在设计串口驱动的过程中，要遵循的两条准则是：
1：尽量的减少程序运行的时间。
2：尽量的减少程序所占用的内存。
譬如，下面的一段程序：
程序段 1-1

```
/*指针是指向 ptr，需要发送 count 个数据*/
void  USART1WriteDataToBuffer（*ptr，u8 count）
{
  /*判断数据是否发送完毕*/
  while（count--）
  {
    /*发送数据*/
    USART1SendByte（*ptr++）;
    /*等待这个数据发送完毕，然后进入下一个数据的发送过程*/
    while(USART_GetFlagStatus(USART1,USART_FLAG_TC);
  }
  /*数据发送完毕，返回*/
}
```

很明显，这段程序在实际应用中将会产生灾难性的后果，首先，当发送数据送到发送寄存器启动发送以后，CPU 就一直在等待这个数据发送完成，然后进入下一个数据的发送，这样，直到所有要发送的数据完成，CPU 才能做其他的事情。相对于 CPU 内核运行的速度而言，串口外设的运行速度是非常快的，让一个速度非常快的设备去等待相对很慢的设备，程序的效率是非常低下的。
所以必须采用中断的方式发送数据。
程序段 1-2

```
/*将数据写入发送缓冲区*/
void  USART1WriteDataToBuffer（*ptr，u8 count）
{
  while(count!='\0')
  {
    USART1SendTCB[Index++]=*ptr++;
    Count=count;
  }
  /.....判断溢出等其他代码省略...../
}
/.....发送中断的 ISR...../
void USART1SendUpdate(void)
{
  /.....判断发送缓冲区中的数据是否发送完毕...../
  /将发送缓冲区的数据发送出去/
  USART1SendByte（*ptr++）;
  /.....发送指针加一，待发送的字节数减一等代码...../
}
```

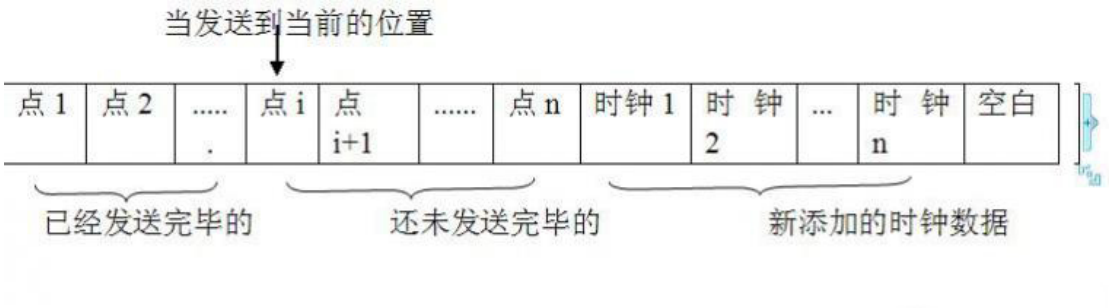
这样，当调用 USART1WriteDataToBuffer 函数将待发送的数据写入发送缓冲区以后，CPU

就可以执行其他的任务，待一个数据发送完成以后，中断 ISR 就会触发，在中断服务程序里面将下一个数据写入发送寄存器，启动下一次发送，知道完全发送完毕为止。

很明显，上述的程序的设计比较好，不用占用过多的 CPU 时间。

在实际的工程应用中，经常会出现类似这种情况：串口显示屏需要显示 1000 个点，通过串口发送这 1000 个点的颜色的 RGB 亮度值。将这 1000 个数据写入发送缓冲区以后，启动发送。在 115200 的波特率，一位起始位，一位停止位，无校验位的情况下，至少需要（10*1000*2）/115200=0.1736 秒，在这期间以内，时钟更新了，需要再发送给串口一串时间更新的数据，这个数据大约有 100 个，这样这串数据需要写入到发送缓冲区的发送字节的后面。

