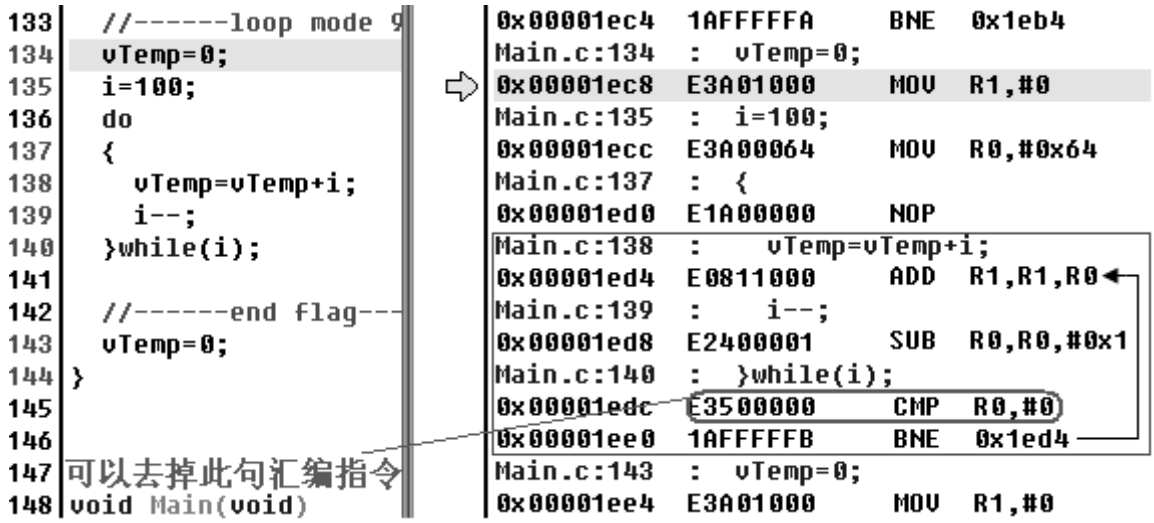

图 4.3.-1 ARM 汇编结果示意图三

对于方式五代码已经没有可以更精简的空间，而方式九我们可以看出循环部分的第三条汇编指令不是必须的，因为前一条代码是执行 i-1 操作，当相减的结果等于 0 的时候会将 CPU 的状态寄存器里面的 Z 状态标志位置上，表示当前相减结果等于 0，而 BNE 指令是通过对 Z 状态标志位来进行判断，当 Z 状态标志位被置位就跳转到后面所带的地址，否则执行下一条指令，这样我们将第三条汇编指令去掉后还可以同样得到正确的程序执行结果。

那编译器能不能去掉第三条汇编指令呢？答案是肯定的，这里我暂时先不展示编译器可以去掉它的结果，因为用我现在的编译器要得到这样的结果需要使用到 C 语言编译的优化功能，下一节我会专门对优化进行讲述，到时再为大家展示。

可以不要第三条指令的原因是利用了单片机指令系统的一个规律，当进行 CPU 进行运算处理后会将状态寄存器里面 Z/C/DC/S 等标志位置位或清零来表示进位、借位、等零状态，条件跳转指令

都是依据这些状态位来决定是否进行跳转。

进行循环控制如果没有特殊的循环控制指令（某些单片机会提供循环控制指令，只要设定循环寄存器就可以让指定的代码循环执行想要的次数）就需要用一个变量进行加减来计算循环次数，通常在C语言里面会用i++/ i—方式来实现。

如果是 i++方式，循环一次都要和需要循环的次数进行比较，然后依据比较结果才知道是否还需要继续循环，但采用 i—则不一样，当达到想要的循环次数时 i 自减的结果为零，这时可以直接利用状态寄存器里等零成立这一条件，选用合适的条件跳转指令来实现跳转，比如BNE/BEQ就是通过判断状态寄存器里等零是否成立来决定是否跳转。

采用i++方式也并不是不能达到与i—同样样的效果。

```
signed long i=-100;
do
{
  i++;
}while(i);   //这里一样可以在i 加到0的时候退出循环
```

但这种负数往上加的方式和日常习惯不一致，会让人们觉得别扭，所以在进行循环控制的时候，建议多用i—的方式，会在符合人们日常习惯的同时得到更高的代码效率，请记住 i—比 i++效率高这一结论。

但从程序可靠性方面来说i—会不如i++。

```
i=100;
while(i)  //当i 不为0 则继续循环
{
  i—;
}
while(i<100)  //当i 小于100 则继续循环
{
  i++;
}
```

如果在循环过程中i因为意外原因导致内容被改变，比如循环了一半时 i 被意外改成了10000，i—方式还要循环10000次才能退出循环，而i++方式即刻退出循环，这样i++和与i--相比i++能早一些从错误中返回。

4.4. 优化的方法与效果

用C 编程很方便的一点是代码容易模块化、循环和跳转之类的操作很好实现，免去了汇编编程那样需要用许多标号来标示程序结构以实现循环或跳转。C 语言采用大括号{}将代码进行分块，除

了可以用大括号来标示循环、选择等块外函数也是用大括号前后包括成一个整体。C 语言将变量分为全局变量和局部变量两类，局部变量的作用域被限定在一个函数之内，全局变量的作用域要大，可以在不同函数中使用。

和汇编相比，C 程序直观易懂，但这个直观易懂建立在 C 语言语法规则基础之上，会受到语法的制约，用 C 编程也会要求程序员尽量让程序结构和层次分明，这样才能更好的体现 C 语言的优点。C 的这个特性存在一个不足，就是前面的要求会让 C 代码效率和空间方面的性能下降，因为用 C 编写程序需要转换成与 MCU 指令对应汇编代码才能被解释执行，转换的过程是按照一定规则进行的，所以转换出来的代码不一定能达到直接用汇编代码编写那么简洁，加上 C 代码对程序结构和层次要求间隔清晰，在编程时容易产生一些冗余代码。

看下面一段 C 代码，对于代码作用一目了然的位置可以不添加注释。

```
行 01：void MCU_CTestFunc(void)
行 02：{
行 03：   unsigned char vAdcX,vAdcY;
行 04：   ...
行 05：   vAdcX=getAdcX();//将 ADC 得到的电压值存入中间变量
行 06：   if(vAdcX>100)
行 07：     printf("X_Voltage>100");
行 08：   else if(vAdcX>50)
行 09：     printf("X_Voltage>50");
行 10：   else
行 11：     printf("X_Voltage<=50");
行 12：   vAdcY=getAdcY();//将 ADC 得到的电压值存入中间变量
行 13：   if(vAdcY>100)
行 14：     printf("Y_Voltage>100");
行 15：   else if(vAdcY>50)
行 16：     printf("Y_Voltage>50");
行 17：   else
行 18：     printf("Y_Voltage<=50");
行 19：   ...
行 20：}
```

行 05 和行 12 先将 ADC 转换得到电压值存入中间变量，以免在行 06、08、13、15 位置再次进行 ADC 转换操作，可以减少代码执行时间；行 05 和行 12 分别用 vAdcX 和 vAdcY 来保存电压值，目的是让程序结构清晰，使 X 和 Y 两部分代码更为直观。如果我们将代码中 vAdcX 和 vAdcY 合并为 vADC，结果并不影响对程序的功能，还能节省出一个 RAM 空间，vAdcX 和 vAdcY 在代码空间上就产生了冗余。

在用汇编进行编程时，程序要求会发生变化，因为汇编语言非常不直观，需要在代码里面尽可能多的加入注释，加上使用汇编编程时程序员会重视代码的空间和效率，从而形成 C 编程中 vAdcX 和 vAdcY 代码空间冗余的几率会降低。

```
行 01：   vADC  EQU 0x40  ;RAM 0x40 用来存放 ADC 转换得到的电压值
行 02：   ...
行 03：getADCX:           ;对 X 电压进行 ADC 转换函数
行 04：   ...
行 05：   LDA ADC_REG     ;将 ADC 转换结果读入累加器
行 06：   STA vADC        ;将累加器中的值存入 vADC
行 07：   RET             ;函数返回，此时 vADC 存放 ADC 转换结果
行 08：   ...
行 09：MCU_ASMTestFunc:
行 10：   ...
行 11：   CALL getADCX    ;调用 X 电压转换函数，执行完 vADC 为 X 电压
行 12：   ...             ;其它代码显示 X 电压区域
行 13：   CALL getADCY    ;调用 Y 电压转换函数，执行完 vADC 为 Y 电压
行 14：   ...             ;其它代码显示 Y 电压区域
行 15：   ...
行 16：   RET
```

汇编代码与 C 代码相比可以看出如果汇编不加上注释会很难看懂代码的意思，因为这些注释的存在可以不用象 C 代码那样用 vAdcX 和 vAdcY 两个变量名，只用 vADC 就可以依靠注释让代码段清功能明晰。

我们可以将 C 代码中的 vAdcX 和 vAdcY 合并为 vADC，但这样做需要 C 程序员改变约定俗成的程序风格，实际上我们不改变 C 代码也同样能实现 vAdcX 和 vAdcY 合并，编译器通过对程序的扫描可以知道在函数 MCU_CTestFunc()内 vAdcX 和 vAdcY 只用来临时存放电压值，而且两个电压值不会同时出现，这样就可以在编译的时候给 vAdcX 和 vAdcY 分配同一个 RAM，程序前半段给 vAdcX 用，后半段给 vAdcY 用，这种做法就是编译器的优化。

编译器对 C 代码编译最简单的方法就是将每一条 C 代码都转换成相应汇编指令，这种处理方法得到的汇编指令从结构层次上和原始 C 代码基本上一致，只要 RAM 等资源够用，就能比较容易做到编译出的汇编代码功能和 C 代码一样。优化则是编译器依据自己的经验，将 C 代码编译转换成汇编指令时在保证代码执行结果正确的前提下做出一些特殊调整来改良代码空间和效率。

……（详见完整版）

4.5. 全局变量的风险

不少单片机程序员喜欢用全局变量传递控制信息，比如中断标志什么的，这样做的优点是简单方便，程序员可以灵活的设定自己想传递的控制信息内容。用全局变量传递控制信息不能随心所欲，需要遵循一些规则，最常见的是不要让多方同时进行写操作的可能性存在，比如一个全局变量 x，在主程序和中断程序中都会对其进行写操作，这是风险系数非常高的操作，因为这样有可能刚好在主程序在更改 x 内容的时候中断产生，中断程序又对 x 写入另外的内容，从而导致程序判断出错，这一点只要稍有经验的工程师都会知道，但实际开发中工程师有可能知道这种风险但不能很好的避免其产生，接下来我会通过两个例子来介绍两种常见的风险疏忽。

变量宽度与单片机位宽不一致时中断使用全局变量导致出错。假定现在有一款 8bits 的单片机，我们用一个周期为 1ms 的 timer 中断做为系统时钟，timer 每中断一次会将一个 32bits 的变量加一，程序通过这个变量知道单片机上电后已经运行的时间。例子采用 C 语言编程，单片机指令系统为 6502 汇编指令。

C 代码

```
unsigned long last_time;
unsigned long ms_counter;
unsigned long get_msCounter(void)
{
  return ms_counter;
}
void main(void)
{
  ...
  ms_counter=0;
  init_timer0_irq();
  enable_irq();
  while(1)
  {
    last_time=get_msCounter();
    ...
    while(get_msCounter()<(last_time+100));
  }
}
void timer0_irq(void)
```

```
{
  CLR_TIMER0_INT_FLAG;
  ms_counter++;
}
```

C 代码汇编指令执行示意

```
unsigned long last_time;
//汇编指令用来存储变量 last_time 的 4 字节 RAM 变量
//DB  LAST_TIME_BYTE1
//DB  LAST_TIME_BYTE2
//DB  LAST_TIME_BYTE3
//DB  LAST_TIME_BYTE4
unsigned long ms_counter;
//汇编指令用来存储变量 ms_counter 的 4 字节 RAM 变量
//DB  MS_COUNTER_BYTE1
//DB  MS_COUNTER_BYTE2
//DB  MS_COUNTER_BYTE3
//DB  MS_COUNTER_BYTE4
//汇编指令用来存储函数返回参数的 4 字节 RAM 变量
//DB  FUNC_RETURN_BYTE1
//DB  FUNC_RETURN_BYTE2
//DB  FUNC_RETURN_BYTE3
//DB  FUNC_RETURN_BYTE4
//汇编指令用于计算处理的 4 字节 RAM 变量
//DB  CALC_TEMP_BYTE1
//DB  CALC_TEMP_BYTE2
//DB  CALC_TEMP_BYTE3
//DB  CALC_TEMP_BYTE4
unsigned long get_msCounter(void)
{
  return ms_counter;
//将 ms_counter 的内容存放到函数返回参数的 4 字节 RAM 变量中
  //LDA  MS_COUNTER_BYTE1          ;A=MS_COUNTER_BYTE1
  //STA  FUNC_RETURN_BYTE1         ;FUNC_RETURN_BYTE1=A
  //LDA  MS_COUNTER_BYTE2          ;A=MS_COUNTER_BYTE2
```

```
    //STA  FUNC_RETURN_BYTE2        ;FUNC_RETURN_BYTE2=A----------③
    //LDA  MS_COUNTER_BYTE3         ;A=MS_COUNTER_BYTE3
    //STA  FUNC_RETURN_BYTE3        ;FUNC_RETURN_BYTE3=A
    //LDA  MS_COUNTER_BYTE4         ;A=MS_COUNTER_BYTE4
    //STA  FUNC_RETURN_BYTE4        ;FUNC_RETURN_BYTE4=A
    //RTS                           ;函数返回
}
void main(void)
{
  ...
  ms_counter=0;
  //将 ms_counter 清零
    //LDA  #0                       ;A=0
    //STA  MS_COUNTER_BYTE1         ;MS_COUNTER_BYTE1=A
    //LDA  #0                       ;A=0
    //STA  MS_COUNTER_BYTE2         ;MS_COUNTER_BYTE2=A
    //LDA  #0                       ;A=0
    //STA  MS_COUNTER_BYTE3         ;MS_COUNTER_BYTE3=A
    //LDA  #0                       ;A=0
    //STA  MS_COUNTER_BYTE4         ;MS_COUNTER_BYTE4=A
    init_timer0_irq();
    //对 timer0 中断进行设置，不示意汇编指令
    enable_irq();
    //开中断，不示意汇编指令
    while(1)
    //ADDRESS_1 为 while(1)死循环地址
    //ADDRESS_1:                    ;地址 1
    {
      last_time=get_msCounter();----------①
      //得到当前循环的起始时间
      //JSR  get_msCounter           ;调用函数 get_msCounter
      //LDA  FUNC_RETURN_BYTE1       ;A=FUNC_RETURN_BYTE1
      //STA  LAST_TIME_BYTE1         ;LAST_TIME_BYTE1=A
      //LDA  FUNC_RETURN_BYTE2       ;A=FUNC_RETURN_BYTE2
      //STA  LAST_TIME_BYTE2         ;LAST_TIME_BYTE2=A
```

```
//LDA  FUNC_RETURN_BYTE3      ;A=FUNC_RETURN_BYTE3
//STA  LAST_TIME_BYTE3        ;LAST_TIME_BYTE3=A
//LDA  FUNC_RETURN_BYTE4      ;A=FUNC_RETURN_BYTE4
//STA  LAST_TIME_BYTE4        ;LAST_TIME_BYTE4=A
...
while(get_msCounter()<(last_time+100));----------------------②
//如果循环时间达到 100ms 则开始下一次循环，未考虑溢出问题
//ADDRESS_2:                  ;地址 2
//JSR  get_msCounter          ;调用函数 get_msCounter
//CLC                         ;清 C 进位标志位
//LDA  LAST_TIME_BYTE1        ;A=LAST_TIME_BYTE1
//ADC  #100                   ;A=A+100+C，如果有进位则 C 置 1
//STA  LAST_TIME_BYTE1        ;LAST_TIME_BYTE1=A，不影响 C 状态
//LDA  LAST_TIME_BYTE2        ;LAST_TIME_BYTE2，不影响 C 状态
//ADC  #0                     ;A=A+0+C，如果有进位则 C 置 1
//STA  LAST_TIME_BYTE2        ;LAST_TIME_BYTE2=A，不影响 C 状态
//LDA  LAST_TIME_BYTE3        ;LAST_TIME_BYTE3，不影响 C 状态
//ADC  #0                     ;A=A+0+C，如果有进位则 C 置 1
//STA  LAST_TIME_BYTE3        ;LAST_TIME_BYTE3=A，不影响 C 状态
//LDA  LAST_TIME_BYTE4        ;LAST_TIME_BYTE4，不影响 C 状态
//ADC  #0                     ;A=A+0+C，如果有进位则 C 置 1
//STA  LAST_TIME_BYTE4        ;LAST_TIME_BYTE4=A，忽略溢出情况
//LDA  FUNC_RETURN_BYTE4      ;A=FUNC_RETURN_BYTE4
//CMP  LAST_TIME_BYTE4        ;比较 A 和 LAST_TIME_BYTE4
//BCS  ADDRESS_3              ;如果 A 小于 LAST_TIME_BYTE4 跳转到地址 3
//LDA  FUNC_RETURN_BYTE3      ;A=FUNC_RETURN_BYTE3
//CMP  LAST_TIME_BYTE3        ;比较 A 和 LAST_TIME_BYTE3
//BCS  ADDRESS_3              ;如果 A 小于 LAST_TIME_BYTE3 跳转到地址 3
//LDA  FUNC_RETURN_BYTE2      ;A=FUNC_RETURN_BYTE2
//CMP  LAST_TIME_BYTE2        ;比较 A 和 LAST_TIME_BYTE2
//BCS  ADDRESS_3              ;如果 A 小于 LAST_TIME_BYTE2 跳转到地址 3
//LDA  FUNC_RETURN_BYTE1      ;A=FUNC_RETURN_BYTE1
//CMP  LAST_TIME_BYTE1        ;比较 A 和 LAST_TIME_BYTE1
//BCS  ADDRESS_3              ;如果 A 小于 LAST_TIME_BYTE1 跳转到地址 3
//JMP  ADDRESS_2              ;跳转到地址 2
```

```
    //ADDRESS_3:                ;地址 3
    //JMP  ADDRESS_1            ;跳转到地址 1
  }
//RTS                           ;函数返回
}
void timer0_irq(void)
{
  CLR_TIMER0_INT_FLAG;
  //清中断标志位操作，不示意汇编指令
  ms_counter++;
  //将 ms_counter 加一
  //CLC                         ;清 C 进位标志位
  //LDA  MS_COUNTER_BYTE1       ;A=MS_COUNTER_BYTE1
  //ADC  #1                     ;A=A+1+C，如果有进位则 C 置 1
  //STA  MS_COUNTER_BYTE1       ;MS_COUNTER_BYTE1=A，不影响 C 状态
  //LDA  MS_COUNTER_BYTE2       ;A=MS_COUNTER_BYTE2，不影响 C 状态
  //ADC  #0                     ;A=A+0+C，如果有进位则 C 置 1
  //STA  MS_COUNTER_BYTE2       ;MS_COUNTER_BYTE2=A，不影响 C 状态
  //LDA  MS_COUNTER_BYTE3       ;A=MS_COUNTER_BYTE3，不影响 C 状态
  //ADC  #0                     ;A=A+0+C，如果有进位则 C 置 1
  //STA  MS_COUNTER_BYTE3       ;MS_COUNTER_BYTE3=A，不影响 C 状态
  //LDA  MS_COUNTER_BYTE4       ;A=MS_COUNTER_BYTE4，不影响 C 状态
  //ADC  #0                     ;A=A+0+C，如果有进位则 C 置 1
  //STA  MS_COUNTER_BYTE4       ;MS_COUNTER_BYTE4=A，忽略溢出情况
  //RTI                         ;中断函数返回
}
```

