# [串口流控制（flow control）](http://blog.csdn.net/songshu5555/article/details/8715695)

分类： [网络通讯](http://blog.csdn.net/songshu5555/article/category/1365871)2013-03-25 10:15 250人阅读 [评论](http://blog.csdn.net/songshu5555/article/details/8715695#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/songshu5555/article/details/8715695#report)

我们在串行通讯处理中,常常看到RTS/CTS和XON/XOFF这两个选项,这就是两个流控制的选项,目前流控制主要应用于调制解调器的数据通讯中,但对普通RS232编程,了解一点这方面的知识是有好处的。那么,流控制在串行通讯中有何作用,在编制串行通讯程序怎样应用呢？这里我们就来谈谈这个问题。

  **1.流控制在串行通讯中的作用**  
   这里讲到的“流”,当然指的是数据流。数据在两个串口之间传输时,常常会出现丢失数据的现象,或者两台计算机的处理速度不同,如台式机与单片机之间的通讯,接收端数据缓冲区已满,则此时继续发送来的数据就会丢失。现在我们在网络上通过MODEM进行数据传输,这个问题就尤为突出。流控制能解决这个问题,当接收端数据处理不过来时,就发出“不再接收”的信号,发送端就停止发送,直到收到“可以继续发送”的信号再发送数据。因此流控制可以控制数据传输的进程,防止数据的丢失。  PC机中常用的两种流控制是硬件流控制（包括RTS/CTS、DTR/CTS等）和软件流控制XON/XOFF（继续/停止）,下面分别说明。

  **2.硬件流控制**  
    硬件流控制常用的有RTS/CTS流控制和DTR/DSR（数据终端就绪/数据设置就绪）流控制。  
    硬件流控制必须将相应的电缆线连上,用RTS/CTS（请求发送/清除发送）流控制时,应将通讯两端的RTS、CTS线对应相连,数据终端设备（如计算机）使用RTS来起始调制解调器或其它数据通讯设备的数据流,而数据通讯设备（如调制解调器）则用CTS来起动和暂停来自计算机的数据流。这种硬件握手方式的过程为：我们在编程时根据接收端缓冲区大小设置一个高位标志（可为缓冲区大小的75％）和一个低位标志（可为缓冲区大小的25％）,当缓冲区内数据量达到高位时,我们在接收端将CTS线置低电平（送逻辑0）,当发送端的程序检测到CTS为低后,就停止发送数据,直到接收端缓冲区的数据量低于低位而将CTS置高电平。RTS则用来标明接收设备有没有准备好接收数据。

  PC端处理：  
      发. 当 发现（不一定及时发现） CTS (-3v to -15v)无效时，停止发送,  
          当 发现（不一定及时发现） CTS (3v to 15v)有效时，恢复发送；

      收. 0<M<N<LEN\_OF\_RX\_BUFFERS  
          当接收buffers中的bytes<M 时，给 RTS 有效信号（+3v to +15v),  
          当接收buffers中的bytes>N 时，给 RTS 无效信号（-3v to -15v);

    常用的流控制还有还有DTR/DSR（数据终端就绪/数据设置就绪）。我们在此不再详述。由于流控制的多样性,我个人认为,当软件里用了流控制时,应做详细的说明,如何接线,如何应用。

    **3.软件流控制**  
    由于电缆线的限制,我们在普通的控制通讯中一般不用硬件流控制,而用软件流控制。一般通过XON/XOFF来实现软件流控制。常用方法是：当接收端的输入缓冲区内数据量超过设定的高位时,就向数据发送端发出XOFF字符（十进制的19或Control-S,设备编程说明书应该有详细阐述）,发送端收到XOFF字符后就立即停止发送数据;当接收端的输入缓冲区内数据量低于设定的低位时,就向数据发送端发出XON字符（十进制的17或Control-Q）,发送端收到XON字符后就立即开始发送数据。一般可以从设备配套源程序中找到发送的是什么字符。  
    应该注意,若传输的是二进制数据,标志字符也有可能在数据流中出现而引起误操作,这是软件流控制的缺陷,而硬件流控制不会有这个问题。  
  
