通信方案设计

1 环形队列串口通信方式的回顾

1.1 接口函数简单的介绍

在前面《串口设备驱动接口》一章中（http://www.ourdev.cn/bbs/bbs_content.jsp?bbs_sn=4516795&bbs_page_no=1&bbs_id=3020），介绍了环形队列动态发送接收数据的好处，同时介绍了内存分配的相关内容，但是有些朋友，特别是初学者可以会比较晕，主要是不知道怎么用，以及为什么要这样设计，下文就是一些应用，我会提供很多典型微控制器的应用案例与工程源代码。但是希望朋友们知道怎么用了以后还是返回原文看看，知道原理以后，稍加修改，就可以使用到很多通讯接口和通信芯片上。

笔者也是发大宏愿，希望能在 IIC、SPI、CAN 中广泛使用这种通信方案，呵呵， 借用乔布斯的一句名言：我们来到这个世界是为了改变世界的，不然的话，我们为什么来到这里！

然而，现实总是残酷地，经过无数打击以后，我们接受了这个现状，自顶向下的腐败、社会的全面溃败、高通胀、高房价、没有出路，我们的未来昏暗而渺茫，至少，我们有共同的爱好，也许在这里我们才能找到一些心灵上的慰藉。

1.1.1 demo 工程包内的文件

我们先打开 Demo 文件夹，在这个文件夹里存放着源文件，所有的代码都在这里。

(1) OSQMem.c：内存分配的相关函数文件

(2) OSQMem.h：内存分配的配置头文件，这个文件非常重要，里面的参数直接设置运行必不可少的参数：

```
#define     OS_MEM_MAX              8           //最多允许的内存块管理区
#define     OS_MEM_USART1_MAX       1024        //发送缓冲区的内存大小
#define     OS_MEM_USART1_BLK       32          //每一个块的长度
```

其中 OS_MEM_MAX 分配内存管理区的数量，前已述及，内存管理区内内存块的大小是必须能够修改的，所以在这里预留了 8 个管理区，例程中只使用了一个，主要是为以后升级方便。对于 CAN 总线包而言，应该设置成 8 个（CAN 数据包最大为 8 个字节数据），而对于 SPI 存储设备而言，其实可以设置成 512 字节（Flash 每一页大约是占 512 字节）， 而对于 IIC 存储设备而言，有些每页 16 字节，有些 8 字节不等。

这里只使用了一个管理区，不过就如例程中演示的，不同的应用程序申请同一管理区的内存块是不会相互干扰的。

OS_MEM_USART1_MAX 指内存缓冲区的大小，这个数值取决于于应用程序需求，我这里设置的是 1024，对于 8 位机的 AVR 而言，设置应该适度减小。

OS_MEM_USART1_BLK 指每一个块的长度，大小参考《串口设备驱动接口 》一文。

(3) USART1.h：USART1.c 的头文件，包含了其所有函数的预定义。

(4) USART1.c：里面有串口发送的所有函数，用户需要设置的参数有：

```
#define USART1_SEND_MAX_Q           (OS_MEM_USART1_BLK-4)     //发送内存块内的最大空间
#define USART1_SEND_MAX_BOX         OS_MEM_USART1_MAX/OS_MEM_USART1_BLK
                                                              //发送内存块的最大数量
```

```
#define USART1_RECV_MAX_Q          32                          //接收内存块内的最大空间
```

具体的含义不再详细介绍了，这里解释一下为什么每个发送块内有用的空间是 OS_MEM_USART1_BLK-4，因为每个内存块中头四个字节已经存储了下一个链表的地址，是不能给用户使用的（对于 8 位机 3 个字节已经足够，具体依赖于硬件）。

(5) USART1ConFig.c

这个函数与底层相关，如果实例库中没有需要用户自己移植，不过很简单，和你们自己写驱动函数一样，你们只需实现简单的几个设置、发送、接收函数，就可以方便的使用这个功能比较强大的通信方案，换句话说，你们自己写底层驱动，这个过程也是必须的，何不试试呢~~保证你会有意外惊喜。