同样道理，在这个时候如果有显示任务更新的话，将会有其他的数据写入到发送缓冲区。

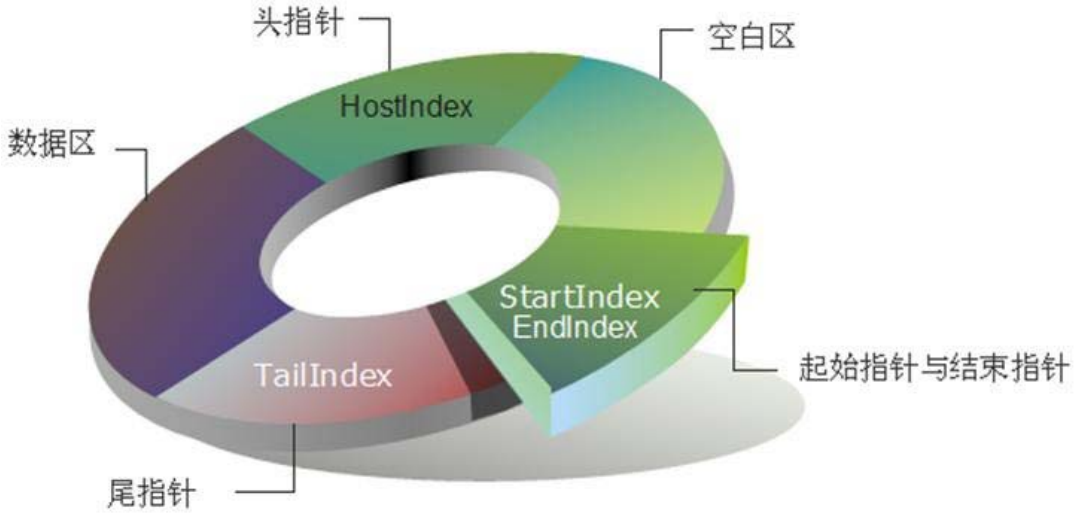


从图上可以看出，程序段 1-2 虽然满足了时间上的要求，却没有满足空间上的要求，它的数据缓冲区是单向的，这样，当发送缓冲区的所有的数据全部发送完毕后，或者当发送缓冲区撑满了以后才能将发送缓冲区内的数据清空，以便装入下次的缓冲数据。这样内存较小的嵌入式系统来说是不能容忍的。

因此，可以将发送缓冲区建立成一个环形的缓冲区，在这个环形缓冲区内，通过头指针(HostIndex)和尾指针(HostIndex)来定位空白区和数据区。

（1）：头指针(HostIndex)指向有数据区的顶部，每次写入数据，都更新头指针，如果到了缓冲区的末端(EndIndex)，就自动返回到缓冲区的起始处(StartIndex)，直到写入到尾指针处为止，这时缓冲区已经被装满，不能再装入数据。

（2）：尾指(TailIndex)针指向有数据区的尾部，当数据发送完毕后，更新尾指针的位置，如果到了缓冲区的末端(EndIndex)，就自动返回到缓冲区的起始处(StartIndex)，直到遇到头指针为止，这是证明所有的数据已经发送完毕。



这样就实现了发送缓冲区的动态调整空白区和数据区，刚刚发送完毕的数据，马上就被开辟出来用于存放下个数据，最大可能的节省了宝贵的发送缓冲区的空间，提高了使用效率。

这个程序比较复杂，大致的流程如下（省略了状态的判定，保护措施等代码）

程序段 1-3

```
/*将数据写入发送缓冲区*/
void  USART1WriteDataToBuffer（*ptr，u8 count）
{
    while(count!='\0')
   {
      /*头指针不等于尾指针，缓冲区没有撑满*/
      if(USART1HosIndext!=USART1TailIndex){
      USART1SendTCB[USART1HosIndex]=*ptr++;
      /*更新头指针，如果到了缓冲区的末端，就自动返回到缓冲区的起始处*/
      if(++USART1HosIndext>=USART1_SEND_MAX_BOX)USART1HosIndext=0;}
   }
   /......判断溢出等其他代码省略...../
}
/......发送中断的 ISR...../
void USART1SendUpdate(void)
{
   /*头指针不等于尾指针，缓冲区尚有未发生完的数据*/
   if(USART1HosIndext!=USART1TailIndex){
   /*将发送缓冲区的数据发送出去*/
   USART1SendByte（*USART1TailIndex）;
   /*更新尾指针的位置，如果到了缓冲区的末端，就自动返回到缓冲区的起始处*/
   if(++USART1TailIndex>=USART1_SEND_MAX_BOX)USART1TailIndex=0;
   /......判断溢出等其他代码省略...../
}
```