●注：汇编代码和 C 代码只是示意用，不能直接理解成实际情况

来看看前面代码什么情况下会出错，在位置①和位置②，程序都有调用函数 get_msCounter()，如果在位置①调用函数时 ms_counter 的值为 0x0000FFFF，当函数 get_msCounter()运行到汇编指令的位置③的时候 timer 中断产生，ms_counter 的值加一变为 0x00010000，正确的结果应该是调用函数 get_msCounter()后 last_time 的值为 0x0000FFFF 或 0x00010000，但现在结果变成了另外的值 0x0001FFFF。

程序运行到位置①时：

MS_COUNTER_BYTE1=0xFF

MS_COUNTER_BYTE2=0xFF

MS_COUNTER_BYTE3=0x00

MS_COUNTER_BYTE4=0x00

程序运行到位置③时：

LAST_TIME_BYTE1=MS_COUNTER_BYTE1=0xFF

LAST_TIME_BYTE2=MS_COUNTER_BYTE2=0xFF

MS_COUNTER_BYTE3=0x00

MS_COUNTER_BYTE4=0x00

LAST_TIME_BYTE3 和 LAST_TIME_BYTE4 还未取得新的值

此时 timer 中断产生，转去执行中断程序：

MS_COUNTER_BYTE1=0x00

MS_COUNTER_BYTE2=0x00

MS_COUNTER_BYTE3=0x01

MS_COUNTER_BYTE4=0x00

中断返回后继续取 LAST_TIME_BYTE3 和 LAST_TIME_BYTE4 的新值：

LAST_TIME_BYTE1 保持 0xFF 不变

LAST_TIME_BYTE2 保持 0xFF 不变

LAST_TIME_BYTE3=MS_COUNTER_BYTE3=0x01

LAST_TIME_BYTE4=MS_COUNTER_BYTE4=0x00

于是 last_time 得到错误的值 0x0001FFFF。

当程序运行到位置②进行循环判断时，循环控制出错，原本 100ms 的循环周期变成需要等待 65 秒才能继续下一次循环。

要避免此种风险的发生是变量位宽改为与单片机位宽一致，或者将中断修改的变量读回来后进行一次重复比较，满足前后读回的值一样的才当作有效值。

修改后的代码

```
unsigned long last_time,current_time;
unsigned long ms_counter;
unsigned long get_msCounter(void)
{
  return ms_counter;
}
void main(void)
{
  ...
  ms_counter=0;
  init_timer0_irq();
```

```
    enable_irq();
    while(1)
    {
      do                    //连续两次读得的值一样才有效
      {
        last_time=get_msCounter();
      }while(last_time!=get_msCounter());
      ...
      do
      {
        do                  //连续两次读得的值一样才有效
        {
          current_time=get_msCounter();
        }while(current_time!=get_msCounter());
      }while(current_time<(last_time+100));
    }
}
```

再来看另外一种情况，单片机依然为 8bits，中断和主程序会修改同一个变量，于是程序通过一个 8bits 变量保护标志位来对主程序的操作进行保护，当中断或主程序需要修改变量的时候，先查看变量保护标志位，如果不为 0 暂不进行修改，如果为 0 则将变量保护标志位设为非 0 值，然后进行修改变量操作，修改完毕将变量保护标志位清 0 以允许主程序或中断继续修改变量。假定 C 语言编程，单片机指令系统为 6502 汇编指令。

……（详见完整版）

4.6. 变量类型与代码效率

毋庸置疑，单片机处理和其宽度相等的数据效率最高，比如 16bits 的单片机去处理 32bits 位宽数据需要 32bits 数据分成 16bits 高低两部分后才能处理，效率自然要低不少。

采用 ARM 内核的单片机会有点特殊，ARM 内核本身是 32bits，但其支持 STRB/LDRB 这样的特殊指令，该指令可以直接读写 8bits 的数据，所以 ARM 处理 8bits 数据的时候效率不一定要比 32bits 低，但同样去传递一块数据，显然还是用 32bits 的要快，只是需要在数据块边缘用 8bits 数据进行对齐处理。

既然处理数据的效率和单片机位宽有关，那么我们在用C语言对单片机编程时就需要考虑单片机位宽，只有这样才能保证程序具有良好的效率，假设现在有两个单片机，一个为8bits，另外一个为32bits，需要我们用这两款单片机编写一段程序，将一段数据复制到其它位置。

程序一

```
void Copy_TestFunc(char * desBuf,char * srcBuf,unsigned long size)
{
  while(size)
  {
    *desBuf=*srcBuf;          //复制1字节
    size--;                   //计数器减一
    desBuf++;                 //目的地址加一
    srcBuf++;                 //源地址加一
  }
}
```

程序二

```
void Copy_TestFunc(char * desBuf,char * srcBuf,unsigned long size)
{
  long *p1,*p2;               //用于32bits传送
  short *p3,*p4;              //用于16bits传送
  char *p5,*p6;               //用于8bits传送
  if((((long)desBuf&0x3)==0)&&(((long)srcBuf&0x3)==0))
  {
    //32bits mode             //目的地址和源地址都满足4字节对齐
    p1=(long *)desBuf;        //得到目的地址long指针
    p2=(long *)srcBuf;        //得到源地址long指针
    while(size>=4)            //还有不少于4字节的数据需要传送就循环
    {
      *p1=*p2;                //源地址传送4字节到目的地址
      size-=4;                //计数器减4
      p1++;                   //目的地址long指针自加，实际上是加了4
      p2++;                   //源地址long指针自加，实际上是加了4
    }
    p5=(char *)p1;            //目的地址改用char指针
    p6=(char *)p2;            //源地址改用char指针
    while(size)               //每次循环复制1字节到结束
```

```
    {
      *p5=*p6;
      size--;
      p5++;
      p6++;
    }
  }
  else if((((long)desBuf&0x1)==0)&&(((long)srcBuf&0x1)==0))
  {
    //16bits mode              //目的地址和源地址都满足 2 字节对齐
    p3=(short *)desBuf;        //得到目的地址 short 指针
    p4=(short *)srcBuf;        //得到源地址 short 指针
    while(size>=2)             //还有不少于 2 字节的数据需要传送就循环
    {
      *p3=*p4;                 //源地址传送 2 字节到目的地址
      size-=2;                 //计数器减 2
      p1++;                    //目的地址 long 指针自加，实际上是加了 4
      p2++;                    //源地址 long 指针自加，实际上是加了 4
    }
    if(size)
    {
      (char *)p3=(char *)p4;//此时只剩余 1 字节需要复制
    }
  }
  else
  {
    //8bits mode               //目的地址或源地址不满足 2 字节对齐
    while(size)                //每次循环复制 1 字节到结束
    {
      *desBuf=*srcBuf;
      size--;
      desBuf++;
      srcBuf++;
    }
  }
```

```
}
```

程序二与程序一相比程序结构要复杂一些，会根据源地址和目的地址的状态自动选择传送方式。对于8bits的单片机，每次只能传一个8bits的数据，所以无论程序一或者程序二单片机都会一个字节一个字节的传送，显然程序一的速度要快。对于32bits的单片机结果会大不一样，如果只是传送几个字节，程序二和程序一比并没有什么优势，甚至会更慢，但当传送的数据个数多而且起始地址满足4字节对齐时，程序二一条指令就可以传送4个字节，传送的速度几乎可以达到程序一的4倍。

所以在用C语言对单片机编程时，同样的C代码用到不同的单片机上效率可能大相径庭，我们需要结合单片机的硬件特性进行编程才能得到高效代码。标准C提供了不少库函数，这些库函数功能完善而且高效，使用它可以让编程人员方便不少，但要留意这些库函数有可能已经基于32bits平台做出了类似前面程序二的特殊处理，在非32bits的单片机上很有可能不能体现出高效的特性。

4.7. 慎用 int

现在用C编写的程序是不计其数，提供软硬件服务的厂商、具有雷锋精神的程序爱好者写出了各种各样的C程序代码，以便其他人在他们所提供C代码的基础上更快的完成产品开发，纵观这些C代码，你会发现里面大量使用int进行数据定义，实际上这不是一个好的做法。

早期的C语言有一个定义上的缺陷，对int的描述不够严谨，只是定义其为整形变量，单纯从语法上说这个定义不存在什么问题，可是C语言需要在相应的硬件平台上运行才有实际意义，而C语言没有进一步明确不同硬件平台下int位宽由硬件平台决定这一点，从而导致后面对int出现多种不同的理解。

不同的硬件平台所支持的数据位宽可能会不一样，在计算机技术发展过程中，主流的CPU经历了8bits、16bits、32bits、64bits这样的转变，不同的人在理解int时往往会基于当时流行的CPU加入自己的个人理解，我记得很清楚国内一很知名的C语言教材早期版本里面对int解释就是16bits，正是这样教材的疏漏使得更多人将int理解成16bits，后来32bits处理器的流行，大家又错误地默认int为32bits。

对int的这种理解不对，实际上int并没有一个具体的位宽限制，由所用硬件平台（MCU）和编译器共同决定位宽为多少，通常情况下编译器会将int的位宽定为与所用MCU的位宽一致，这样MCU对定义为int的数据进行处理时因为刚好是整数不用取舍或拼凑而最为方便。

不同的MCU和所提供的编译器对int的解释各不相同，8bits、16bits、32bits、64bits都有可能，如果在程序中使用int来定义数据类型，就会将一些不确定因素引入程序中，使得程序的风险系数增高。

比如编程人员忽视了编译器对int定义的位宽，习惯性的将int理解成现在大家所默认的32bits，如果现在我们用一个MCU在C语言下开发了一款产品，所用的MCU和编译器为16bits宽度，产品需要显示开机后的时间，显示精度需要达到秒，程序员在程序里变量SecondCounter来纪

录开机时间，程序员将 unsigned int 错误的理解成了 32bits，认为用 unsigned int 定义的 SecondCounter 可以保证所记录的开机时间超过 100 年，完全满足任何用户的需求。

然而编译器实际是按 16bits 来处理，只能记录到 65535/3600≈18 小时。这个产品大多数应用都只是在正常上班时段，早上上班时开机，晚上下班时关机，内部测试和用户试用都是按早开机晚关机器模式进行，运行结果一切正常，于是开始量产。然而另外客户使用中发现了问题，该客户不是早开机晚关机模式，而是三班倒，需要一周甚至一个月才关机一次，产品根本无法满足客户的需求，结果是只能将已经生产的机器当作问题机处理。

这种情况主要还是因为开发人员的疏忽导致，另外一种情况就和人为疏忽无关，用 C 语言编程一大优点就是移植容易，可以很快从一个硬件平台转换到另外一个硬件平台上。产品最初阶段是采用 32bits 的 MCU，编译器解释 int 为 32bits，程序员理解没有错误也是 32bits，产品程序成功完成，运行一切正常。32bits 的 MCU 价格比较贵，成本上和其它同类产品不具竞争力，现发现有一款 16bits 的 MCU 可以满足性能需求，而且价格要低不少，于是希望改用这款 16bits 的 MCU。

先前已经写好的程序由于使用了大量的 int，直接移植到 16bits 的 MCU 编译器就将原来解释为 32bits 的 int 改解释为 16bits，显然存在数据溢出的风险，就需要我们对原程序中所有 int 都进行修订。如果我们将原来的 int 全部替换为 long 就可以避免数据溢出的风险，但这样做需要我们将程序里面所有的 int 定义逐个找出来进行替换，有一个遗漏都不算成功移植，虽然我们可以用编辑工具提供的替换功能，但替换功能会将所有“int”字符都替换掉，会将本不需要更改的地方改错。如果我们一开始就将 int 用 long 来定义，移植过程根本不需要担心数据类型解释不对的问题，使得移植程序的工作量会大为减少。

正是这些因素的存在，我的建议是尽量少在程序中使用 int 进行数据定义，C 语言提供更为精准的数据类别 char、short、long 、long long 分别严格对应 8bits、16bits、32bits、64bits，任何编译器编译出来的结果都一定满足这种对应关系，这样定义出来的数据宽度和硬件无关，在任何硬件平台上编译出来的宽度都是一样的，所以用它们定义数据类型会更好。

请记住：用 C 编写程序的时候，一定要谨慎使用 int，最好是不用。

4.8. 危险的指针

刚接触 C 语言的人可能会不大明白指针到底是什么，从我的个人理解，指针就是存储器的地址，但不能完全的等同地址，指针除了直接提供地址信息外，另外指针类型还间接告诉大家所指向的地址包含多少内容。

用一个例子来帮助大家理解指针，假设现在你是一个统领部队的指挥长，将手下的士兵成一列纵队站好，第一个士兵编号为 0，第二个编号为 1，依次类推，这里编号你就可以对应为存储器的地址。为了便于管理，你将士兵用班排连的组织来进行管理，一个班有十个士兵，一个排有三个班，一个连有三个排和另外一个独立的炊事班。