1.1.2 使用配置

以下几个参数是每次使用项目前根据需求必须配置的。

```
#define    OS_MEM_USART1_MAX          1024              //发送缓冲区的内存大小
#define    OS_MEM_USART1_BLK          32                //每一个块的长度
#define    USART1_SEND_MAX_Q          (OS_MEM_USART1_BLK-4)   //发送内存块内的最大空间
#define    USART1_SEND_MAX_BOX        OS_MEM_USART1_MAX/OS_MEM_USART1_BLK
                                                        //发送内存块的最大数量
#define    USART1_RECV_MAX_Q          32                //接收内存块内的最大空间
```

1.2 使用介绍

（1）首先建立工程，配置好路径参数什么的，然后添加移植好了的工程，编译通过以后，再加入 demo 内的除了 main.c 之外的源文件。

（2）在 OSQMem.h 中设置以下参数：

```
#define    OS_MEM_MAX                 8                 //最多允许的内存块管理区
#define    OS_MEM_USART1_MAX          1024              //发送缓冲区的内存大小
#define    OS_MEM_USART1_BLK          32                //每一个块的长度
```

这些参数与内存管理相关，具体的各位看源代码吧，这些东西不是什么高难度的东西，各位都应该能够看懂，不懂的可以给我发邮件。

（3）USART1.c：里面有串口发送的所有函数，用户需要设置的参数有：

```
#define USART1_SEND_MAX_Q          (OS_MEM_USART1_BLK-4)     //发送内存块内的最大空间
#define USART1_SEND_MAX_BOX        OS_MEM_USART1_MAX/OS_MEM_USART1_BLK
                                                             //发送内存块的最大数量
#define USART1_RECV_MAX_Q          32                        //接收内存块内的最大空间
```

到这里配置就结束了，如果编译可以通过，就可以进入下一步了。

（4）定义内存缓冲区，和一些运行相关的变量，建立内存管理区，参数的含义见注释，函数会返回一个内存管理区指针，以后申请内存全部是通过它来完成，然后配置串口等等，串口函数的参数是波特率，这个功能我还没有使用，仅仅是把 stm32 的配置加进去了，其他的微处理器我不熟，没有时间去深究，波特率还需要你们自己去查手册。

```
char MemUSART1TestBuf[OS_MEM_USART1_MAX];          //空白缓冲区地址，用于建立内存块
OSMEMTcb *OSQUSART1Index;                           //内存块管理区指针
char MemTestErr;                                    //指示错误用的，非 0 时错误

//在内存管理区注册，返回一个内存管理区指针，这个指针很重要，申请内存全部需要它完成
```

```
    //                              空白缓冲区地址，用于建立内存块
    //                               |        每一个内存块的长度
    //                               |        |            共有多少个内存块
    //                               |        |            |            错误标志
    //                               |        |            |            |
OSQUSART1Index=(OSMEMTcb       |        |            |            |
*)OSMemCreate(MemUSART1TestBuf,OS_MEM_USART1_BLK,OS_MEM_USART1_MAX/OS_MEM_USA
RT1_BLK,&MemTestErr);
//初始化串口端口，波特率为 115200，目前只针对 stm32 有用，其他的硬件需要查手册去完成配置
USART1_Configuration(115200);
```

（5）接下来就是怎么发送和接收数据了，提供了两种发送方法，一个是USART1DispFun()函数，这个函数通过指针传递参数，一遇到 0x00 就截止了：

```
/*****************************************************************************
* 文件名          : USART1DispFun
* 描述            : 检查发送缓冲区的大小，若空间足够，将待发送的数据放入到发送缓冲
                    区中去,并且启动发送,与 USART1WriteDataToBuffer 不同的是，启动发送
                    函数时不需要指定文件大小的，这就给调用提供了方便.
* 输入            : buffer 待发送的数据的指针
* 输出            : 无
* 返回            : 若正确放入到发送缓冲区中去了，就返回 0x00，否则返回 0x01
*****************************************************************************/
unsigned char USART1DispFun(unsigned char *buffer)
```

一个是 USART1WriteDataToBuffer（）函数,这个函数可以发送带 0x00 数据，但须指定待发送字节数量：

```
/*****************************************************************************
* 文件名          : USART1WriteDataToBuffer
* 描述            : 检查发送缓冲区的大小，若空间足够，将待发送的数据放入到发送缓冲
                    区中去,并且启动发送
* 输入            : buffer 待发送的数据的指针，count 待发送的数据的数量
* 输出            : 无
* 返回            : 若正确放入到发送缓冲区中去了，就返回 0x00 ，否则返回 0x01
*****************************************************************************/
unsigned char USART1WriteDataToBuffer(unsigned char *buffer,unsigned int count)
```