值得注意的是，一些微控制器中，例如在 Cortex-M3 的微控制器架构中，有 DMA 传送模式，可以配置一个内部的通道，将指定的地址处的数据，在无须 CPU 的管理下，直接将其发送到串口发送寄存器里去。通过这个方法，可以大大的降低了发送过程中重复进入中断的次数，从而大大提高了效率。这样，如果使用了这个芯片，就可以使用 DMA 模式进行发送。但是 DMA 发送模式下，对于头指针和尾指针就得做出一些修改，因为 DMA 传送过程中，是不能让头指针到达缓冲区终点后，自动将指针调整到起点位置的。

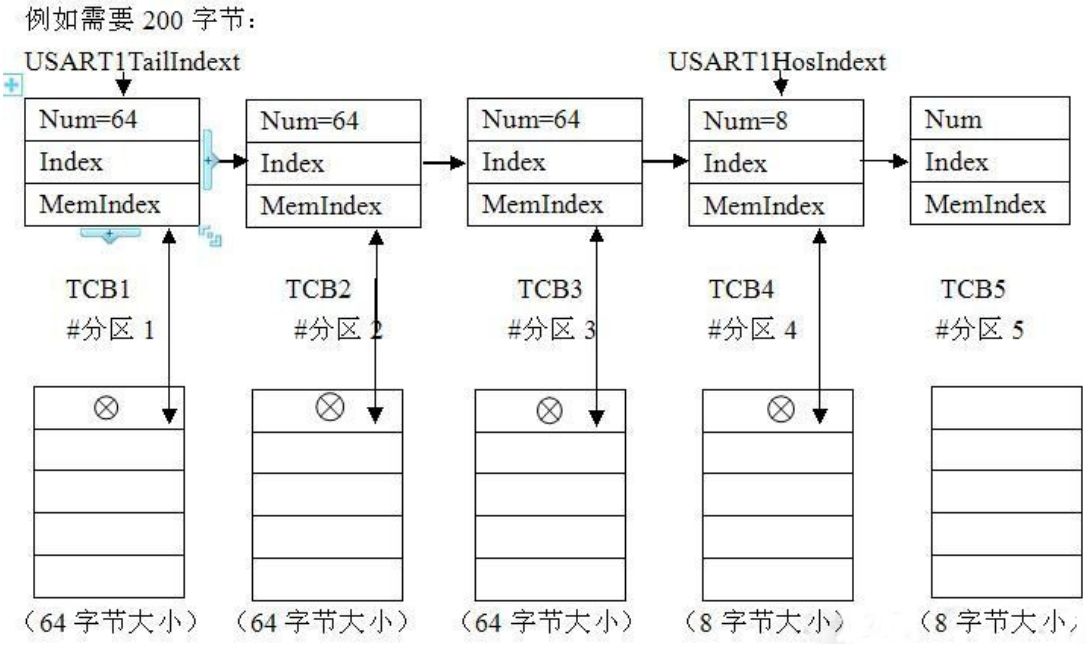
但是，加入发送管理结构体以后，上述问题可以得到解决。

利用内存块动态分配可以大大减少提高内存的使用效率，尤其是对于串口通信而言，更是如此。利用内存管理模块可以将微控制器除全局变量和静态结构变量以外的剩余的内存统一管理，在需要时候申请，在不用的时候释放，如串口的发送缓冲区，以太网，SD 卡，外部数据存储器等等均可以用内存来管理，可以重复使用，大大提高了使用效率。

首先定义发送缓冲区管理块的结构体

```
typedef struct{
unsigned char Num; //该存储区保存的有效字节数量
unsigned char *Index; //该存储区申请的内存块的指针
unsigned char *MemIndex; //该存储区申请的内存块管理区的指针
```

}USART1SendTcb;
例如需要 200 字节：



这样，加入动态内存与发送缓冲区管理块以后，无论是采用 DAM 模式发送数据还是普通的方式，都可以轻易的配置。将内存块的指针值 Index 传给 DMA 的发送地址，将待发送的字节数 Num 传给 DMA 的发送字节计数寄存器，就可以完成无须 CPU 管理的操作，最大的减少了 CPU 的使用，大大提高了内存效率。或者采用普通的中断模式。