一连一排一班：士兵编号 0~9

一连一排二班：士兵编号 10~19

一连一排三班：士兵编号 20~29

一连二排一班：士兵编号 30~39

一连二排二班：士兵编号 40~49

一连二排三班：士兵编号 50~59

一连三排一班：士兵编号 60~69

一连三排二班：士兵编号 70~79

一连三排三班：士兵编号 80~89

一连炊事班：士兵编号 90~99

一连一排一班：士兵编号 100~109

二连一排二班：士兵编号 110~119

二连一排三班：士兵编号 120~129

二连二排一班：士兵编号 130~139

……

这里的连排班以及士兵编号就可以理解成指针，每个士兵是最基本的组成单位，其对应的编号对应为存储器的地址，一连二排二班表示编号 40~49 的士兵，他们现在位于队伍中的第 40+1 个人的位置，一共十个人。现在需要一个班的士兵去执行任务，你就可以通过班号选择执行任务的班，下次需要一个排你又可以通过排号来选则执行任务的排，这种方式就类似指针的操作。

前面例子也许还不能很好的解释指针，接下来直接回到 C 语言中对指针进行讲解，先从最基本的 char/short/long 类型说起。(little endian 模式)

```
char  *p_char;    //定义一个类型为 char 的指针
short  *p_short;  //定义一个类型为 short 的指针
long  *p_long;    //定义一个类型为 long 的指针
```

在 C 语言里用*号来表示指针，这一点是当时定义 C 语言语法的人决定的，没有为什么可言，当初定义语法的人选用了*号而已，如果当时选用的是其它符号现在也就是其它符号。

```
p_char=(char *)0x1000;      //将指针指向地址 0x1000
p_short =(short *)0x1000;   //将指针指向地址 0x1000
p_long =(long *)0x1000;     //将指针指向地址 0x1000
*p_char=0x12;               //地址 0x1000 内容为 0x12
*p_short=0x1234;            //地址 0x1000 内容为 0x34，地址 0x1001 内容为 0x12
*p_long=0x12345678;         //地址 0x1000 内容为 0x78，地址 0x1001 内容为 0x56，地址 0x1003 内容为 0x34，地址 0x1004 内容为 0x12
```

可以看出指向同样地址的不同类型的指针所代表的内容并不相同，char 的指针为一个字节的内容，而 short 和 long 却分别为两个字节和四个字节，显然指针除了带有地址信息外还有所包含内容大小的信息，这就是指针与地址所不同的地方。

注意将指针指向一个地址的操作并不是使用 p_char=0x1000 这样的语句直接等，而是使用了(char *)进行强制转换，如果不加(char *)的话编译会报告错误。这是C语言使用指针时的一个限制，所有对于指针的操作必须是同类型的指针才可以进行，对于 0x1000 这样一个数字，虽然我们可以当它是一个地址，但用做指针的时候需要表明指针类型才能使用，否则程序无法知道*0x1000 这样的操作表示的到底是一个字节的 char 类型还是两个字节的 short 类型，C语言有这样的限制后就可以让指针和地址区分开。

用C 语言编程变量的分配大都由编译器决定，也就是说程序员可以不用理会所用变量究竟放在什么位置，而用C语言对单片机编程许多时候需要直接对某个地址进行读写操作，引入指针就会使这类操作变的更简单直接。对于 char 类型的指针，每个指针只对应一个字节的内容，而 short 类型则是两个字节，C语言为了提升效率对于这类指针做出了起始地址对齐的要求，比如 short 指针地址需要能被二整除，而 long 指针地址则需要能被四整除。

这里不想对什么是指针这样的概念性问题阐述过多，总之一点是指针让程序控制存储器空间变的非常简捷，但正是这个简捷打破了C语言的封闭性，让程序员可以随心所欲的去读写存储器空间，所以在便捷的同时也给程序带来了一定的风险性。

现在需要用程序来完成这样的功能：从地址 0x1000 开始将 0、1、2、…、255 依次写入头 256 字节，为了检验程序是否写入正确还需要将写入内容回读回来进行校验。

正确的代码：

```
long i;
char *p1,*p2;           //这里指针类型是 char
p1=(long *)0x1000;      //将指针 p1 地址指向 0x1000
p2=(long *)0x1000;      //将指针 p2 地址指向 0x1000
for(i=0;i<256;i++)
{
  *p1=i;                //将指针内容依次写为 0、1、2、…、255
  p1++;                 //指针加一，指针所指向地址随之加一
}
for(i=0;i<256;i++)
{
  if(*p2!=i)
  {
    while(1);           //如果校验出错则进入这个死循环
  }
  p2++;                 //指针加一，指针所指向地址随之加一
}
```

错误的代码：

```
long i,*p1,*p2;          //这里指针类型是 long
p1=(long *)0x1000;       //将指针 p1 地址指向 0x1000
p2=(long *)0x1000;       //将指针 p2 地址指向 0x1000
for(i=0;i<256;i++)
{
  *p1=i;                 //将指针内容依次写为 0、1、2、…、255
  p1++;                  //指针加一，实际上指针所指向地址加四
}
for(i=0;i<256;i++)
{
  if(*p2!=i)
  {
    while(1);            //如果校验出错则进入这个死循环
  }
  p2++;                  //指针加一，实际上指针所指向地址加四
}
```

错误代码的原因是未能正确使用指针，对于 char 类型的指针，执行自加或者加一操作后指针地址同样是加一，而 short 和 long 则不一样，分别加二和加四。

```
char *p=0x1000;  //char 类型指针 p 指向地址 0x1000
p++;             //执行完该指令后指针 p 指向地址 0x1001，注意这里指针地址加一
long *p=0x1000;  // char 类型指针 p 指向地址 0x1000
p++;             //执行完该指令后指针 p 指向地址 0x1004，注意这里指针地址加四
```

正确写入的结果：

```
地址     内容
0x1000   0x00
0x1001   0x01
0x1002   0x02
…
0x10FF   0xFF
```

错误写入的结果：

```
地址     内容
0x1000   0x00
0x1001   0x00     多写入的 0x00
0x1002   0x00     多写入的 0x00
```

0x1003 0x00 多写入的0x00

0x1004 0x01

0x1005 0x00 多写入的0x00

0x1006 0x00 多写入的0x00

0x1007 0x00 多写入的0x00

…

0x13FC 0xFF

0x13FD 0x00 多写入的0x00

0x13FE 0x00 多写入的0x00

0x13FF 0x00 多写入的0x00

可以看出错误是每写一个数据会另外在后面多写入三个0x00，同时地址往后加四，虽然代码也有进行回读校验的操作，但回读操作完全是写入操作的逆向过程，所以这个校验不会发现该错误，如果不发生空间溢出错误，这个错误就会被隐藏起来而不被发现，从外表看程序运行一切正常，但内部的实际运行结果并不是程序员所预期的。

不要认为这种错误不严重，如果位于0x1100~0x13FF的空间有其它作用，前面的错误就会导致这部分空间的内容被错误修改，从而导致程序运行出错，严重的可以让程序崩溃。

使用指针最大的风险是容易产生空间溢出或者地址错误的情况，用C语言对数组或者变量进行读写的时候，虽然程序员不清楚这些数组或者变量在存储器中的具体位置，但读写操作是用数组或变量名来进行读写的，所以很容易做到所需要进行读写的对象不出错，如果在编写程序时因为书写等原因让名称出错，编译器也会报告名称错误，这样读写到错误位置的几率会相当小，除非在代码中定义了名称非常相似的数组或变量名。而用指针来进行读写则不一样，需要程序员自我对读写的位置进行保护确认，只要有一点疏漏就可能导致错误的产生。

来看一个例子，这个例子是因为其它用途方便而定义了四个 64 字节的数组，现在想将 0~255 分别写到这四个数组里面去，数组 Temp_BufA[]写入 0~63，数组 Temp_BufB[]写入 64~127，数组 Temp_BufC[]写入 128~191，数组 Temp_BufD[]写入 192~255。

```
char Temp_Byte;
char __align(256) Temp_BufE[256];  //__align(256)表示起始地址需要256字节对齐
char Temp_BufA[64];
char Temp_BufB[64];
char Temp_BufC[64];
char Temp_BufD[64];
char *p;
long i;
p=Temp_BufA;                       //指针地址指向数组Temp_BufA[64]的首字节地址
```

```
for(i=0;i<256;i++)
{
  *p++=i;                              //依次写为0、1、2、…、255
}
```

可能不少人会认为这段程序没有问题，他们会说数组 Temp_BufA[]到数组 Temp_BufD[]是连续定义的，在程序里面所占用的存储空间自然也是连续的，前面的程序只用一个指针循环就可以将这四个数组写入所需的值，所以程序不但没有问题，而且还是一段很精简的代码，值得大家学习。实际情况并不如此，我们并不能保证数组 Temp_BufA[]到数组 Temp_BufD[]占用连续的存储空间，有可能这四个数组是被间隔开。

假定存储空间的分配是从地址 0x1000 开始，编译器看到 Temp_Byte，于是将地址 0x1000 分配给 Temp_Byte 用，可接下来的是数组 Temp_BufE[]，需要 256 字节对齐，显然不能再将 0x1001 分配为数组 Temp_BufE[]的起始地址，而是分配 0x1100 才满足要求。这时编译器处理到数组 Temp_BufA[]，因为地址 0x1001~0x10FF 区间还没有被分配出去，所以将 0x1001~0x1040 区间分配给数组 Temp_BufA[]，而不是从地址 0x1200 开始分配。

与数组 Temp_BufA[]一样数组 Temp_BufB[]和数组 Temp_BufC[]会被分配到地址 0x1041~0x1080 和 0x1081~0x10C0 这两段区域。但处理到数组 Temp_BufD[]时遇到新问题，显然数组 Temp_BufD[]需要连续的 64 字节，而现在从地址 0x10C1 开始到 0x10FF 只有 63 个字节的剩余空间，所以数组 Temp_BufD[]所分配的空间只好分配为地址 0x1200~0x123F 这一段，这样四个数组的空间并不连续，如果运行前面的代码自然就会得到错误的结果。

按前面假定编译器可以得出的存储器分配空间如下（实际情况不一定和此相同）：

地址	内容
0x1000	变量 Temp_Byte
0x1001~0x1040	数组 Temp_BufA[64]
0x1041~0x1080	数组 Temp_BufB[64]
0x1081~0x10C0	数组 Temp_BufC[64]
0x10C1~0x10C3	空余区间
0x10C4~0x10C7	指针*p
0x10C8~0x10CB	变量 i
0x10CC~0x10FF	空余区间
0x1100~0x11FF	数组 Temp_BufE[256]
0x1200~0x123F	数组 Temp_BufD[64]

指针有的时候会有地址对齐的要求，比如 short 和 long 类型的指针分别要求两字节和四字节对齐，有的编译器并不能对这些细节进行可靠的检查，稍有不慎就可能用错指针。

```
long *p;              //申明类型为 long 的指针，按要求需四字节对齐
```

p=(long *)0x1001; //这里将指针地址指向 0x1001

这样的代码有的编译器不会报告错误，而实际上类型为 long 的指针地址是必须四字节对齐的，所以代码带有错误，执行这样的代码会导致错误发生，而程序员还以为自己已经成功将地址 0x1001 交给指针 p。

指针空间溢出的错误在实际中最容易发生，常常会遇到这样的情况，变量没有被任何代码直接修改，但程序运行的结果发现变量内容被修改成程序员所不期望的内容，这种错误大都是使用指针空间溢出导致的。对于这种错误由于不能通过查看代码是否更改了变量，加上错误的产生是偶然的，需要在某些特定条件下才会出现，所以如果仿真调试工具功能不是很强大的话很难调试跟踪这种错误。

虽然这种错误经常会遇到，但形成的原因往往错综复杂，不大容易用一个简洁的例子就能说明清楚，这里给出一个例子，错误产生的根源也许并不能完全归于指针，但确实是和使用指针有关。我们通过 UART 收发数据，数据依照下面格式进行传送，最大长度不超过 1024 字节。（例子代码是针对演示错误而特意写成下面形式）

Byte1　　同步字 0xFF，用来标示数据包开始

Byte2~3　　两个字节用来表示数据包长度

Byte4~N　　数据内容，规定数据包中不可以包含 0xFF 以免与同步字冲突

ByteN+1　　数据包内容校验码低位字节

ByteN+2　　数据包内容校验码高位字节

接收端代码：

```
unsigned short receive_count;          //数据接收计数器
unsigned short check_sum;              //用来计算数据包校验码
unsigned char receive_buf[1024],*p;    //数据存放缓冲区和接收存放指针
char receive_flag=0;                   //数据接收状态标志
UART_RxInt()                           //每收到一个字节都会触发该中断函数
{
  unsigned char temp;
  temp=*UART_RX_DATA;                  //将接收到的数据读到 temp 中
  …//其它代码
  if((receive_flag==0)&&(temp==0xFF))  //如果接收状态为 0 且当前收到同步字
  {
    receive_falg=1;                    //接收状态转为 1
    p=receive_buf;                     //接收指针指向接收缓冲区首地址
  }
  else if(receive_flag==1)             //如果接收状态为 1
```

```
    {
      receive_flag=2;                    //接收状态转为 2
      receive_count=temp;                //保存数据包长度低位字节
    }
    else if(receive_flag==2)             //如果接收状态为 2
    {
      receive_flag=3;                    //接收状态转为 3
      receive_count|=(temp<<8);          //保存数据包长度高位字节
    }
    else if(receive_flag==3)             //如果接收状态为 3
    {
      if(receive_count>0)                //保存数据直到数据接收计数器为零
      {
        *p++=temp;                       //保存数据到缓冲区，指针地址加一
        receive_count--;                 //数据接收计数器减一
      }
      else
      {
        receive_flag=4;                  //接收状态转为 4
      }
    }
    else if(receive_flag==4)             //如果接收状态为 4
    {
      receive_flag=5;                    //接收状态转为 5
      check_sum=temp;                    //得到校验码低位字节
    }
    else if(receive_flag==5)             //如果接收状态为 5
    {
      receive_flag=0;                    //接收状态转为 0
      check_sum|=(temp<<8);              //得到校验码高位字节
      …                                  //数据处理代码
    }
  }
```

不能说上面代码是错误的，正常情况下上面的代码运行结果都不会有什么问题，但这部分代码是用来接收另外设备通过 UART 发送过来的数据，这个收发过程可能会因为外部干扰等原因造成传

输数据出错，当这种情况发生时，程序就会变的不在稳定可靠。实际上前面的代码也有考虑到数据传输可能会出错，在数据包中加上校验码就是用来判断数据传输是否出错，如果数据传输出错，校验码会不对，这样程序就可以把所接收到的数据包做相应处理。

但对这部分代码传输出错的考虑不够周到，只是去判断数据包是否出错，并没有去考虑出错位置不同可能带来的不同影响。干扰是随机的，所以传输的错误可以产生在数据包的任意位置，如果果然刚好在传送数据包长度的时候导致数据传输出错，比如现在数据包长度原本是 0x200(512 字节)，干扰导致这个长度变为 0x1200(4608)字节，这时前面代码面临的问题就非常严重，会从 receive_buf[]的起始位置开始存放 4608 字节后才终止接收过程，而 receive_buf[]只有 1024 字节的空间，也就是说后面会有 4608-1024=3584 字节的空间会被指针错误修改。

象这种错误的调试跟踪会非常困难，因为它发生的概率本身就非常小，可以说连续发生两次的几率几乎为零，而这个错误一旦发生，很有可能让整个程序崩溃，即便当时想通过仿真调试工具去查看现场也是很困难，可以说是无法通过调试的方法手段来找出这类错误根源，如果想避免这类错误，只能是在编写代码阶段就充分考虑到各种可能性，在代码中对各种异常做出相应保护而避免意外出错。

真可谓是指针虽好，可用起来并不简单，刚开始接触单片机 C 语言编程的朋友还是多用数组来编写代码比较好一些，当自己对 C 语言有一个良好的掌握程度后再去逐步使用指针，使用指针的时候一定要仔细确认指针是不是依照自己的想法在变动，最好在指针改变后查看一下地址是否正确。

4.9. 循环延时

用 C 语言编写单片机程序时，常会遇到一些需要延时等待的情况，不少程序员为了图方便经常会用 for(i=0;i<1000;i++)这样循环来实现延时，我自己有时候也都这么做，这不是一个好的方法。

首先这样的循环代码写出来后程序员自己也不清楚这段代码到底延时有多久，还需要用仿真器观察对应汇编指令或者用仪器测试延时时间的大小。

其次所得到延时时间稳定性不够好，如果在延时循环中有中断产生就会使得延时变大，所以在实际程序运行时执行完这个循环所耗用的时间存在长短不一的可能，中断对循环延时的影响还不是特别大，因为其它方式实现延时也会受到中断的影响，比如我们想实现延时 100us，延时循环中一个需要耗时 200us 的中断产生，这样不管用什么方法，当响应完中断返回时时间就多出了 200us，也就是说这时无论什么方法来实现的延时都不会少于 200us。对循环方式延时影响真正大的是编译优化、系统时钟改变这两种操作。