同样，接收数据的方法也有两种，马潮老师在《AVR 微控制器与嵌入式系统》一书中介绍了基于状态机的方法，这种方法对接收时间没有要求，只要数据传送正确就可以了。这种接收数据的方式需要指定一帧数据的大小，只需调用函数 USART1RecvData（countt，0），count 只数据帧的大小，后面代表不会启动超时中断。

```
/*****************************************************************************
* 文件名          : USART1RecvData
* 描述            : 当接收到完整的一帧数据以后的处理函数
* 输入            : count：要接收到的一帧数据数据的个数，flag：1 开启超时中断
                    0 关闭超时中断
* 输出            : 无
```

```
* 返回              : 无
*********************************************************************************/
unsigned char USART1RecvData(unsigned int count,unsigned char flag)
```

接收数据包后，在 USART1.c 文件中的 USART1RecvResetBufferIndex（）函数中处理数据，朋友们在此函数中写应用函数，但是不要把该函数中的任何数据删除，只需在函数的最后一行加上你自己的代码就可以了。其实我最初的想法是在固定的设置每一个帧的大小，但是在项目的过程中，遇到了很多串口线上设备没有固定的大小，有时需要传送一些数据流，有时是指令，这样我们怎么判定一帧已经结束了呢，就是超时中断。

这就相当于一个看门狗程序，启动串口定时定时器以后，譬如在 9600 波特率下（此时发送一个字节的数据需要 1.014ms），每隔 2ms 中断一次，如果在这 2ms 内收到了串口数据，将串口定时定时器内的计数器变为 0，重新计数，如此循环，直到最后一个字节时，不再有程序将计数器变为 0，定时器将会发生中断，这时一帧数据就结束了。可以对该命令或者数据进行处理了。

同样，接收数据包后，在 USART1.c 文件中的 USART1RecvResetBufferIndex（）函数中处理数据，

（6）申请和释放内存函数 OSMemGet（）、OSMemDelete（）。

```
/*********************************************************************************
* 文件名            : OSMemGet
* 描述              : 从一个内存管理区获取一个内存块
* 移植步骤          : 无
* 输入              : ptr 内存管理区的指针
* 输出              : 无
* 返回              : 获取的空白内存块的首地址
*********************************************************************************/
u8 *OSMemGet(OSMEMTcb *ptr,u8 *err)
/*********************************************************************************
* 文件名            : OSMemDelete
* 描述              : 从一个内存管理区删除一个内存块
* 移植步骤          : 无
* 输入              : ptr 内存管理区的指针,index,申请到的内存块的指针
* 输出              : 无
* 返回              : 如果要删除的内存块是一个空指针，则返回 0xff，若能够删除，返回 0
*********************************************************************************/
u8 OSMemDelete(OSMEMTcb *ptr,u8 *index)
```

2 通信数据块结构与内存管理模块

2.1 为何要在缓冲区中建立通信数据块，以及加入内存管理模块

在嵌入式系统中，很多外围设备的速度是比较慢的，譬如串口和 IIC 设备，这些设备的输出输入的速度大约在数十微秒到数百微秒。在较复杂的应用中，例如在笔者的一个控制器的项目中，开机后主设备必须检测从设备的枚举情况，其流程图如下：

通讯使用的是 232 总线，经过我自己画的一块小板来完成 232—CAN 的转换。在网络中存在着 60 个从设备，对应不同的数据帧（具体的指令不详述），每

一个设备的枚举要通过发送一帧指令，经过 CAN 转换，到达从设备，从设备响应以后返回给主设备。在这里计算一下，如果是在 9600 波特率的情况下，每个帧发送大约需要 10ms 的时间，加上返回的时间大约是 20ms，那么 60 个设备大约需要 1200ms，如果不使用操作系统，同时使用传统的等待方式发送，这 1200ms 微控制器什么也做不了，这在学校、研究所、“教授”、“专家”们那里糊弄糊弄还行，到了社会上，就不能这么干了。