具体的代码较长，见工程文件，不再详细列出。

内存管理

在嵌入式设备中，往往会存在一些任务需要大量的内存，在内存相对较少的微控制器中，怎样有效管理这些宝贵的资源，是必须解决的一个重要问题。

在上位机的编程中，我们通常使用 malloc（）函数以及 Free（）函数来完成对内存的管理。这是因为相对嵌入式系统而言，上位机的内存非常大，而且 Windows 提供了很好的内存管理接口，所以不存在问题。但是在嵌入式系统中，大量使用上述函数会出现两个问题:

(1)产生内存碎片的问题。

在运行的过程中，各个任务频繁的调用内存分配和释放，会导致原本一整块空间地址连续的区域分散成一堆物理地址上相互独立的区域，这样有可能导致一个程序需要一个较大的内存，空余的内存块没有一个连续的地址，无法分配给任务。久而久之，最后系统可能连一个很小的物理地址都分配不到，最后导致系统的崩溃。

如下图所示：

如下图所示：

200000	已使用	已使用	已使用	已使用	已使用	已使用	已使用		已使用
200010	已使用		已使用		已使用	已使用	已使用	已使用	已使用
200020	已使用	已使用		已使用	已使用		已使用		已使用
200030	已使用			已使用				已使用	已使用
200040	已使用	已使用		已使用				已使用	

在上图中可以看到，虽然起始地址为 20000 的内存区有 16 个空白的字节，但是仍然无法为任务分配到四个字节的物理内存。

(2)运行的时间不确定的问题

在 free（）函数中，存在着一些内存合并等功能，例如将释放完成以后，将空间上相近的两个空白区域合并为同一个，将存在内存碎片的区域重新整合，甚至可能使用了二叉树等非线性数据结构，等等操作。

而这些函数所耗费的时间是无法确定的，在实际的应用中，对于内存这种全局变量，多个任务都要用到，为避免会存在可重入性的问题，必须采用信号同步的方法，或者暂时关闭中断的方法，来同步对各个任务对共享资源的使用。这样，导致了系统死区时间的增加，响应速度的变慢，不确定性增加。