我们来看一下用 ADS 编译器不同优化条件下对循环 for(i=0;i<1000;i++)的影响，先关闭优化功能，可以看出循环部分代码需要四条汇编指令

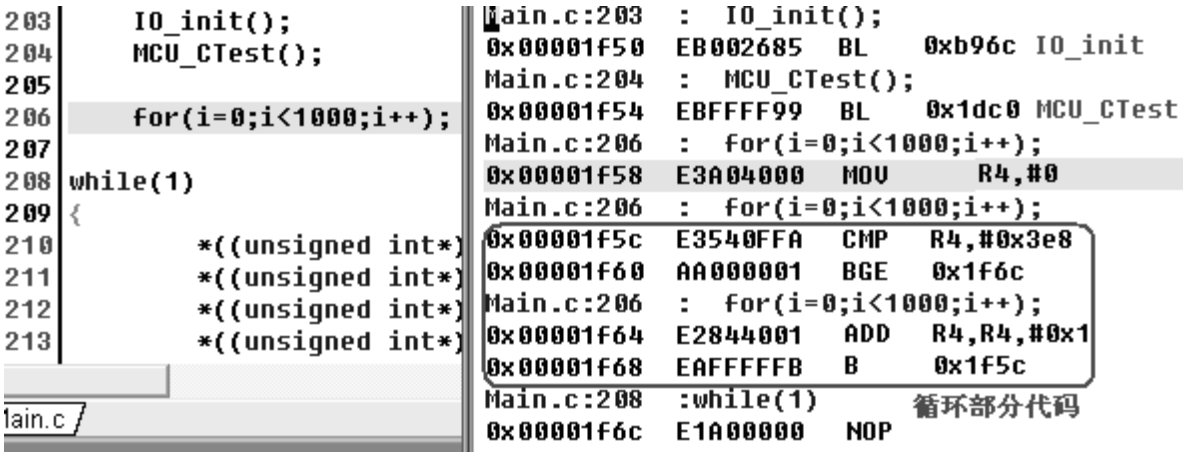

图 4.9.-1 ADS 对 for 循环无优化编译结果图

再来看看打开优化功能后的情况，循环部分的代码从原来的四条变成了三条，也就是说在硬件不做任何变动的情况循环的时间会改变为原来的四分之三，这个影响是非常明显的，很有可能就形成延时不够的情况。

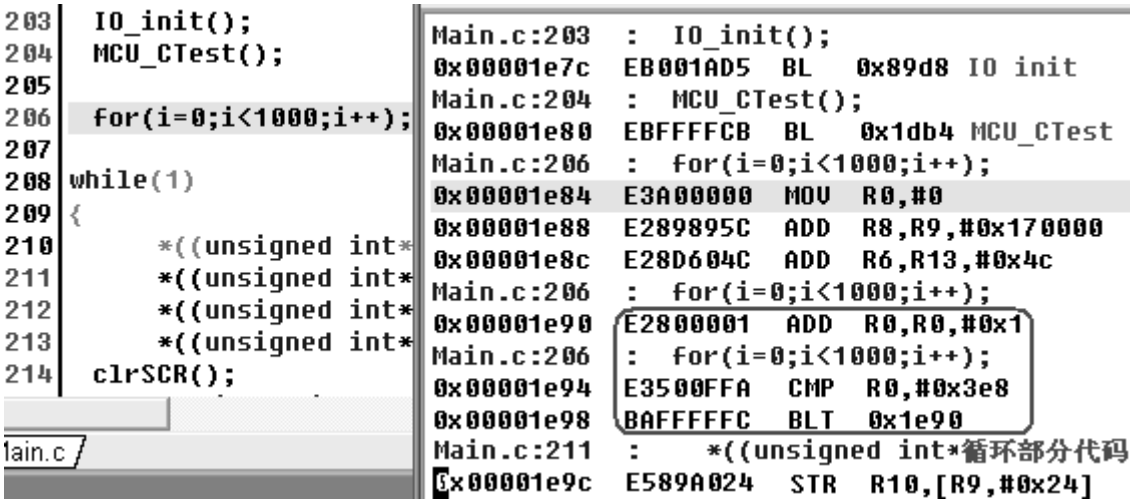

图 4.9.-2 ADS 对 for 循环有优化编译结果图

对于系统时钟改变的情况很容易理解，系统时钟被改变，每条指令执行的时间同样会相应发生变化，举个极端的例子，原本是用 8MHz 的晶振现在改用 4MHz，每条执行执行的时间就会增加一倍，循环所花的时间自然也就增加了一倍。有人可能有疑问，我们在实际应用中不会改用不同频率的晶振，那不就不用担心这样的情况了吗？是的，大多数时候不用担心这个问题，但在某些特殊情况下就需要考虑该问题，比如功能强大一点的 MCU 可以通过 PLL 来设定系统时钟，即便不改变晶振频率也可以改变系统时钟，当系统时钟频率高的时候 MCU 处理速度快，但会有芯片发热大、对外辐射增加、稳定性变差这些问题产生，所以有的产品当需要高速处理数据的时候就将 PLL 设为高频率以满

足数据处理需求，当数据处理完又恢复为 PLL 低频率得到一个稳定可靠的工作状态，对于这种情况循环延时就会不准确。

我们知道电源电压波动、电阻阻值误差这些因素都会影响 RC 振荡器的工作频率，所以当 MCU 使用 RC 振荡器的时候也会让循环延时不准，PLL 还可以根据对 PLL 的设定值判断当前工作频率来解决时钟变化导致的影响，而 RC 振荡器则完全无法知道时钟的变化，通过程序是无法让循环延时修正误差。

如果想要避免因优化对循环延时产生的影响，建议用汇编指令来延时，编译器不会对汇编指令进行优化，所以编译器优化的选择与否不会影响汇编指令的那部分代码，汇编语言写的延时代码只要系统时钟不变就能保证延时恒定。另外通过查询 MCU 的编程手册，可以知道每条汇编指令所需要的机器周期数，汇编语言写的延时代码可以精确一个机器周期，程序员通过计算汇编指令数就能知道延时时间的长短，从而解决了用 C 语言写循环代码延时时间不透明的问题。

时钟改变的情况不容易处理，只能是在程序中依据硬件的实际情况提前做出预测，尽量做到延时时间不小于最低要求，从而保证程序的可靠执行，比如现在采用的是 RC 振荡器，经过分析预计其工作频率可能存在 2%的误差，如果需要延时 100 微秒，MCU 一个机器周期是 1/8 微秒，我们就要保证延时代码的理论延时不小于 102*8=816 个机器周期，否则可能有少量硬件延时达不到 100 微秒，对于实际应用还需要留有足够的余量，我一般是将这个余量定为 20%，也就是说按 120 微秒来编写代码。

这段以 6502 指令为基础的 C 汇编延时代码可以用来帮助大家了解如何在 C 语言编程中实现汇编代码延时，假定是指令周期等于机器周期，不考虑中断的影响。

```
void _delay_cycle(unsigned char n)
{
#ASM                ;标示下面为嵌入到 C 代码中的汇编代码
  TAX               ;执行该汇编指令需 1 个机器周期，C 函数参数传入
DELAY_100LOOP:      ;循环体执行需要 3*X-1 个机器周期
  DEX               ;执行该汇编指令需 1 个机器周期
  BNE  DELAY_LOOP   ;跳转回去需 2 个机器周期，结束循环需 1 个机器周期
#ENDASM             ;标示嵌入到 C 代码中的汇编代码结束
  //C 语言函数返回需要 2 个机器周期
  //RTS
}
```

调用示例：

```
_delay_cycle(100);
```

```
//C 函数输入参数 n 通过累加器 A 传人，将 n 写入累加器 A 需 1 个机器周期
//LDA  #100
//调用函数需 2 个机器周期
//JSR  _delay_cycle
```

可以看出函数被调用一次需要耗费 1+2+(1+(n*3-1)+2)=n*3+5 个机器周期，n 的取值范围为 0~255，这样通过调用该函数可以实现的延时为 5~770 个机器周期，间隔为每档 3 个机器周期,如果是一个系统时钟为 8MHz 的系统，该函数的理论延时是 0.625~96.25 微秒，间隔为 0.375 微秒。

上面是用循环实现延时，即便是用汇编代码来实现从我个人的角度看也不值得提倡，比较好的做法应该是充分利用 MCU 所带的 timer 资源，用 timer 中断实现延时，这样做在计算延时时间时要便捷不少，同时还能保证到具有良好延时精准度。

这章最后用 C 的伪代码给大家示意一下如何可以通过 timer 中断实现精确延时，假定可以将 timer 中断设置到比较高的中断优先级，通过 timer 中断来停止在主程序中启动的 testing，这样的做法可以让延时非常精准。

```
void (*timer_callback)(void);        //定义一个函数指针
unsigned long timer_int_flag=0;      //定义 timer 中断标志变量
void timer_irq(void)                 //timer 中断函数
{
  if(timer_callback!=NULL)
  {
    timer_callback();                //如果函数指针不为空则执行对应函数
                                     //如指针为 stop_testing 则执行 stop_testing ()
  }
  DISABLE_TIMER_IRQ();               //禁止 timer 中断，具体代码忽略
  timer_callback=NULL;               //清函数指针为空
  timer_int_flag=1;                  //置 timer 中断标志
}
void timer_delay(unsigned long us,void *callback)
                                     //将第二输入参数的指针给函数指针
{
  timer_callback=callback;           //将第二输入参数的指针给函数指针
  set_timer_counter(us);             //将 timer 相关寄存器设定为所需要的内容
                                     //具体代码忽略
  ENABLE_TIMER_IRQ();                //使能 timer 中断，具体代码忽略
}
```

```
void start_testing(void)              //启动测试函数
{
  ...                                 //启动测试的代码，具体代码忽略
}
void stop_testing(void)               //停止测试函数
{
  ...                                 //停止测试的代码，具体代码忽略
}
void main(void)
{
  ...
  timer_int_flag=0;                   //清 timer 中断标志
  start_testing();                    //调用启动测试函数
  timer_delay(10000,stop_testing);    //延时 10000us 后产生 timer 中断停止测试
                                      //在 timer 中断中会执行函数 stop_testing()
  while(time_int_flag==0);            //等待 timer 中断标志被置为 1
  ...
}
```

例子中的伪代码没有考虑代码自身产生的延时，也没有考虑如果有优化时对全局变量应采取的相应保护措施。留一个问题给大家，为什么需要将停止测试的函数 stop_testing()放在 timer 中断中，而不是在 while(time_int_flag==0);之后调用?

4.10. 运算表达式

用 C 语言编程，少不了使用运算表达式，C 语言的运算表达式和日常生活中的数学表达式非常接近，所以运算表达式对使用者来说非常直观，也很容易理解。数学里面一个数是正是负、是整数还是小数可以用负号和小数点来让人知道这个数的类型，知道了数的类型人们就可以按照数学法则进行运算。但 C 语言毕竟不是我们上小学就开始学习的数学，因为计算机技术的限制，在 C 程序中出现整数和小数、有符号数和无符号数混用等情况时，为了保证计算结果正确，C 语言对这些情况做出了一些自己的特性约定，让不同类型的数混用时按某一个规律自动进行转换，这样一来就出现同样的运算表达式 C 语言和日常生活中的数学运算结果不相同的情况。

在用 C 语言进行编程时，一定要留意它和日常生活中数学在处理运算表达式方面的不同，否则就会让程序出现一些本可以避免的错误。

我曾经遇到过这样的情况，用 check_sum 对通讯的数据进行一个简单的保护，保护的方法相当

简单，就是将通讯的数据累加求和，发送完数据后再将计算所得的和发送给接收方，接收方也同样将所接收的数据累加求和，然后和发送过来的校验和做比较，如果两者一致就认为数据传输过程是可靠的。按说这个方法很简单，不大可能出错，可实际情况就是我所写的C程序会出错，会将原本传输正确的数据错误判断为传输出错。

出错的原因就是对C语言的运算表达式理解有误，C语言在进行处理运算表达式的时候有一条基本法则：如果在运算表达式中有不同类型的变量，运算的时候是signed遇到unsigned会自动将signed转换成unsigned，数据位宽度小的自动转换成和大的数据位宽一致。

暂先不对这条基本法则做过多的解释，看看我当时是如何出错的，接收方C代码是将接收到的数据放到unsigned char data_buf[]中，再将全部数据累加求和得出校验码。发送方采用汇编编写代码，这里不列出具体代码。

```
unsigned char buf[7];              //前面6个字节放数据，第7个字节为校验码
unsigned char receive_flag;        //用来表示校验结果的标志变量
if((buf[0]+buf[1]+buf[2]+buf[3]+buf[4]+buf[5])==buf[6])
{
  receive_flag=DATA_CHECK_RIGHT;  //校验结果正确，认为传输可靠
}
else
{
  receive_flag=DATA_CHECK_ERROR;  //校验出错，认为传输有错误
}
```

传输的数据包长度不长，为节省代码，没有用循环语句而是直接将6个字节的数据相加求和，当时我的理解是 if((buf[0]+buf[1]+buf[2]+buf[3]+buf[4]+buf[5])==buf[6])表达式中全部是同一类型的unsigned char数组成员变量，那计算过程也应该是自动按unsigned char类型处理，如果数据相加求和溢出也会将求和自动转换成 unsigned char 类型，比如这 6 个字节加起来的和为0x0123,会自动将这个和处理成0x23，溢出的高位将被舍弃掉，这样代码是可靠的。

实际情况并不是我所想像的样子，假如现在 buf[7]={0x11,0x22,0x33,0x44,0x55,0x66,0x65}，人工计算0x11+0x22+…+0x66=0x165，那么汇编计算出来的校验码应是0x65，和数组最后一个字节一致，表明这次传输的数据是可靠的，但C代码执行的结果并不是这样，会报告数据校验出错，通过调试查看C代码对应的汇编指令知道错误原因，对数据相加的结果不是和我所想的一样会自动转换成unsigned char类型，这个时候是将加出来的结果按MCU的位宽来处理，所以校验出错，正确代码应如下所示。

将运算表达式的结果先进行强制转换。

```
if(((buf[0]+buf[1]+buf[2]+buf[3]+buf[4]+buf[5])&0xFF)==buf[6])
//或 if((unsigned char)((buf[0]+buf[1]+buf[2]+buf[3]+buf[4]+buf[5]))==buf[6])
{
```

```
  receive_flag=DATA_CHECK_RIGHT;  //校验结果正确，认为传输可靠
}
else
{
  receive_flag=DATA_CHECK_ERROR;  //校验出错，认为传输有错误
}
```

或者先将运算表达式的结果赋给指定变量。

```
unsigned check_sum;
check_sum=buf[0]+buf[1]+buf[2]+buf[3]+buf[4]+buf[5];
if(check_sum==buf[6])
{
  receive_flag=DATA_CHECK_RIGHT;  //校验结果正确，认为传输可靠
}
else
{
  receive_flag=DATA_CHECK_ERROR;  //校验出错，认为传输有错误
}
```

之所以产生这样的错误，是因为我对 C 语言的运算表达式的处理方式理解不够透彻，对运算结果用想当然的方法去理解。接下来看看运算表达式中有不同类型的变量时 C 语言究竟是如何进行自动转换的，例子中的结果以 ADS 编译器结果为准，其它编译器可能得出不同结果。

```
signed char x;
unsigned char y;
unsigned short z;
x=-1;          //用十六进制表示为 0xFF(-1)
y=x;           //运行结果为 z=0xFF(255)
z=x;           //运行结果为 z=0xFFFF(65535)
```

当 y=x 时，y 和 x 都是 char 类型，直接将 x 的内容从有符号转换成无符号类型就赋给 y，但 z 和 z 除了无符号和有符号的区别外数据位宽也不一样，分别是 16bits 和 8bits 宽，x 自身为-1，但是 8bits 类型，当将其赋给 z 时，比 y=x 需要多数据位数的扩展，这样就要将原本 8bits 的-1(0xFF)转换成 16bits 的-1(0xFFFF)，所以最后 z 的值为 0xFFFF，和 x 的 0xFF 不同。

这里有一点要特别提醒大家：在 C 语言里不要将-1 和 0xFF 视为等同数据，大多数编译器会将 0xFF 当做 255 来处理。

```
signed char x;
```

```
x=-1;          //用十六进制表示为 0xFF(-1)
if(x==0xFF)    //实际上这句等效于 if(x==255)
{
  x=-1;        //程序不会进入此处
}
if(x==-1)
{
  x=0;         //程序会进入此处
}
```

```
Main.c:88    :  if(x==0xFF)
0x00001e14   CMP            R1,#0xff
0x00001e18   BNE            0x1e20
Main.c:90    :    z=0; 不同的比较指令
0x00001e1c   MOV            R0,#0
Main.c:92    :  if(x==-1)/ CMN表示和-1比较
0x00001e20   CMN            R1,#0x1
0x00001e24   BNE            0x1e2c
Main.c:94    :    z=0;
0x00001e28   MOV            R0,#0
Main.c:96    :}
0x00001e2c   BX             R14
```