以下是我使用环形队列发送一个欢迎界面的（一共 374 个字符）所耗费的时间，使用 MDK4.02 的性能分析器进行分析：可见在发送的大部分时间中，CPU 实际上都是在睡大觉（35ms 在 DelayMs 函数中），只有大约不到 2ms 的时间是在做事情，如果加上笔者曾经所写的合作式操作系统的方式（见笔者的另外一篇帖子 http://www.ourdev.cn/bbs/bbs_content.jsp?bbs_sn=3719375&bbs_page_no=1&search_mode=3&search_text=linquan315&bbs_id=9999 ），CPU 同时还可以做很多其他的事情。

当然使用 RT-thread 或者 UCOS 那就更加锦上添花了。

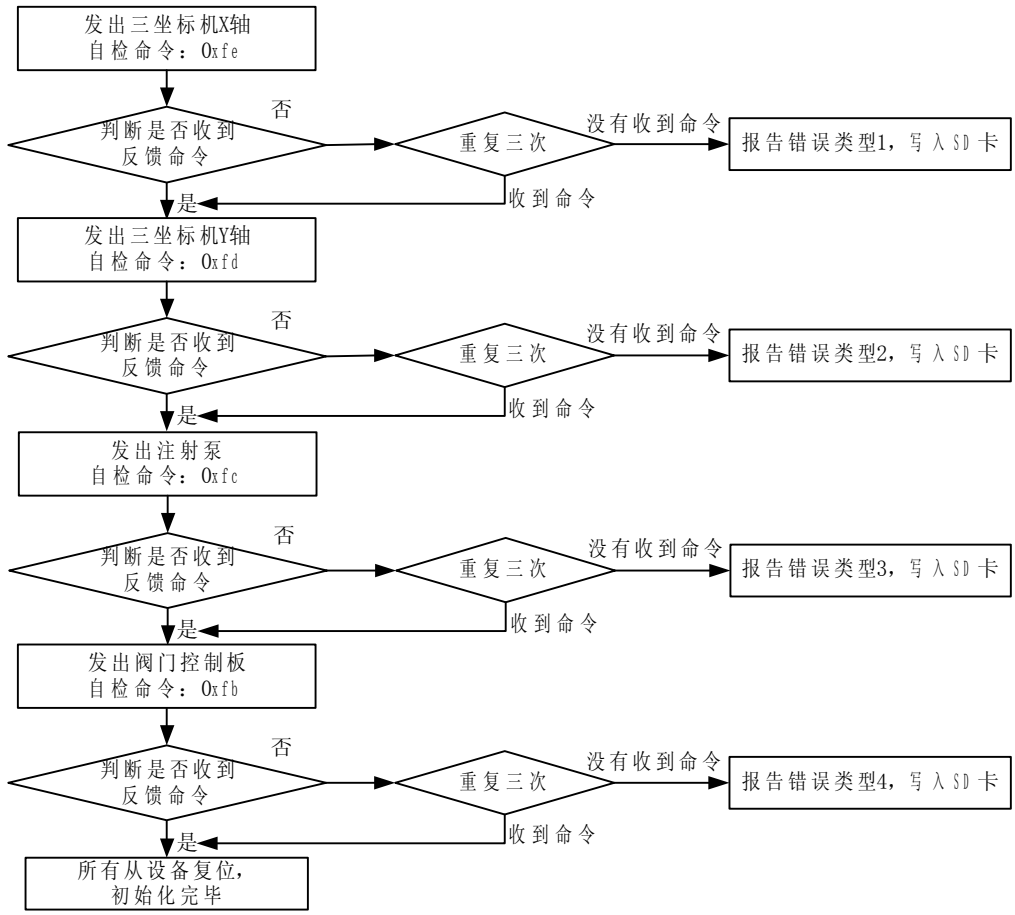


图 2-1 主控制器枚举从设备

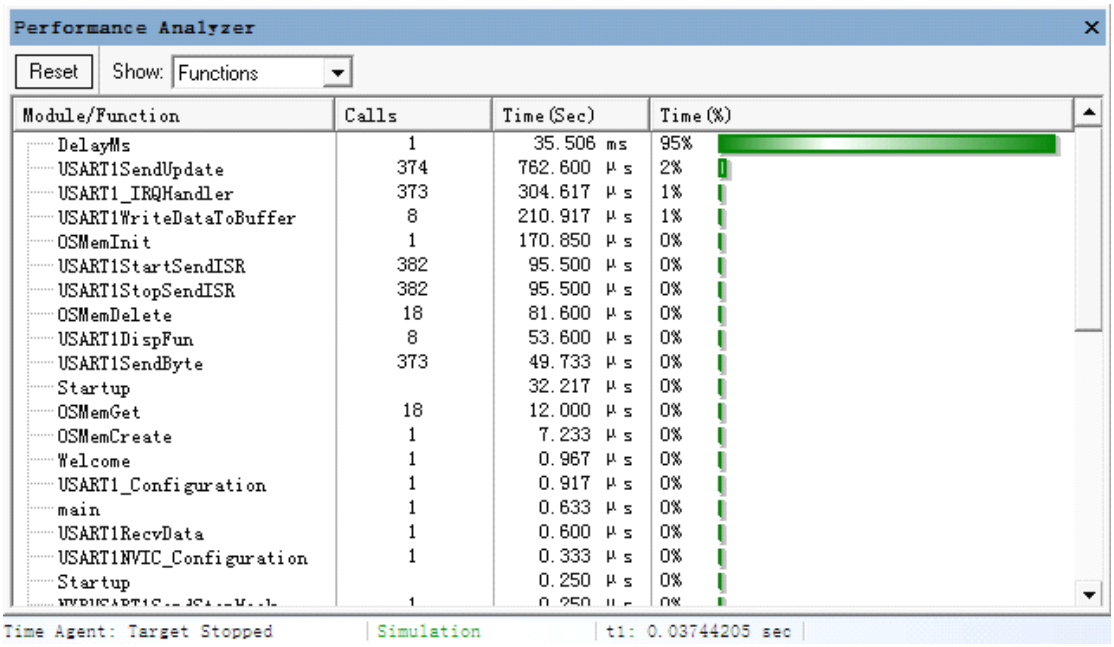


图 2-2 使用 MDK4.02 的性能分析器分析各个函数的使用情况

2.2 内存管理模块的申请、释放内存测试

测试方法如下，

（1）首先定义相关变量，建立内存管理区，各个参数的定义见注释。

```
char MemUSART1TestBuf[OS_MEM_USART1_MAX];          //空白缓冲区地址，用于建立内存块
OSMEMTcb *OSQUSART1Index;                          //内存块管理区指针
char *MemTestIndex[10];                            //分配到了的内存块的指针，测试用的
                                                   //在这里观察分配的内存块的地址，若
                                                   //是出现了异常，证明出现了错误
char MemTestErr;                                   //指示错误用的，非 0 时错误
```

值得注意的是为了便于观察内存分配情况，在这里定义了一个指针数组*MemTestIndex[10]，用于观察分配的内存块的地址，若是出现了异常，证明出现了错误。

```
//在内存管理区注册，返回一个内存管理区指针，这个指针很重要，申请内存全部需要它完成
    //                          空白缓冲区地址，用于建立内存块
    //                           |        每一个内存块的长度
    //                           |        |          共有多少个内存块
    //                           |        |          |          错误标志
    //                           |        |          |          |
OSQUSART1Index=(OSMEMTcb         |        |          |          |
*)OSMemCreate(MemUSART1TestBuf,OS_MEM_USART1_BLK,OS_MEM_USART1_MAX/OS_MEM_USA
RT1_BLK,&MemTestErr);
    //初始化串口端口，波特率为 115200，目前只针对 stm32 有用，其他的硬件需要查手册去完成配置
    USART1_Configuration(115200);
    //显示欢迎界面
    Welcome();
    DelayMs(100);
```