因此，在大多数嵌入式系统中，通常采用静态内存块池的方法。将系统空余的内存统一管理，生成一系列的大小固定的内存块池，在实际的操作中，以这一整个内存块进行操作。

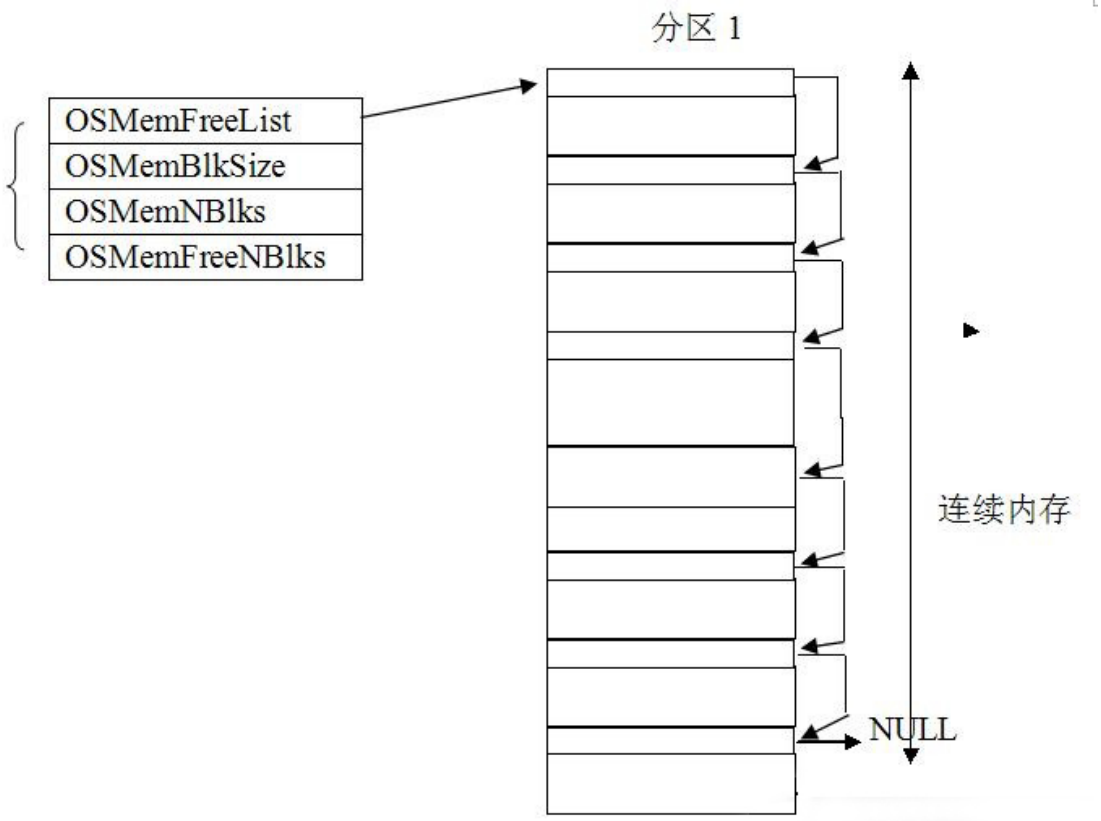
实现过程

首先定义内存管理块的结构体

```
typedef struct OSMEMTCB{
void   *OSMemFreeList;//用于指向该管理区中的空白的内存块
u8   OSMemBlkSize;//用于该管理区中的每个内存块的字节数
u8   OSMemNBlks;//用于该管理区中的分为多少个内存块
u8   OSMemFreeNBlks;//用于该管理区还剩多少空白内存块

}OSMEMTcb;
```

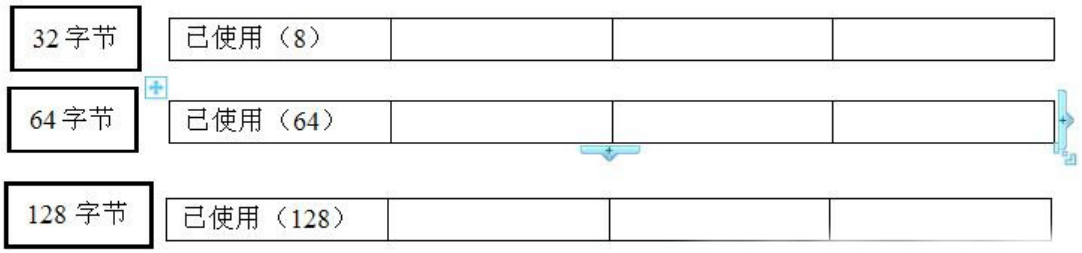
将一个静态的存储区分配给内存配置函数，内存管理块的各个列表的含义如下图所示：



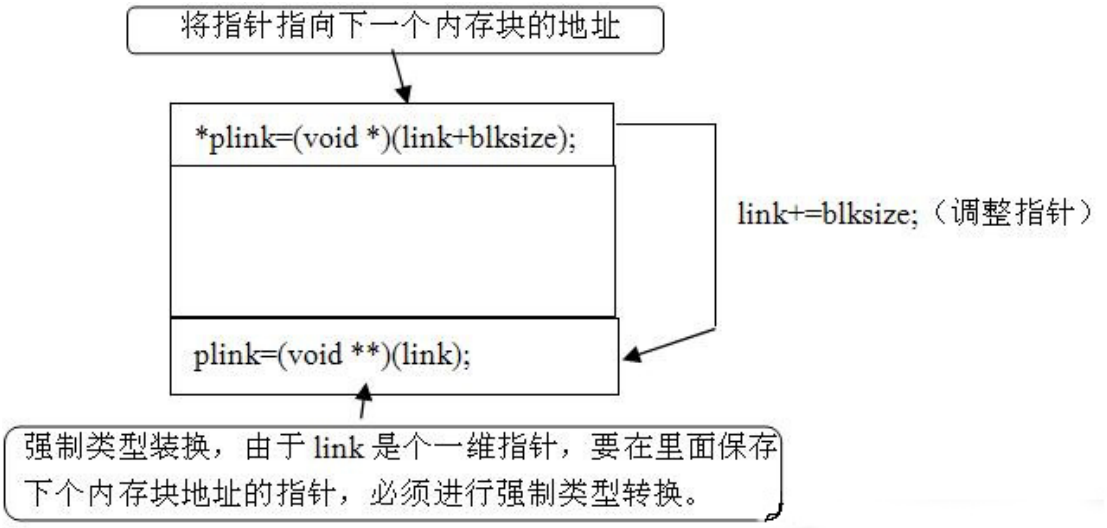
每个内存块的头四个字节用于存储下一个内存块的指针地址，直到倒数第一个为止，最后一个指针指向一个空的指针，表明已经到达内存区的的末端。