图 4.10.-1　ADS 对-1 和 0xFF 编译对比结果图

接下来是一段奇妙的代码，也许会给你一种不可思议的感觉，但实际结果确实如此，如果你理解了这段代码，我想你对 C 语言运算表达式中的自动转换规则已经是非常清楚。

```
signed char x;
x=128;         //注意 signed  char 有效范围为-128~127
if(x==128)
{
  x=128;       //程序不会进入此处
}
if(x==-128)
{
  x=0;         //程序会进入此处
}
```

是不是有点奇妙？明明用 C 代码给 x 的值是 128，在后面进行比较就变成了-128。如果你明白

了计算机内部的数据表示方法后就不难理解其中奥妙，我们知道计算机内部数据采用二进制表示方法，正数是多少就是对应的二进制数，负数则先将数的二进制数取反，然后加一。十六进制是让人们对二进制数可以更直观表示的中间方法，数值自身的内容完全和二进制一致。来看看十六进制是如何表示 8bits 数的：1 为 0x01，-1 为 0xFF(0x01 取反得到 0xFE 再加一得到 0xFF)。

C 语言在处理运算表达的时候，如果表达式中有数字，通常会将这个数字默认为与 MCU 位宽一致的整型数，再用这个整型数的二进制数进行运算处理，例如上面的 x=128 编译器会先将 128 转换成 0x00000080，因为是要将这个数给 signed char 类型的 x，所以在产生汇编指令的时候就只需要 0x80 作为操作数，另外三个字节在生成汇编指令阶段就给抛弃掉。

```
signed char x;
x=0x1234;
if(x==0x34)    //产生汇编指令的时候只将低位字节 0x34 给 x
{
  x=0;         //程序会进入此处
}

signed char x;
x=-129;        //十六进制为 0xFFFFFF7F
if(x==0x7F)
{
  x=0;         //程序会进入此处
}
```

到这里就更加容易理解前面的奇妙代码。

```
signed char x;
x=128;         //实际上是将十六进制数 0x00000080 的低位字节 0x80 给 x
if(x==128)     //实际上是将 x 和十六进制数 0x00000080 进行比较
               //可以理解成 if((-128)==128)，也就是 if(0xFFFFFF80==0x00000080)
{
  x=128;       //显然比较结果不相等程序不会进入此处
}
if(x==-128)    //实际上是将 x 和十六进制数 0xFFFFFF80 进行比较
               //可以理解成 if((-128)==（-128）)，也就是 if(0xFFFFFF80==0xFFFFFF80)
{
```

```
    x=0;          //比较结果相等程序会进入此处
}
```

转换规律小结：

有符号类型和无符号类型混用时自动转换为无符号类型。

数字默认为与 MCU 位宽一致的整型数值。

不同位宽的数据（或变量）混用时自动转换成宽度更宽的类型。

不同宽度的数据进行赋值运算时候自动进行高位扩展或截取。

还有一点同表达式有关，C 语言对于运算符自己有一张优先级表，不少人编写程序时可能是为了减少表达式长度或证明自己对 C 很熟悉，在代码中充分利用运算符的默认优先级。

```
if(a>b && b>0)
a=10>4&&!(100<99)||3<=5
```

不能说这种写法错误，有些经典样例代码可能都是按此方式书写，但我个人绝不赞同这种写法。即便是经典样例代码，也是基于艺高人胆大的基础之上，不值得推广，稳妥的做法是用括号将表达式的优先顺序括起来。这样做虽然代码会繁琐一些，但通用性和可读性强，就算是记不住各级运算符优先级的人也不会弄错运算的先后顺序，编译得到的汇编指令也不会变多。

```
if((a>b) && (b>0))
a=(10>4)&&(!(100<99))||(3<=5)
```

后一种写法是不是要清晰许多？如果你以前习惯采用 C 默认优先级的方式，来看一下你真的记住了所有的运算符优级顺序了吗？

优先级	运算表达式
最高	()(小括号) []（数组下标）.(结构成员) ->(指针型结构成员)
↑	!(逻辑非) ~(位取反) -(负号) ++(加 1) —(减 1) &(变量地址)
↑	*(指针所指内容) type(函数说明) sizeof(长度计算)
↑	*(乘) /(除) %(取模)
↑	+(加) -(减)
↑	<<(位左移) >>(位右移)
↑	<(小于) <=(小于等于) >(大于) >=(大于等于)
↑	==(等于) !=(不等于)
↑	&(位与)
↑	^(位异或)
↑	|(位或)

↑	&&(逻辑与)
↑	||(逻辑或)
↑	?:(?表达式)
↑	= += -=(联合操作)
最低	,(逗号运算符)

图 4.10.-1 标准 C 运算符优先级表

4.11. 溢出

除了宇宙，一切事物都是有限的，水杯子满了水会溢出，水缸满了水也会溢出，水池满了水还是会溢出，就是水库满了同样也是溢出，没有其它选择。在编写单片机程序时，程序员同样要面对数据溢出情况，如果在写程序时不预先对数据溢出情况加以考虑，最终结果很可能是错误隐藏在程序之中，在特定条件下就会暴露出来导致程序出错。

实际上无论是汇编还是 C 语言，都需要面对数据溢出情况，只不过程序员在使用汇编编程时候，因为没有 C 语言中的 char/short/long 这些数据类型，需要程序员自己对数据的位宽进行处理，所以这个时候程序员往往会下意识的留意到数据溢出情况，而使用 C 语言时，数据类型的处理由 C 语言编译器完成，程序员关注的重点会放在程序结构方面，常常忽略数据类型这些细节，所以相较汇编语言使用 C 语言编程时更容易在数据溢出方面出问题。

我自己就有在数据溢出方面考虑不周差点让产品出大问题的经历，该产品是用一片小容量的 SPI FLASH 芯片来存储数据，数据存储采用自定义的文件系统格式，为了让存储的数据可靠性更高，我将文件系统的 FAT（文件分配表）的内容用 CHECK SUM 进行校验，只有校验正确的记录才会被当作有效数据。

……（详见完整版）

4.12. 强制转换

不少用 C 语言进行单片机编程的人都遇到过这样的麻烦，想用指针去读写某个地址，如果这个地址是数组没有什么问题，但如果这个地址是用数字 0x1000 这样表示的绝对地址，编译的时候总提示出错，从代码看又好象没有什么错误，搞得是一头雾水，不胜其烦。

```
char *p;
char data_buf[1024];
p=data_buf;                    //这样给指针 p 赋值正确
p=&data_buf[0];                //这样给指针 p 赋值编译出错
```

```
p=0x1000;                    //这样给指针 p 赋值编译出错
```

要解决这种问题其实很简单，将出错的语句改成下面形式，编译器则不会报告错误。

```
p=(char*)&data_buf[0];       //编译正确
p=(char*)0x1000;             //编译正确
```

造成上面编译无法通过的原因是赋值操作等号两侧的类型不一致，C 语言语法要求赋值操作时两侧的数据类型必须一致，否则就会告警或报错。C 语言这样处理可以防止程序不同类型数据混用时产生程序员未预料到的逻辑错误，比如上面 p 定义成 char 类型的指针，虽然指针地址也是一个数字，但不能将起等同数字，p=0x1000 这样的语句存在表述不清的问题，因为指针有许多种，光用一个 0x1000 数字我们不能确定这个数字代表的到底是 char 还是 short 或其它类型的指针，加上(char*)后我们就能明确其具体含义。

对于编译器对于赋值操作等号两侧的类型不一致会出错这一结论我们可以多做一些验证，比如有两个不同类型的指针，当把一个指针的地址赋给另外一个指针时同样也会报错。

```
char *p1;
short *p2;
p2=(short*)0x1000;
p1=p2;                       //编译出错
```

改成下面形式：

```
p1=(char*)p2;                //编译正确
```

象(char*)p2/(char*)0x1000 这样的用法就是强制转换，当编译器遇到这样的处理语句后就知道这个地方程序员已经发现数据类型可能不一致，了解数据类型转换可能产生的影响（比如将浮点数转为整型会丢失小数部分），编译器可以进行转换操作后一并编译，如果不加这样的语句编译器会担心程序员没有考虑到数据转换产生的影响而报告错误。

强制转换是一种非常实用的操作，刚接触单片机 C 语言编程的人可能还不会感觉到这类操作的必要性，随着实际开发项目的增多就会逐渐感受到实在在哪些方面，比如常常会发现所写的程序有一大堆警告，但编译依然能通过，这些警告错误相当大一部分都可以用强制转换操作消除掉。

可能有人会有这样的想法，只要编译器编译通过，有几个警告错误无所谓，这种想法不可取，一个好的 C 程序应该是没有任何警告，每一个警告都代表编译器发现一个存在风险的位置，只是编译器也认为这个风险对程序的影响可能比较小，所以编译通过，良好的习惯是当自己的程序编译有警告产生时，应该找出警告的原因，警告往往比错误更难找原因，所以开始可能需要花较多的时间来消除这些警告，要知道警告的原因就那么多，只要积累一定经验后查找警告原因就会变容易。

来看一个浮点数转整型时产生的警告：

```
long x;
float y;
y=(float)3.14;
x=y;                         //将浮点类型的变量赋置给整型变量，产生警告
```

改成下面形式，警告消除。

```
x=(long)y;                    //添加(long)强制转换后警告被消除
```

实际上有无(long)强制转换最终编译产生的代码是一样的，最后 x 都是等于 3，所以编译器这里只是警告，并不报告错误，那警告有什么作用呢？可以提醒程序员留意这里，不要在后面把 x 当作 3.14 继续使用，应该是 3，加了(long)强制转换编译器就理解为程序员发现了 3.14 和 3 的区别，不用再用警告来提醒程序员留意这个地方。

强制转换的另一个好处是可以让程序员对数据类型的控制更加灵活透明，用汇编语言编程的时候各种不同数据类型之间的转换需要自己按照数据类型的格式进行处理，这样虽然程序员在程序中随时都清楚自己需要处理的数据类型，但数据类型之间的转换操作用汇编实现繁琐而复杂，也容易出错，C 语言的强制转换将类型转换的工作交给编译器完成，可以让程序员省心不少。

继续浮点数强制转整型的测试：

```
long x;
float y;
y=(float)3.14;
x=y;                          //执行完这句 x 值为 3，编译器对于这行有警告提示
x=(long)y;                    //执行完这句 x 值为 3，无警告提示
x=*((long*)(&y));             //执行完这句 x 值为 0x4048F5C3
```

对于这个例子中的前两行赋值操作不存在什么疑问，都得到我们所预期的整数结果 3，但最后一行的赋值操作 x=*((long*)(&y))却得到 0x4048F5C3 这样一个有趣的结果，为什么会和前两行的赋值操作所得的结果不相同呢？让我们来做一个对比分析。

```
24:   y=(float)3.14;
0040109E   mov   dword ptr [ebp-110h],4048F5C3h
25:   x=y;          将4048F5C3h存入y所在位置[ebp-110h]
004010A8   fld   dword ptr [ebp-110h]从y所在位置[ebp-110h]取输入参数
004010AE   call  __ftol (004011ac) 调用浮点转长整型函数
004010B3   mov   dword ptr [ebp-114h],eax
26:   x=(long)y;   将转换结果存入x所在位置[ebp-114h]
004010B9   fld   dword ptr [ebp-110h]
004010BF   call  __ftol (004011ac)
004010C4   mov   dword ptr [ebp-114h],eax
27:   x=*((long*)(&y));
004010CA   mov   eax,dword ptr [ebp-110h]从y所在位置[ebp-110h]取出长整型内容
004010D0   mov   dword ptr [ebp-114h],eax将取得的内容存入x所在位置[ebp-114h]
```

图 4.12.-1　VC 强制转换结果图

对比编译器产生的汇编代码可以看出 x=y 与 x=(long)y 所产生的码是全一样，都是调用了一个浮点数转整型的函数进行转转，x=*((long*)(&y))则不同，并没有调用浮点数转整型的函数，是直接将 y 在 RAM 中的地址取出来，然后将这个地址作为一个 long*的指针取去里面的内容。

这里我将 x=*((long*)(&y))的操作步骤分解出来：

a. &y 是取存放数据 y 的地址。

b. (long*)(&y)是将取得的地址转换成一个 long*类型的指针。

c. *((long*)(&y))取出这个指针所指的内容。

从这些步骤可以看出整个过程并没有数据强制转换，只是强制转换了一个指针类型，这个转换并不会改变 RAM 中的内容，0x4048F5C3 如果按浮点数格式代表 3.14，如果按长整型则就是十六进制 0x4048F5C3 自己，所以最后一行得到的结果和前两行赋值操作得到的结果不同。

如果我们另外执行 x=*((float*)(&y))的操作，此时 x 又会得到 3 的结果，这里不给出汇编验证的结果，有兴趣的朋友可以自己进行验证。

4.13. 高效实用位运算

标准 C 自身提供有移位、逻辑与或非等位运算指令，但对于一名用 C 语言编写电脑应用程序的程序员，他们往往不太习惯使用 C 语言的位运算功能，因为他们编写程序时候所面对的硬件平台是电脑，电脑所提供的 RAM 资源是单片机无法相提并论的，而且电脑应用程序编写强调的是硬件无关，不需要程序员去了解控制硬件设备，这样使得他们对位运算的依赖性非常低，自然而然就会少用到位运算。

单片机程序则不一样，单片机程序尤其是底层驱动程序需要直接控制硬件，对硬件的控制主要是是设置硬件的相关控制寄存器，这些寄存器往往是一个位就控制一种功能，所以单片机程序可以说是离开了位运算操作真就不行。来看一个 MCU 进行 IO 输入输出功能选择的寄存器，该寄存器控制 IOD 的 8 条 GPIO 口，bit0~bit7 每一个位对应控制一条 GPIO，如果我们想将某条 GPIO 设置成输出，只要将对应的控制位设为 1 即可。

P_IOD_ OutputEn　　0x112000C0　　IOD Output Enable Register

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Function	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Default	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Function	-	-	-	-	-	-	-	-	IOD_Output_Enable							
Default	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	Function	Type	Description	Condition
[31:8			Reserved	

]				
[7:0]	IOD_Output_Enable	R/W	IOD Output Enable	0 = Disable 1 = Enable

图 4.13.-1 IO 控制寄存器表

如果不使用位操作指令，就只能是程序知道所有 8 条 GPIO 的输入输出选择，需要更改设置时候将这 8 个控制位一同设为新状态。这样做对于程序员来说无异是一场噩梦，会让程序变得晦涩难懂，一不小心就有可能出错。使用位操作指令可以很好的解决这个问题，比如上面的 IOD，我们现在需要将其中的 IOD3 更改输入输出选择，使用位操作可以让程序一目了然。

定义寄存器指针：

```
#define P_IOD_OutputEn (volatile unsigned long *0x112000C0)
```

定义 bit3 的宏，采用 1 向左移位 3 位的形式更直观：

```
#define BIT3 (1<<3)
```

将 IOD3 设为输出，先读出 IOD 的所有 8 个输入输出控制位，然后通过位或运算将 bit3 设为 1，再将新的设定值回控制寄存器，这样就只将 IOD3 设为输出，其它 GPIO 输入输出状态保持不变：

```
*P_IOD_OutputEn=BIT3|(*P_IOD_OutputEn);
```

将 IOD3 输入和设为输出一样，只是将 bit3 清 0，其它位保持不变：

```
*P_IOD_OutputEn=(~BIT3)&(*P_IOD_OutputEn);
```

另外一些小的 MCU 所能提供的 RAM 资源有限，可能只有几十个字节的 RAM 可供单片机程序员使用，对于程序中只需要 0 和 1 两种状态的状态标志最好是用单个位来表示，否则容易出现 RAM 资源不够用的情况。

标准 C 在位操作方面功能相对有限，比如前面更改 IOD3 的输入输出控制，需要经过“将原设定值从寄存器读出→与/或操作需要设定的位得到新设定值→将新设定值写回寄存器”这三步才能完成，显然代码效率并不高，如果能够直接一步就完成 IOD3 的设定，无疑是一个不错的选择，标准 C 在这方面就存在不足，虽然可以通过位结构来定义位，但实质上只是从语法上看上去简单一些，在汇编层面看还是需要前面三步通过与非操作来设定相应位。