（2）在超级循环中不断申请、释放内存，同时启动串口发送。由于串口发送的同时也在不停的申请和释放内存块，两者交替进行，为了最大层度的仿真实际情况，申请和释放内存的顺序被人为的打乱，不同的应用程序同时申请一个管理区的内存，经过测试，没有发现异常。

```
While(1)
{
    //测试该内存块还可以他用，除了发送串口，还可以发送 CAN 数据包，IIC、SPI 等等
    for(i=0;i<10;i++)
    {
        //申请内存
        MemTestIndex[i]=(u8 *)OSMemGet(OSQUSART1Index,&MemTestErr);
    }
    //发送数据
    USART1WriteDataToBuffer(&count,1);count++;
    DelayMs(2);
    //测试该内存块还可以他用，除了发送串口，还可以发送 CAN 数据包，IIC、SPI 等等
    //此处故意打乱顺序，测试能否通过
    for(i=10;i>0;i--)
    {
        //释放内存
        OSMemDelete(OSQUSART1Index,MemTestIndex[i-1]);
    }
}
```

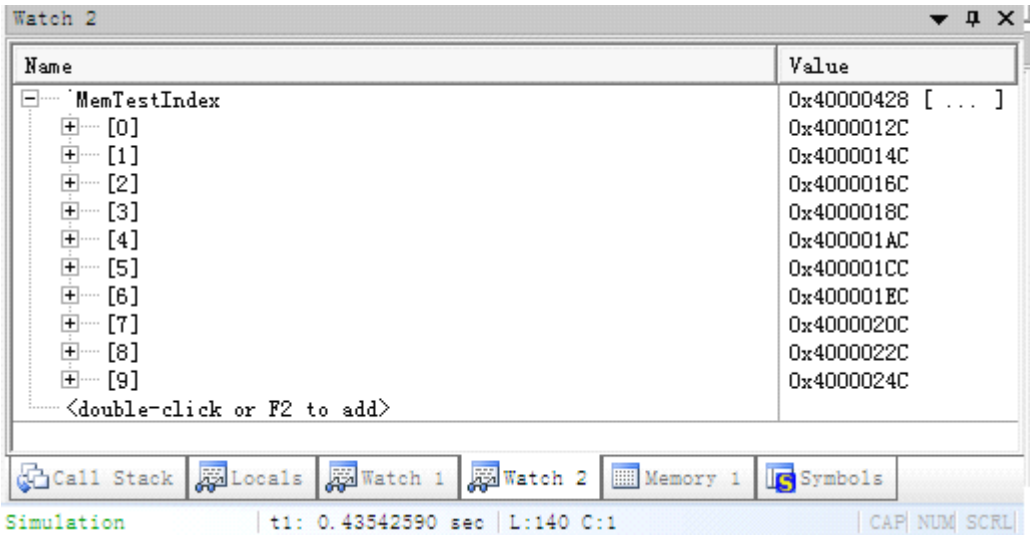


图 2-3 内存块测试

3 移植要点以及范例

3.1 移植步骤与注意事项

移植其实很简单，当然，前提是你对这个器件有足够的了解，我发觉这个过程其实比较痛苦，我没有用过 AT91SRAM64、LPC2148、LM3S1138 等芯片，但

是想将其移植，苦于没有代码库或者例程，要完整的搭建一个工程其实是比较困难的，至少需要一段时间，但是我没有足够的时间。仅仅完成了 LPC 的移植，LPC 的寄存器比较简单。或者说大家都是极其自私的人，总是盼望别人来拯救自己（笔者不喜欢那些经常在网上求救的帖子，相信很多人也不喜欢，每个人遇到的问题千差万别，需要自己去解决，工作了以后，有问题谁鸟你），自己学了点东西就捂着，别人做出了东西就希望人家免费的教你，还把原理图、源代码、PCB 都供上。我在网上找了许久都没有找到合适的简单的串口收发例程，使我十分生气。

这些工作后来我也不想移植了，如果朋友们有简单的串口、IIC、SPI、CAN 的例程包及仿真软件，不妨发给我，我帮你们移植，不然看一堆数据手册和编译环境从头到尾，我没那个耐心（ST 的除外，我对 ST 非常熟悉）。