在实际运用过程中，OSMemFreeList 是指向空白的内存块的指针，通过它来申请内存，当申请到内存块以后，OSMemFreeList 指向当前数据块的下一个内存块节点地址（内存管理函数已经自动将所有内存块通过指针链接成一个单向链表），当释放内存块的时候，将 OSMemFreeList 指向当前释放的内存块，将当前内存块的下一个内存块指针指向先前的 OSMemFreeList。OSMemFreeNBlks 保存着该内存区空白块的数量，若内存块已满，返回错误代码，OSMemBlkSize 指的是每个内存块内字节数量，它的大小可以根据需要指定，理论上是它越小，内块的利用率就越高，例如保存一个 101 个字节的数据，若一个内存块的大小是 10 个字节，则需要 11 个内存块，若一个内存块的大小是 100 个字节，则需要 2 个内存块，最后一个内存块仅仅使用了一个字节。但并非内存块的越小越好，因为保存下个内存块节点的地址需要 4 个地址位，内存块越小，保存地址的数据所占比例越高。在实际操作 32 字节过程中，可以定义大小不同的内存块，灵活运用。

例 1.1：需要 200 个字节，可以如下分配：



内存配置函数的核心代码：OSMemCreate（.....）




```
for(i=0;i<nblks-1;i++)
  {
    plink=(void **)(link); //将二维指针定位到框的首位
    *plink=(void *)(link+blksize); //该内存块的地址存放的
    //是第二片内存区的首地址
    link+=blksize; //一维指针重新定位
  }
  //最后一个二维指针指向一个空指针
```

获取内存块的核心代码：OSMemGet（.....）

```
 tcb=(*ptr).OSMemFreeList;
```

```
if((*ptr).OSMemFreeNBlks==0){return (void *)0;}// 如果空白内存块的数量为 //返回,若正确返回，收到的数据应该是 0
(*ptr).OSMemFreeNBlks--; //空白内存块块数量减一
//空白内存块指针指向下一个内存区
//tcb 指向的是内存块节点指针，不能直接使用，加上偏移值 4 个字节
index=(u8 *)tcb;
index+=4;
//返回内存块指针
return index;
```

释放内存块的核心代码：OSMemDelete(.....)

```
(void **)tcb=(*ptr).OSMemFreeList; //将 OSMemFreeList 重新指向这个已经变成空白了的指针
(*ptr).OSMemFreeList=tcb; //将这个空白的指针的下个指针指向原先的空白区指针
(*ptr).OSMemFreeNBlks++; //空白内存块数量加 1
```

值得说明的是，工程文件中的 OSQMem.h 文件中

```
OS_MEM_MAX              //最多允许的内存块管理区
OS_MEM_USART1_MAX   1024   //发送缓冲区的内存大小
OS_MEM_USART1_BLK   32     //每一个块的长度
```

而 USART.h 文件中

```
DMA_MODE    //定义是采用 DMA 模式，还是普通的中断模式
```

推荐是用 DMA 模式

再就是很多朋友可能觉得奇怪的是为什么一个是

USART1.c

USART1Cinfig.c

USART1.c 是上层文件，与硬件无关，

USART1Cinfig.c 是底层文件，与硬件相关，为了方便移植，只需改变 USART1Cinfig.c 的内容就可以，我只有 STM32 的板子，

Mega16 的板子，和 340 的板子，都是我自己做的，这个程序经过移植到上述三个板子以后已经用在项目中了，在下是个菜鸟，

希望朋友们多多指教。

一直在这里学习到了很多东西，本人比较懒，老是索取而没有回报，希望能对初学的朋友们有用。

我的邮箱是 linquan315@gmail.com 欢迎朋友们多多交流。