下面定义了一个位结构。

```
struct{
  unsigned incon: 8;          //incon 占用低字节的 0~7 共 8 位
  unsigned txcolor: 4;        //txcolor 占用高字节的 0~3 位共 4 位
  unsigned bgcolor: 3;        //bgcolor 占用高字节的 4~6 位共 3 位
```

```
    unsigned blink: 1;          //blink 占用高字节的第 7 位
}ch;
```

对于 ch.blink=1 这样的操作，编译器得到的汇编指令等同 ch=BIT7|ch，因为电脑的 CPU 和现在一些使用 ARM 内核的 MCU 都没有提供可以将单个位置 1 或清 0 的指令，为了提供良好的通用性，编译器处理这类代码会得到同样的汇编指令。

那是不是单片机的 C 语言程序只能按这种三步走的方式来设置控制位了呢？答案是否定的，为了得到更好的代码效率，不少小的 MCU 都支持位操作，也就是有一些 MCU 支持将单个位置 1 或清 0 的汇编指令，对于这类 MCU，如果支持 C 编程那么厂商提供的 C 编译器肯定也支持单个位置 1 或清 0 的操作，编译器在标准 C 的指令之外提供了额外的指令来支持位操作。

比如最常见的 51 系列的 MCU 就支持单个位置 1 或清 0，来看看用 KEIL C 的编译器是怎么样对单个位进行操作的。

KEIL C 在标准 C 之外提供 sbit 指令，用来定义位变量。

```
sbit MCU_LED0 = P1^2;          //将 GPIO P1.2 定义成为 MCU_LED0，P1 寄存器为 90H
MCU_LED0=1;                    //P1.2 输出 1，对应汇编指令 SETB 90H.2
MCU_LED0=0;                    //P1.2 输出 0，对应汇编指令 CLR 90H.2
```

注意这里是将 MCU_LED0 用 sbit 直接定义成位变量，和位结构中的位变量完全不同，当编译器遇到 MCU_LED0 就会将当前的 C 代码用 51 的位操作汇编指令实现。

不是所有的单片机都直接支持直接对单个位置 1 或清 0 的操作，对于一些采用通用编译器和 CPU 内核的单片机为了平台的一致性，大都不支持这种做法，比如使用 ARM 内核的各种 MCU 就不支持直接对单个位置 1 或清 0 的操作，这类 MCU 还是必须用三步走的方式来完成。

这样看来对单个位置 1 或清 0 要想做到高效还得需要 MCU 自身有相应位操作指令支持，并不是对所有 MCU 都有效，那有哪些方面可以体现位操作的高效性呢？让我们来看看在乘除运算方面的影响。

有的 MCU 自己支持乘法除法指令，但乘法和除法指令相较其它指令往往会占用更多的指令周期，如果 MCU 不支持乘法除法指令，那就需要用软件方法来实现，更为缓慢，这样对于 C 语言中的乘除运算 MCU 会占用较多的时间执行，总体上说对于乘除运算处理 MCU 难以高效。

有些特殊的乘除运算我们可以用位操作的移位指令来实现，这样做可以将乘除运算转为高效的位运算，比如 x=x/2 和 x=x*8，如果用 x=x>>1 和 x=x<<3 来实现的话效率肯定会更高。

除了可以让程序变得更加高效外，使用位操作指令还可以让程序简洁可靠，如果不用移位指令，要对某个数据位进行定义就必须先将该位为 1 其它位全为 0 的十六进制数人工计算出来，比如现在要定义 bit11，就得知道 bit11 为 1 其它位为 0 的十六进制数是 0x0800，然后#define BIT11 (0x0800)。如果只对单个位进行定义对于二进制和十六进制转换关系熟悉的程序员还不是难事，但如果现在需要定义 32bits 中的 2、9、15、21、28 这些位的组合呢？恐怕就有点麻烦，稍有不慎就可能定义出错。

来看一下利用了移位操作指令的情况：

```
#define BIT7 (1<<7)
#define BIT_2_9_15_21_28 ((1<<2)|(1<<9)|(1<<15)|(1<<21)|(1<<28))
```

是不是变得非常简单，复杂的二进制和十六进制转换工作由编译器完成，想定义错误都难啦。也许会有朋友担心这样做会不会产生代码效率变低的问题，从代码看这样的宏定义包含不少移位操作，按照编译原理编译时会用这个定义替换后面代码中相应的宏，那只要是用到这样宏的地方都会包含这些移位操作，和前面直接定义成十六进制数相比好像代码效率要低。有这种顾虑说明对 C 语言的宏有了一定了解，在实际应用中不用担心这个问题，现在稍微做得好一点的编译器都考虑到了这些问题，编译器会尽量让编译出来的汇编指令简洁高效，象上面的宏 BIT_2_9_15_21_28 编译器会先判断所定义的内容是否可以优化，如果能优化优化出简洁的表达式，这样对于位运算表达式((1<<2)|(1<<9)|(1<<15)|(1<<21)|(1<<28))会被预先处理为 0x10208204，然后用 0x10208204 替换后面宏 BIT_2_9_15_21_28。

单片机程序员常会遇到这样一道关于位操作的笔试题，请写一个函数将数实现高低位交换，比如现在有一个 16bits 的整型数，该函数将其 bit0 与 bit15 交换、bit1 与 bit14 交换.....，题目并不难，但却能难倒不少对 C 语言语法还算熟悉的人，究其原因就是不能熟练使用 C 语言有关位操作运算的指令。

这里给出两种利用位操作运算实现的函数代码，以便加深对位操作运算作用的理解。

函数一：

将数据从左向右、结果从右向左进行移位，每次移位先判断数据的最高位，如果该位为 1 则将结果的最高位也置 1，否则清 0，循环 16 次之后实现要求。

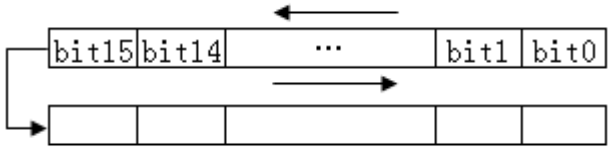

图 4.13.-2　移位操作示意图一

```
#define BIT15 (0x8000)        //定义最高位 bit15 的宏
void swap_bits(unsigned short *data)
{
  unsigned short data_temp,i;
  data_temp=0;                //先将结果所有位清 0
  for(i=0;i<16;i++)
  {
```

```
    data_temp>>=1;              //结果右移一位
    if((*data)&BIT15)           //如果数据最高位为 1 则结果最高位置 1
    {
      data_temp|=BIT15;
    }
    *data<<=1;                  //数据坐移一位
  }
  *data=data_temp;              //返回结果
}
```

函数二：

从最低位开始依次判断数据的每个位，如果该位为 1 则将结果中与之相映射的位置 1，否则清 0，循环 16 次后实现要求。

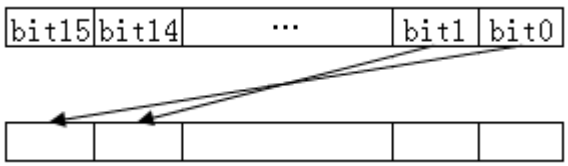

图 4.13.-3　移位操作示意图二

```
void swap_bits(unsigned short *data)
{
  unsigned short data_temp,i;
  data_temp=0;                  //先将结果所有位清 0
  for(i=0;i<16;i++)
  {
    if((*data)&(1<<i))          //判断当前的第 i 位是否为 1
    {
      data_temp|=1<<(15-i);     //为 1 则将结果中与之相映射的位置 1
    }
  }
  *data=data_temp;              //返回结果
}
```

4.14. 宏和 register

只要接触过 C 语言，大都熟悉宏的作用，即便是汇编语言，同样也有宏的概念，所以宏对于单片机程序员来说并不陌生。虽然大多数人不陌生宏，但能将宏用好的人并不多见，宏的优点不少书籍都有详细说明，这里我不再累述，而是通过一些例子来让大家感性的学习宏的使用。

1.宏不要使用小写字母

这一点是定义宏时的第一基本准则，目的是为了在程序中能将宏和函数、变量很明显进行区分，使用小写字母虽然不会产生错误，但对程序的易读性有负面作用，这是一条约定俗成的规则，没有太多的理由，如果你想用好宏，就必须将这条谨记在心。

2.程序中的各种定义尽量用宏

不少人都遇到过这样的事情，程序写得已经差不多了，发现某些地方存在问题需要修改，可能只需要修改一两种操作，可这一两种操作分布在程序的不同位置，于是就要一个一个的找出来，这个工作比写程序还麻烦，一不小心就会出现漏改的情况，许多时候造成这种麻烦的原因都是程序员没有很好的使用宏。

来看一个例子，一个简单的 MCU 提供 pa0~pa7 共 8 条 IO 口，最初将 pa2 定为输入按键，pa5 为输出 LED 控制信号，程序已经写好后突然提出新要求，要求硬件和另外一款芯片兼容，这样就需要交换 pa2 和 pa5 的功能才能做到。

如果不使用宏，这个修改就需要将程序里面所有关于 pa2 和 pa5 的操作找出来，然后逐个交换，这样需要修改的地方可能有几十甚至上百。如果使用了宏呢？情况则大不一样。

只需要将

```
#define LED_OUTPUT_ENABLE()    (_pa5=1)
#define LED_OUTPUT_DISABLE()   (_pa5=0)
#define IS_KEY_RELEASED()      (_pa2)
```

改为

```
#define LED_OUTPUT_ENABLE()    (_pa2=1)
#define LED_OUTPUT_DISABLE()   (_pa2=0)
#define IS_KEY_RELEASED()      (_pa5)
```

从原来的几十甚至上百个地方需要修改一下就变成了只需要修改三处，工作量大为减少，而且因为程序中相关操作都是使用宏来表述，完全不用担心改错。

既然对系统硬件资源的定义最好使用宏，代码中常用做比较等操作常量同样也最好是使用宏，再来看一个常量使用宏的例子。

```
#define VOLTAGE_12V6    (96)
#define VOLTAGE_12V3    (94)
#define VOLTAGE_12V0    (91)
#define CURRENT_300MA   (96)
#define CURRENT_250MA   (80)
#define CURRENT_200MA   (64)
```

同样为 96 的值，有的地方可能是用于电流比较，有的地方却又是用做电压比较，如果不用宏的话两者就会混在一起，当程序员需要修改电流比较用的 96 的时候，即使用了编辑工具的查找功能，还需要人为检查是不是电压比较，否则就会出错。用了宏则和前面硬件定义一样，只要在定义宏的地方修改一下就行，基本上不用再去管程序中的宏。

当然使用宏也不代表完美，当程序量比较大而且功能复杂时，需要写出大量的宏才能满足需求，这样就会让程序编写要麻烦一些；另外还有一些功能有时候用宏来实现也不是很方便，有时候遇到问题后要用不同方式尝试，这时会在程序中先用一些临时测试代码进行尝试，如果要让临时测试代码的增减与相关宏同步会让程序员感觉更繁琐；再就是调试的时候宏对应的代码往往不能单步跟踪，一执行就整个宏。

相较优点宏的不足显然只占极少部分，所以写程序的时候一定要养成多使用宏的习惯，当你代码写得足够多的时候就会越发感受到宏给你带来的便利。

3.防止头文件被重复包含

```
#ifndef MYHEADER_H
#define MYHEADER_H
  //头文件内容
#endif
```

示例：

头文件 myheader.h 内容

```
//#ifndef MYHEADER_H
//#define MYHEADER_H

//#endif
```

头文件 mydrv.h 内容

```
//#ifndef MYDRV_H
//#define MYDRV_H
void mydrv1(void);
void mydrv2(void);
//#endif
```

头文件 myapp.h 内容

```
//#ifndef MYAPP_H
//#define MYAPP_H
void myapp(void);
//#endif
```

文件 mydrv.c 内容

```
#include “mydrv.h”
void mydrv1(void)
{
  //代码
}
void mydrv2(void)
{
  //代码
}
```

文件 myapp.c 内容

```
#include “mydrv.h”      //需要引用该头文件申明 mydrv()以便在程序中调用
#include “myapp.h”
void myapp(void)
{
  //其它代码
  mydrv2();
  //其它代码
}
```

文件 main.c 内容

```
#include “mydrv.h”       //需要引用该头文件申明 mydrv1()以便在程序中调用
#include “myapp.h”       //需要引用该头文件申明 myapp()以便在程序中调用
void main(void)
{
  //其它代码
  mydrv1();
  myapp();
  //其它代码
}
```

编译器编译上面 main.c 的时候就会产生重复申明的错误，在主程序 main.c 的头两行分别引用了头文件 mydrv.h 和 myapp.h，留意在 myapp.h 一开始也有引用头文件 mydrv.h，这样就在将 mydrv.h 重复引用了两次，形成将 mydrv1()和 mydrv2()申明两次的错误。

如果将上面代码中的注释去掉，重复申明的错误将被消除，编译器在 main.c 的第一行先将头文件 mydrv.h 引用进来，此时发现并没有定义 MYDRV_H，于是进行定义，当处理第二行的 myapp.h 再次需要引用 mydrv.h 时，不同的编译结果产生，因为此时 MYDRV_H 已经被定义，所以 mydrv.h 中间的内容全部被跳过不进行再次编译，从而也就不会产生重复申明的错误。

4.用宏定义表达式时，要使用完备的括号防止出错

```
#define ADD(a,b)  (a+b)
#define ADD(a,b)  a+b
```

有括号 ADD(2,3)*4=(2+3)*4=20

无括号 ADD(2,3)*4=2+3*4=14

显然无括号情况得到的结果与我们所期望的不一致。

好像(a+b)的方式再不存在什么问题，其实不然，还有潜在风险隐藏在里面，对于 ADD(a,b)的例子可能不会出错，另外一个例子则不一样。

```
#define SQUARE(x)  (x*x)
```

SQUARE(1+2)=1+2*1+2=5

显然我们是想得到 1+2 结果的平方，正确结果应该是 9，现在错误的得到 5。

```
#define SQUARE(x)  ((x)*(x))
```

SQUARE(1+2)=(1+2)*(1+2)=9

这次得到的结果是正确的，所以为了防止宏定义出错，必须将表达式中的参数全部用括号括起

来。

正确的宏定义方式：

```
#define ADD(a,b)  ((a)+(b))
#define SQUARE(x)  ((x)*(x))
```

5.使用宏时不允许参数发生变化

接着看用来求平方的宏 SQUARE。

```
#define SQUARE(x) ((x)*(x))

int a=2;
int b;
b=SQUARE(a++);     //SQUARE(a++)=(a++)*(a++)，结果执行了两次加一操作得到 a=4
```

正确的用法是：

```
b=SQUARE(a);
a++;               //只执行一次加一操作得到 a=3
```

6.宏所定义的多条表达式应放在大括号中

下面的语句只有宏的第一条表达式被执行：

```
#define INTI_VALUE(x,y)\
  x=1;\
  y=2;

  for(i=0;i<TOTAL_NUM;i++)
    INTI_VALUE(buf[i][0], buf[i][1]);
```

这里的 for(;;)循环语句展开后为

```
  for(i=0;i<TOTAL_NUM;i++)
  buf[i][0]=1;\                  //此处反斜杠被编译器理解为接下一行
  buf[i][1]=2;
```

for(;;)循环因为 buf[i][0]=1;后面的分号而将 buf[i][1]=2 排除在循环体内。

正确的用法应为：

```
#define INTI_VALUE(x,y)\
{\
  x=0;\
  y=0;\
}

for(i=0;i<TOTAL_NUM;i++)
{
  INTI_VALUE(buf[i][0], buf[i][1]);
}
```

注：这里的反斜杠\表示下一行继续为宏定义的内容

7.将自己常用的类型重新定义，防止由于平台不同而产生的类型字节数差异，方便移植

```
typedef unsigned long   UINT32;
typedef unsigned short  UINT16;
typedef unsigned char   UINT8;
typedef signed char     INT8;
typedef signed long     INT32;
typedef signed short    INT16;
```

在程序中用后面定义的新类型名来定义变量类型，比如现在需要定义一个 8bits 的单字节无符号数，应定义为 UINT8 x。这种定义的优点直接说明了变量的数据位宽，程序员在编写程序时不容易出现数据位宽混淆的错误。如果需要将程序移植到另外的 MCU 平台上，即使编译器对数据类型的定义不一致也很容易移植，只要将这些宏定义更改成正确的类型即可。