废话少说，首先需要的外设是：

1、通讯接口，可以是串口（232 或者 485），SPI、IIC、CAN，以太网和 USB 我不熟悉。我们需要提供配置代码、接收及发送代码、中断接口代码。

2、一个通用定时器，用来检测帧超时。我们需要配置代码、开始计数代码、结束计数代码、清零计数器代码。

3.1.1 通讯接口的移植

如图 3-1 所示，需要配置如下函数。

函数名	功能
USART 1PinConfiguration	配置 USART 1对应Tx、Rx的管脚
USART 1NVIC _Configuration	USART 1中断通道配置
USART 1_Configuration	配置USART 1
USART 1SendByte	USART 1发送函数
USART 1RecvByte	返回串口接收到了的数据
USART 1_IRQHandler	USART 1_IRQHandler （USART 1发送）中断函数通道
TIM 2_Configuration	配置检测通信帧超时的定时器
TIM 2NVIC _Configuration	配置检测通信帧超时的定时器的中断配置
TIM 2_IRQHandler	配置检测通信帧超时的定时器的中断通道
USART 1ClearCounter	清零定时器的值
TIM 2StartCounter	定时器开始计时
TIM 2StopCounter	定时器停止计时

图 3-1 需要移植的函数

在 USAR1Config.c 中，其他的函数不是非常重要，仅仅供以后的程序升级之用。

3.2 在 51 内核系列的 C8051F020 上的移植范例与 MKD 仿真

以下开始介绍C8051F020的移植范例。在C8051F020中使用的是传统的8051的内核，我在移植时发觉当加入内存分配的时候，就会出现程序跑飞的情况，经过仿真后，在内存释放的时候总是出现程序跳转异常，感觉是 51 的寄存器不适合比较复杂的指针运算以及函数嵌套，在将优化登记调至最低后，还是出现问题，笔者不得不放弃了内存分配的方法，与此同时将函数的块发送也取消了。

（1）USART1PinConfiguration 函数

```
void USART1PinConfiguration(void)
{//交叉开关配置
    XBR0 = 0x04;
    XBR1 = 0x00;
```

```
    XBR2 = 0x40;
    // P0  口分配状况
    // P0.0 = UART0 TX
    // P0.1 = UART0 RX
    //输出方式
    P74OUT = 0x08;
}
```

（2）USART1NVIC_Configuration 函数，由于 C8051f020 不需通道配置，这里仅仅简单的打开 USART 中断允许寄存器。有些微处理器没有发送完毕中断使能，而仅仅只有发送寄存器空的中断（特别是 SPI 通信中，很多处理器没有提供发送数据完成的中断），这时情况有些变化，不能在这里就使能了该中断，需要在发送开始的时候再使能，发送结束以后关闭这个中断。

```
 void USART1NVIC_Configuration(void)
{
    IE|=0x90;
}
```

（3）USART1_Configuration 函数，配置波特率，发送接收模式等函数。必须使能发送和接收，

```
USART1_Configuration
{
    USART1PinConfiguration();
    TMOD &= 0x0f;//选择 T1 工作模式
    TMOD |= 0x20;
    SCON0 = 0x50;
    TH1 = 256 - 48000000 / 9600 / 32 / 12;
    TR1 = 1;
    USART1NVIC_Configuration();
}
```

（4）USART1 发送函数

```
void USART1SendByte(unsigned char temp)
{
    SBUF0=temp;
}
```

（5）USART 的接收函数

```
unsigned int USART1RecvByte(void)
{
    return SBUF0;
}
```

（6）USART 的中断函数，在有些微处理器中，发送后中断通道与接收的通道不同，有些在同一个通道中，但是，这个函数的主要作用是，如果是发送中断，就必须调用 USARTSendUpdate 函数，如果是接受中断，就必须调用 USARTRecvUpdate 函数，至于清除中断标志等等代码，诸位应该知道，不再累述了。

```
void USART1_IRQHandler(void) interrupt 4 using 1
```

```
{
    if(TI0)
    {
        USART1SendUpdate();
        TI0=0;
    }
    else if(RI0)
    {
        USART1RecvUpdate();
        RI0=0;
    }
}
```

（7）TIM2_Configuration 函数，用于产生超时中断，配置的时候要注意，定时器的时间是不能定的太长，也不能太短，大约是接收一个字符的 2 倍时间左右，至于定时器怎么配置和怎么分频各位自己去看手册，这里不再累述。

```
void TIM2_Configuration(void)
{
    TMOD|=0x01;                         //定时器 016 位模式
    CKCON|= 0x08;                 //定时器 0 使用系统时钟
    TIM2NVIC_Configuration();
}
```