8.使用宏进行跟踪调试

程序员在对程序调试时有时候并不想通过调试器设置断点，因为程序一旦执行到断点位置，虽然可以用调试器去查看 MCU 的工作状态，但这么做不能让程序处于连续不间断工作状态，所以断点调试和真实的程序运行状态还是存在一定差异，用宏输出调试信息就可以让程序更加接近实际工作状态。

```
#define DEBUGMSG(msg)   \
{\
#ifdef _DEBUG_\
```

```
printf(msg);\
#endif\
}
```

这样在程序中只要定义是否打开 DEBUG 调试功能的宏_DEBUG_就可以在程序中选择是否打开调试输出功能，程序调试完毕需要发放程序时关闭宏_DEBUG_所有的 DEBUGMSG()都被编译器处理成空操作。

9.宏定义里面的#和##

通常我们使用#把宏参数变为一个字符串，用##把两个宏参数贴合在一起。

```
#define STR(s)  #s
#define CONS(x,y)  int(x##f##y)

printf(STR(123));          //输出字符串"123"
printf("%d",CONS(2,6));  //2f6 输出 758
```

这类宏我比较少用，个人也建议少用，因为理解起来相对要晦涩一些。

10.一些实用的宏示例

求最大值和最小值

```
#define MAX(x,y)  (((x)>(y))?(x):(y))
#define MIN(x,y)  (((x)<(y))?(x):(y))
```

高低字节操作（可以不用&0xFF，类型为自定义类型）

```
#define HI_BYTE(n)  (UINT8)((n>>8)&0xFF)
#define LO_BYTE(n)  (UINT8)(n&0xFF)
#define BYTE2WORD(hi,lo)  (UINT16)((hi<<8)|lo)
```

高低字操作（可以不用&0xFFFF，类型为自定义类型）

```
#define HI_WORD(n)  (UINT16)((n>>16)&0x0FFFF)
#define LO_WORD(n)  (UINT16)(n&0x0FFFF)
#define WORD2DWORD(hi,lo)  (UINT32)(((UINT32)hi<<16)|lo)
```

读写指定地址上的字节

```
#define MEM_BYTE(x)  (*((UINT8 *)(x)))
```

字母大小写转换

```
#define UPCASE(c)  (((c)>='a'&&(c)<='z')?((c)-0x20):(c))
#define LOWCASE(c)  (((c)>='A'&&(c)<='A')?((c)+0x20):(c))
```

十进制和 BCD 转换

```
#define BCD_TO_DEC(bcd)  ((((UINT8)(bcd))>>4)*10+(((UINT8)(bcd))&0x0f)))
#define DEC_TO_BCD(dec)  ((((((UINT8)(dec))/10)<<4)|((UINT8)(dec))%10))
```

返回数组元素的个数

```
#define ARR_SIZE(a)  (sizeof((a))/sizeof((a[0])))
```

位操作

```
#define TEST_BIT(x,offset)  (1&((x)>>(offset)))
#define SET_BIT(x,offset)  ((x)|= (1<<(offset)))
#define CLR_BIT(x,offset)  ((x)&=(~(1<<(offset))))
```

面试常问的用宏表示一年有多少时间（留意溢出）

```
#define MINS_OF_YEAR  ((UINT32)(365*24*60))
#define SECS_OF_YEAR  ((UINT32)(365*24*60*60))
```

一组有关 8/16/32bits 数处理的宏

```
typedef union
{
    UINT16      WordCode;
    struct
    {
        UINT16  _byte0      : 8;    //LSB byte code
        UInt16  _byte1      : 8;    //MSB byte code
    } ByteCode;
}UWORD;
typedef union
{
    INT16       WordCode;
    struct
    {
```

```
        UINT16  _byte0      : 8;    //LSB byte code
        INT16   _byte1      : 8;    //MSB byte code
    }ByteCode;
}SWORD;
typedef union
{
    UINT32      LongCode;
    struct
    {
        UINT16  _word0      :16;    //LSB word code
        UINT16  _word1      :16;    //MSB word code
    }WordCode;
    struct
    {
        UINT16  _byte0      :8;      //bit 7  ~  0
        UINT16  _byte1      :8;      //bit 15 ~  8
        UINT16  _byte2      :8;      //bit 23 ~ 16
        UINT16  _byte3      :8;      //bit 31 ~ 24
    }ByteCode;
} ULONG;
typedef union
{
    INT32       LongCode;
    struct
    {
        UINT16  _word0      :16;    //LSB word code
        INT16   _word1      :16;    //MSB word code
    }WordCode;
    struct
    {
        UINT16  _byte0      :8;      //bit  7 ~  0
        UINT16  _byte1      :8;      //bit 15 ~  8
        UINT16  _byte2      :8;      //bit 23 ~ 16
        INT16   _byte3      :8;      //bit 31 ~ 24
    }ByteCode;
```

```
}SLONG;
#define BYTE0          (ByteCode._byte0)
#define BYTE1          (ByteCode._byte1)
#define BYTE2          (ByteCode._byte2)
#define BYTE3          (ByteCode._byte3)
#define WORD0          (WordCode._word0)
#define WORD1          (WordCode._word1)
#define ByteCode(x)    ((x._byte1<<8)|x._byte0)
#define WordCode(x)    ((x._byte3<<24)|(x._byte2<<16)|(x._byte1<<8)|x._byte0)
#define Byte0(x)       (x._byte0)
#define Byte1(x)       (x._byte1)
#define Byte2(x)       (x._byte2)
#define Byte3(x)       (x._byte3)
```

讲完宏再讲另外一个很有用的伪指令 register，使用 register 在提高代码执行效率方面有着强大的威力，但要注意的是一些简单 MCU 的编译器可能不支持该指令。

为什么 register 指令在提高代码执行效率方面存在着优势呢？原因需要从 regsiter 指令本身含义开始找，该指令表示变量不是在 RAM 中申请，而是使用 MCU 的通用寄存器，这样程序在处理用 register 定义的变量的时候就不需要在访问 RAM，直接读写寄存器就能完成任务，可以将代码效率提高。

使用 register 指令存在着限制条件，首先因为这种变量占用了 MCU 的通用寄存器，如果一直占用这些寄存器的话，其它代码会没有可用的通用寄存器而无法执行，所以 register 变量只能是动态局部变量和函数参数可以使用，以避免某个变量长期占用通用寄存器；其次任何 MCU 的通用寄存器个数都不多，数目有限，所以一段代码同时能支持的 register 变量也相应有限，如果申请的个数太多，编译器会将超出的 register 变量处理为普通 RAM 变量。

4.15. 手机里的计算器

在没有接触到数学的极限概念之前，不少人都会为这样的问题困扰：1/3=0.33333333…，1/3*3=1，可是另外却有 0.33333333…*3=0.99999999…，0.99999999…明明不是 1，但通过这三个等式却得出 0.99999999…等于 1 的结论，让人着实有些困惑，数学里的极限概念为我们解释了这个困惑。

这个问题好像和单片机 C 语言编程无关，实质上两者见也确实没有任何直接关系，在这里提出来只是为了引申出一个单片机 C 语言编程的问题：数据类型的处理。C 语言常用的数据类型在我看

来可以分成两类，浮点和整型，浮点就是小数，整型顾名思义就是整数。就单片机来说，许多时候都用不上浮点数，只需要整数即可满足应用要求，年长日久，使得许多单片机程序员对浮点数陌生起来，在一些特定的应用需要使用小数时，程序编写就可能不够理想。

让我们来看一个需要使用到小数的应用实例，无论是电脑里面的计算器还是街上可以买到的计算器，当你输入 1/3 之后可以得到 0.333333…的结果，如果此时你接着输入*3，显示的结果会明确无误的告诉你为 1，看来微软的工程师和专业做计算器芯片的工程师在数制处理上不存在什么问题。

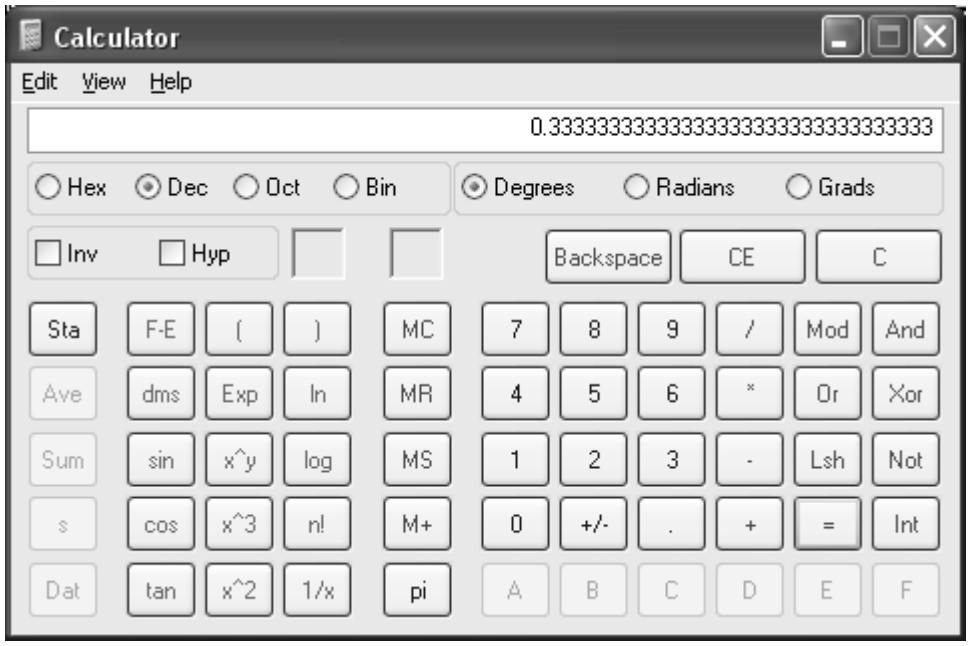

图 4.15.-1　电脑计算器图

手机实质是一个用单片机实现以通讯功能为主、内部带有其它若干附加功能的电子产品，它里面的程序自然也就是单片机程序。现在的手机基本上都带有简单的计算器功能，让我们来看看手机里面的计算器功能怎么样？结果可能会让你失望，当输入 1/3*3 之后，会看到显示的结果并不为 1，而是 0.99999999，可能这个结果在实际生活中影响不是很大，毕竟手机的计算器功能只是帮助使用者算个帐什么的，不需要太高精度，但从数学的角度看这无疑是一个错误的结果。

不要认为只是山寨的手机才有上面的问题，象在知名的 N 记、M 记手机中这个问题同样存在，我没有做过手机软件的开发，或许在设计、编写手机程序的工程技术人员眼里这也许不是一个问题，但在我看来这起码是一点不足，一个程序员应该尽可能的让自己实现的功能与已有的标准一致，除非是自己的创新。

因为自己不在手机行业，所以只能推测造成这个现象的原因，既然电脑和真正的计算器都可以做到正确显示 1，我想单片机程序同样也能够实现。不过这只是我个人的观点，如果想要别人认可我的观点，就需要更多的证据，可没有手机可以让我们自己编程来进行验证，没关系，有变通的方法，只要用 C 语言在电脑上验证算法就行。

我个人猜测手机中计算器计算结果不够精确是没有采用浮点数做为中间变量，为便于对验证过程做出解释说明，验证程序并不写成真正的计算器功能，只是对除法结果为无穷小数的情况选出几种情况将计算过程进行验证，因此在验证程序中会用浮点数和整型数做对比，计算过程显示 8 位有效数字。

程序编译环境：VC6.0 Console App32 模式

验证思路：因为 32bits 浮点数只能提供 6 位有效数字，验证过程需要提供 8 位有效数字，所以用一个整型变量存放去处小数点后的整数运算结果，另外一个整型变量存放小数点位置，另外还有一个浮点变量，直接浮点运算的结果存放在这个变量中。整数模式只利用两个整型变量，显示时通过这两个变量将运算结果的实际 8 位有效数字显示出来；浮点模式显示的时候还需要依据计算结果做一些特殊判别以确定使用整数结果还是浮点结果显示，这里没有将这部分程序加进来。

```
#include "stdafx.h"
int exp(int x,int y)//未调用 VC 库函数，该函数实现 x 的 y 次幂运算
{
    int ret;
    ret=1;
    while(y--)
    {
        ret=ret*x;
    }
    return ret;
}
int main(int argc, char* argv[])                    //验证 a/b*b 过程
{
    char string[256];                               //用来显示计算过程中的数字
    signed long result_int;                         //整数模式保存计算过程结果
    signed long radix_point;                        //整数模式保存小数点位置
    float result_float;                             //浮点模式保存计算过程结果
    int x,y,a,b;
    int i;
    a=1;                                            //验证用被除数
    b=3;                                            //验证用除数
    x=a;
    y=b;
    printf("Int mode calculate (%d/%d)*%d:\n",a,b,b);  //整数模式运算结果
```

```
printf("Step 1. Calculate %d/%d\n",a,b);          //整数模式计算 a/b 过程
result_int=0;
radix_point=0;
i=0;
while(x>=(y*(exp(10,i))))                          //计算整数结果和小数点位置
{
    i++;
}
if(i!=0)
{
    result_int=x/y;
    x=x%y;
}
for(;i<8;i++)
{
    radix_point++;
    result_int=(result_int*10)+(10*x/y);
    x=(10*x)%y;
}
for(i=0;i<256;i++)                                 //清空显示 buffer
{
    string[i]=0;
}
sprintf(string,"%d",result_int);                   //整数结果填入显示 buffer
for(i=0;i<radix_point;i++)                         //整数结果中插入小数点
{
    string[8-i]=string[7-i];
}
string[8-radix_point]='.';
printf("        %d/%d=%s\n",a,b,string);           //显示计算结果
printf("Step 2. Calculate *%d\n",b);               //整数模式计算*b 过程
i=0;
while((result_int*b)>=exp(10,i++));                //计算整数结果和小数点位置
radix_point=radix_point+8+1-i;
result_int=result_int*b;
```

```
for(i=0;i<256;i++)
{
    string[i]=0;
}
sprintf(string,"%d",result_int);                    //整数结果填入显示 buffer
for(i=0;i<radix_point;i++)                          //整数结果中插入小数点
{
    string[8-i+1]=string[8-i];
}
string[8-radix_point]='.';
for(i=9;i<256;i++)                                  //清除显示 buffer 多余的位数
{
    string[i]=0;
}
printf("        %d/%d*%d=%s\n",a,b,b,string);       //显示整数计算结果
printf("\n");

x=a;
y=b;
printf("Float mode calculate (%d/%d)*%d:\n",a,b,b);//浮点模式运算结果
result_float=(float)x/(float)y;                     //计算浮点结果
result_float=result_float*(float)y;                 //计算浮点结果
printf("Step 1. Calculate %d/%d\n",a,b);           //浮点模式计算 a/b 过程
result_int=0;
radix_point=0;
i=0;
while(x>=(y*(exp(10,i))))                           //计算整数结果和小数点位置
{
    i++;
}
if(i!=0)
{
    result_int=x/y;
    x=x%y;
}
```

```
for(;i<8;i++)
{
    radix_point++;
    result_int=(result_int*10)+(10*x/y);
    x=(10*x)%y;
}
for(i=0;i<256;i++)
{
    string[i]=0;
}
sprintf(string,"%d",result_int);
for(i=0;i<radix_point;i++)
{
    string[8-i]=string[7-i];
}
string[8-radix_point]='.';
printf("        %d/%d=%s\n",a,b,string);          //显示整数计算结果
printf("Step 2. Calculate *%d\n",b);             //浮点模式计算*b 过程
i=0;
while((result_int*b)>=exp(10,i++));              //计算整数结果和小数点位置
radix_point=radix_point+8+1-i;
result_int=result_int*b;
for(i=0;i<256;i++)
{
    string[i]=0;
}
sprintf(string,"%d",result_int);                 //整数结果填入显示 buffer
for(i=0;i<radix_point;i++)                       //整数结果中插入小数点
{
    string[8-i+1]=string[8-i];
}
string[8-radix_point]='.';
for(i=9;i<256;i++)                               //清除显示 buffer 多余的位数
{
    string[i]=0;
```

```
    }
    printf("        %d/%d*%d=%d\n",a,b,b,(int)result_float);
                                                    //显示浮点计算结果
    printf("\n");

    return 0;
}
```

运行该程序所得到的 1/3*3 运算结果：

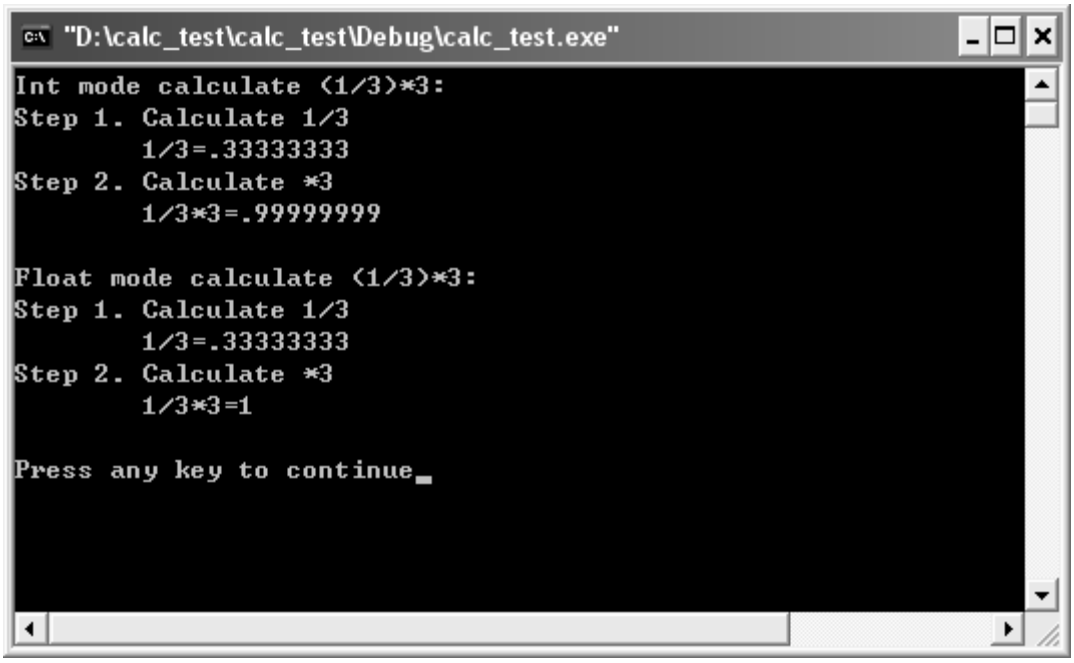

图 4.15.-2　整数模式电脑模拟结果图

运行该程序所得到的 59/23*23 运算结果：

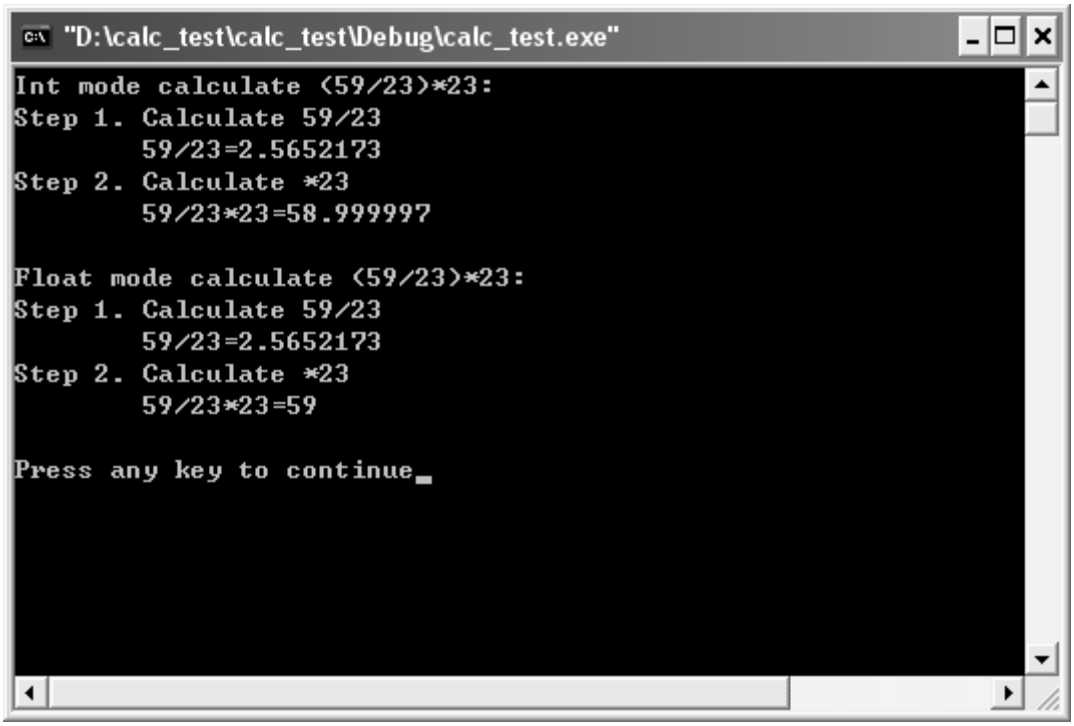

图 4.15.-3　浮点模式电脑模拟结果图

由上面的验证结果看确实可以做到与电脑中的计算器一样的效果，只是实现的程序要复杂一些，对于知名品牌，程序复杂不是一个可以认可的理由，而且手机提供计算器功能已经存在许多年，完全有时间将计算器功能做得更完善，从技术角度看这样的做法不可取。

4.16. 函数设计

想要设计出一个好的函数并没有固定的规律，主要是靠程序员自己的经验积累，基本规则是设计的函数高效、安全、完善、易懂，只要写出的函数做到这几点，就算得上是好函数。

函数一般依据功能分类整理后放在独立的程序文件中，为了让别人一眼就能知道这个文件中间的内容，需要在文件头放置文件注释信息，以方便他人阅读。

```
//-------------------------------------------------------------
//Copyright (C), 2000-2009, XXX Co., Ltd.
//Filename:     DmaDrv.c
//Description:  简要说明该程序文件完成什么功能
//Function List:
//              具体函数列表略
//Remark:       注意事项
```

```
//History:
//        1.Date:    2009/07/02
//          Author:  XXX
//          Version: 0.1
//          Modification:
//            First release version.
//        2.Date:    2009/07/24
//          Author:  XXX
//          Version: 0.2
//          Modification:
//            2.1 Remove some member of struct DmaRequest
//            2.2 Add Dma_setCEMode(),Dma_getCEMode()
//------------------------------------------------------------
```

这是我常用的注释方式，里面包含有功能描述、注意事项、作者、版本和日期等信息，这部分注释信息如果想表达得更加清晰准确可以使用中文，有的人可能习惯使用/******** ********/这样的 C 语言注释方式，我习惯在每一条需要注释的行前加双斜线//，与/******** ********/会麻烦一些，但可以避免产生/**** /**** ****/ ****/阴影部分未被屏蔽的错误。

接下来是函数的主体部分，也就是函数的具体代码，我们通过两个函数设计的实例来介绍如何进行函数设计，例子还是选用面试常问的一个题，不调用 C 的库函数实现函数 char * strcpy(char * strDest,const char * strSrc)，另外我们再实现 void* memcpy(void* dest,const void* src,int count)。

先给出面试题的“标准”答案。

```
char* strcpy(char* strDest,const char* strSrc)
{
  if((strDest==NULL)||(strSrc==NULL))
  {
    return NULL;                          //如果源地址或目的地址为空（0）返回空
  }
  char *strDestCopy=strDest;              //保留源地址指针
  while((*strDest++=*strSrc++)!='\0');    //复制直到内容为 0x00，这个 0x00 也要复制
  return strDestCopy;                     //返回源地址指针
}
```

这是标准 C 自己的实现方法，如果 PC 编程从 C 语言角度系统的看这确实是一段优秀的代码，简洁高效，但把这个函数独立出来做为面试题将前面代码当做标准答案我个人看不大妥当，尤其是追问为什么需要返回 char*这个问题后显得更为不妥。

在定制C 语言的一系列标准的时候，设计者并没有预计到今天C 语言遍地开花的局面，只是基于电脑程序的准则来进行相关设计，函数中源地址或目的地址为空时返回空就是电脑的程序是不允许对地址0 进行读写操作的，那个位置属于程序保留区域，一旦修改就会导致程序崩溃。需要返回char*的答案是方便进行链式调用，比如strlen(strcpy(buf1,buf2))，实质上就是当时设计的所有关于字符串的函数都遵循着设计者自己定义返回指的规则，如果是空指针代表出错，非空则表示操作成功，返回的指针为目的指针。

C 语言那时定义的函数是不是就完美了呢，答案是否定的。比如返回参数只能告诉程序员当前调用有没有产生错误，并不能告诉程序员具体的错误类型，加上现在C 语言的使用范围早已经远超电脑程序的领域，在单片机应用方面，程序运行不一定继续遵循地址0 不能读写的准则，有些特殊应用甚至还特意需要向这个地址进行读写，比如前面章节中提到的程序重载的功能就需要这样操作。

如果你在面试中遇到这个问题而考官继续坚持返回 NULL 和方便链式调用为标准答案，恐怕你要在考官以后在技术方面会不会成为你的良师益友方面多加斟酌。

转回正题，看看这两个函数到底怎么写为好，因为本书是以介绍单片机相关经验为主线，所以接下来这两个函数也会以适合单片机应用的方向进行设计。

假定MCU 为32bits，函数原型需要做出修改以满足我返回参数的设计。

```
UINT32 strcpy(UINT8* strDest,const UINT8* strSrc)
UINT32 memcpy(UINT8* dest,const UINT8* src,UINT32 count)
```

函数同样需要有函数说明。

```
//-------------------------------------------------------
//Name:          UINT32 strcpy(UINT8* desBuf,const UINT8* srcBuf)
//Description:   从源地址strSrc 复制数据到目的地址strDest
//               遇到内容为0x00 结束复制，0x00 自己需要被复制
//Input:         desBuf — 目的地址（从以此地址为起始位置的空间读数据）
//               srcBuf — 源地址
//Output:        desBuf — 目的地址（数据写入以此地址为起始位置的空间）
//               srcBuf — 源地址
//Return:        >0              成功复制的数据字节数
//               ERR_PARAMETER 输入参数错误
//               ERR_EMPTY_STR 复制对象为空的字符串
//Remark:        注意本函数没有溢出保护，使用时候应避免产生溢出
//History:
//         1.Date:     2009/11/12
//           Author:   XXX
```

```
//          Version: 0.1
//          Modification:
//            First release version.
//-------------------------------------------------------------
UINT32 strcpy(UINT8* desBuf,const UINT8* srcBuf)
{
  UINT32 count              //用来累计复制的数据个数
  count=0;                  //将其清 0
  if((UINT32)desBuf==(UINT32)srcBuf)
  {
    return ERR_PARAMETER;   //目的地址和源地址相同认为参数错误
  }
  if(*strSrc==0x00)
  {
    return ERR_EMPTY_STR;   //目的地址首字节为 0x00 认为是空字符串
  }
  while((*strDest++=*strSrc++)!=0x00)
  {
    count++
  }
  return count;             //返回成功复制的字节数
}
```

再来看另一个函数 memcpy()。

```
//-------------------------------------------------------------
//Name:         UINT32 memcpy(UINT8* dest,const UINT8* src,UINT32 count)
//Description:  从源地址 strSrc 复制数据到目的地址 strDest
//              复制个数由
//Input:        desBuf — 目的地址（从以此地址为起始位置的空间读数据）
//              srcBuf — 源地址
//              count  — 需要复制的数据字节数
//Output:       desBuf — 目的地址（数据写入以此地址为起始位置的空间）
//              srcBuf — 源地址
//Return:       >0            成功复制的数据字节数
//              ERR_PARAMETER 输入参数错误
```

```
//Remark:       注意本函数没有溢出保护，使用时候应避免 count 过大产生溢出
//History:
//        1.Date:    2009/11/12
//          Author:  XXX
//          Version: 0.1
//          Modification:
//            First release version.
//-------------------------------------------------------------
UINT32 memcpy(UINT8* dest,const UINT8* src,UINT32 count)
{
  UINT32 *p1,*p2;           //用于 32bits 传送
  UINT16 *p3,*p4;           //用于 16bits 传送
  UINT8  *p5,*p6;           //用于 8bits 传送
  UINT32 size;
  if(count==0)
  {
    return ERR_PARAMETER;   //需要传输的字节数为零当作参数错误
  }
  if((UINT32)desBuf==(UINT32)srcBuf)
  {
    return ERR_PARAMETER;   //目的地址和源地址相同当作参数错误
  }
  size=count;               //得到需要传送的字节数，放在此处为零时可减少执行时间
  if((((long)desBuf&0x3)==0)&&(((long)srcBuf&0x3)==0))
  {
    //32bits mode           //目的地址和源地址都满足 4 字节对齐
    p1=(UINT32 *)desBuf;    //得到目的地址 long 指针
    p2=(UINT32 *)srcBuf;    //得到源地址 long 指针
    while(size>=4)          //还有不少于 4 字节的数据需要传送就循环
    {
      *p1=*p2;              //源地址传送 4 字节到目的地址
      size-=4;              //计数器减 4
      p1++;                 //目的地址 long 指针自加，实际上是加了 4
      p2++;                 //源地址 long 指针自加，实际上是加了 4
    }
```

```
  p5=(char *)p1;          //目的地址改用 char 指针
  p6=(char *)p2;          //源地址改用 char 指针
  while(size)             //每次循环复制 1 字节到结束
  {
    *p5=*p6;
    size--;
    p5++;
    p6++;
  }
}
else if((((long)desBuf&0x1)==0)&&(((long)srcBuf&0x1)==0))
{
  //16bits mode           //目的地址和源地址都满足 2 字节对齐
  p3=(short *)desBuf;     //得到目的地址 short 指针
  p4=(short *)srcBuf;     //得到源地址 short 指针
  while(size>=2)          //还有不少于 2 字节的数据需要传送就循环
  {
    *p3=*p4;              //源地址传送 2 字节到目的地址
    size-=2;              //计数器减 2
    p1++;                 //目的地址 long 指针自加，实际上是加了 4
    p2++;                 //源地址 long 指针自加，实际上是加了 4
  }
  if(size)
  {
    (char *)p3=(char *)p4;//此时只剩余 1 字节需要复制
  }
}
else
{
  //8bits mode            //目的地址或源地址不满足 2 字节对齐
  while(size)             //每次循环复制 1 字节到结束
  {
    *desBuf=*srcBuf;
    size--;
    desBuf++;
```

```
            srcBuf++;
        }
    }
    return count;                //返回实际复制的数据字节数
}
```

和 strcpy()相比 memcpy()函数会根据源地址和目的地址的状态自动选择传送方式，当源地址和目的地址均满足 4 字节对齐时，一次复制 4 个字节，当源地址和目的地址不同时满足 4 字节对齐但满足 2 字节对齐时，一次复制 2 个字节，这样做的结果是当需要复制的字节数较多而且满足一定对齐方式时复制的效率要高许多。

为什么 memcpy()函数有做对齐判断加速处理而 strcpy()函数没有呢？因为 strcpy()函数没有告诉使用者复制长度的参数 count，需要一个字节一个字节的找 0xFF 来判断结束，这样即使加入对齐判断加速处理也不会有太明显的效果，反而会使程序变得更加复杂，所以没有必要加入这些处理。

再回过头来看看开始提出的高效、安全、完善、易懂几点要求。通过 strcpy()函数已经说明我们考虑到尽可能的让程序执行效率高一些，另外函数中的循环采用 while()而不用 for()也是让代码效率高效的方式之一，通常 while()循环比 for()循环效率要高；安全性虽然这两个例子没有直接体现出来，但在注释中强调了溢出的可能性；设计了不同的返回参数就为了让程序员在使用时候更方便，这点可以归类到完善性之中；函数中的大量注释就是为了让程序易懂。

当然不能说这里设计的两个函数例子就是最好的，在编写程序时还需要根据实际情况出发决定函数该如何设计,比如这里的memcpy()函数对于32bits的MCU加速处理是有效方法,但对一个8bits的MCU不但不能加速，反而会让效率变低。例子只是提供一个样板供大家参考，真正优质高效的函数还得要靠你们自己写出来。

函数的设计还有一点很重要，那就是函数和变量名的命名规则，最常见的是匈牙利记法，不过该记法相对比较繁琐，对于习惯汇编编程的程序员用起来会有些不习惯，另外对于一些小的单片机确实会有大材小用的感觉，所以并不建议一定按此记法来命名函数和变量。

匈牙利记法主要思想是“在变量和函数名中加入前缀以增进人们对程序的理解”，我们可以在记法的基础上根据实际情况做出一定删减或变更，定义出适合自己的命名法则，后面有自己定义的命名法则示例，这里先不细述。

4.17. 某产品函数编写规则

本文档对 V.Sxxxx Axxxxxx 的 API 函数编写规则进行约定，在编写 V.Sxxxx Axxxxxx 底层驱动函数的时候应尽量按照此文档约定进行编写。

……（详见完整版）