（8）TIM2NVIC_Configuration 函数，用于配置中断。

```
void TIM2_IRQHandler(void)       interrupt 1
{
    ET0|=0x02;                          //允许 TIM0 中断
}
```

（9）TIM2_IRQHandler 配置检测通信帧超时的定时器的中断通道，此时需要关闭定时器，以及调用 USART1RecvResetBufferIndex 函数，这个函数是收到一帧数据以后的处理函数。上次一些朋友说不知道怎么接收数据，这里就是。

```
void TIM2_IRQHandler(void)       interrupt 1
{
    TCON&=~0x20;
    USART1RecvResetBufferIndex();
}
```

（10）USART1ClearCounter函数，清零定时器的值

```
void USART1ClearCounter(void)
{
    TL0=0;
    TH0=0;
}
```

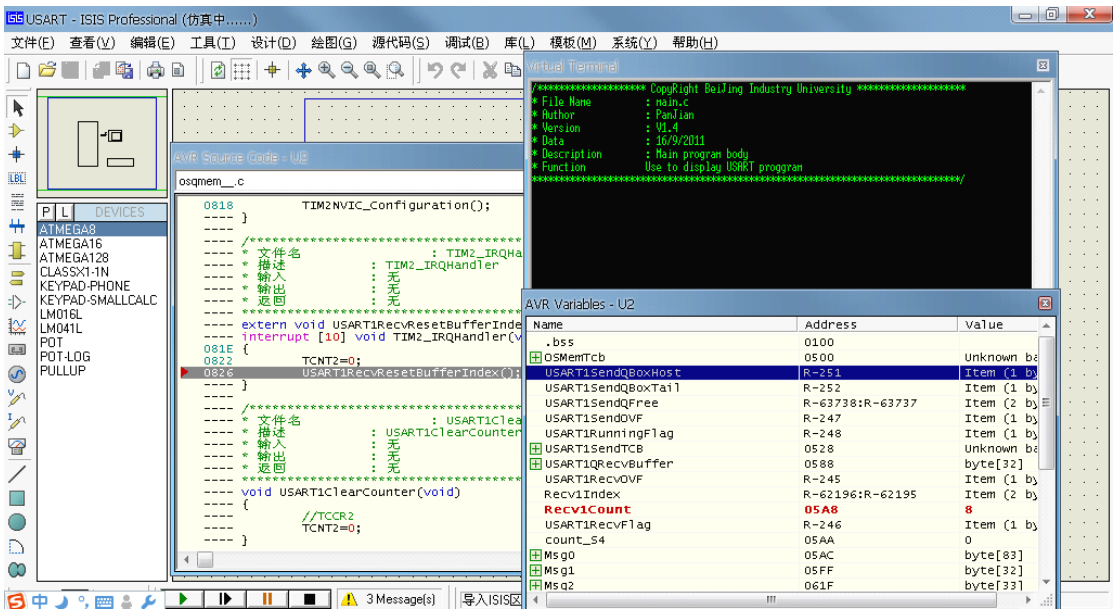
（11）USART1StartCounter函数，定时器开始计时

```
void USART1StartCounter(void)
{
    TCON|=0x10;                         //打开 TIM0
```

}

（12）USART1StartCounter函数，定时器开停止始计时

```
void USART1StopCounter(void)
{
    TCON&=~0x10;                    //关闭 TIM0
}
```



3.3 在 **8** 位 **AVR** 系列的 **Mega16**、**Mega128** 上的移植范例与 **Proteus** 仿真

3.3 在 **ARM7TDMI** 内核的 **NXP** 系列的 **LPC2148** 上的移植范例与 **MKD** 仿真

3.4 在 **Cortex-M3** 内核的 **STM32F** 系列的 **STM32F103ZET6** 上的移植范例

4 代码性能分析

4.1 使用 **MDK4.02** 的性能分析器进行分析

4.2 死区时间测试与可重入性分析

5 移植到 SPI、IIC、CAN 接口上

5.1 移植到 **STM32F103ZET6** 上，驱动 **SPI** 存储器芯片 **AT45DB161** 上

5.2 移植到 **STM32F103ZET6** 上，使用 **IIC** 总线，软件仿真

5.3 移植到 **STM32F103ZET6** 上，使用 **CAN** 总线，软件仿真