    顺便说明一下,有不少朋友问到,为什么不在我编写的软件串口调试助手中将流控制加进去,我最初将这个调试工具定位在各种自动控制的串口程序调试上,经过计算和实验验证,在设置的特定采样周期内可以完成通讯任务,就干脆不用流控制。而且在工控中您即使不懂流控制,也能编写出简单的串口通讯程序来,就如我写的串口调试助手

[串口流控制DCB结构体解析及设置](http://blog.csdn.net/songshu5555/article/details/8742652)

分类： [网络通讯](http://blog.csdn.net/songshu5555/article/category/1365871)2013-03-31 13:31 521人阅读 [评论](http://blog.csdn.net/songshu5555/article/details/8742652#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/songshu5555/article/details/8742652#report)

一、串口通信结构体意义解析：

typedef struct \_DCB

{ DWORD DCBlength;

DWORD BaudRate; //波特率

DWORD fBinary :1;

DWORD fParity :1; //是否奇偶校验

DWORD fOutxCtsFlow :1; // CTS output flow control 指定CTS是否用于检测发送控制。当为TRUE时CTS为OFF，发送将被挂起。（发送清除）

DWORD fOutxDsrFlow :1; // DSR output flow control 指定DSR是否用于检测发送控制。（数据装备好） 当为TRUE是DSR为OFF，发送将被挂起。

DWORD fDtrControl :2; // DTR flow control type   
//DTR\_CONTROL\_DISABLE值将DTR置为OFF,

//DTR\_CONTROL\_ENABLE值将DTR置为ON,

//DTR\_CONTROL\_HANDSHAKE 允许DTR"握手",

DWORD fDsrSensitivity :1; //若为TRUE，通讯驱动程序对DSR信号状态敏感。驱动程序将忽略任何接收的字节数，除非DSR调制解调器的输入线为高。

DWORD fTXContinueOnXoff :1; //为TRUE，输入缓冲区内字节已经满XoffLim及驱动程序已经发送XoffChar停止接收字节时，仍然继续发送。为FALSE，输入缓冲区内XonLim是空的，及驱动程序已经发送XonChar字符恢复接收的字节传输后，才会继续接收。

DWORD fOutX :1; //发送方的行为定义，为TRUE时，接收到XoffChar之后便停止发送，接收到XonChar之后将重新开始发送；

DWORD fInX :1;  //接收方的行为定义，为TRUE时，接收缓冲区接收到代表缓冲区满的XoffLim之后，XoffChar发送出去；接收缓冲区空的Buffer达到XonLim之后，XonChar发送出去。

DWORD fErrorChar :1;

DWORD fNull :1;

DWORD fRtsControl :2; // RTS Control Flow

//RTS\_CONTROL\_DISABLE时,RTS置为OFF  
//RTS\_CONTROL\_ENABLE时, RTS置为ON  
//RTS\_CONTROL\_HANDSHAKE时,  
//当接收缓冲区小于半满时RTS为ON  
//当接收缓冲区超过四分之三满时RTS为OFF  
//RTS\_CONTROL\_TOGGLE时,  
//当接收缓冲区仍有剩余字节时RTS为ON ,否则缺省为OFF

DWORD fAbortOnError :1; // abort reads/writes on error，为TRUE时,有错误发生时中止读和写操作

DWORD fDummy2 :17;

WORD wReserved;

WORD XonLim; //指定在XON字符发送之前接收缓冲区中空缓冲区可允许的最小字节数

WORD XoffLim; //指定在XOFF字符发送这前接收缓冲区中数据缓冲可允许的最小字节数

BYTE ByteSize;

BYTE Parity; //奇偶校验方式

BYTE StopBits; //停止位

char XonChar;  //请求发送方继续发送时的字符 0x11

char XoffChar; //请求发送方停止发送时的字符 0x13

char ErrorChar;

char EofChar;

char EvtChar;

WORD wReserved1;

} DCB, \*LPDCB;

二、设置流控制属性：

           dcb.fDsrSensitivity = FALSE;

           dcb.fTXContinueOnXoff = FALSE;

            dcb.fRtsControl = RTS\_CONTROL\_DISABLE;  
            dcb.fDtrControl = DTR\_CONTROL\_ENABLE;  
  
            switch (g\_lpInst->flowControl)  
            {  
                case NoFlowControl:  
                {  
                    dcb.fOutxCtsFlow = FALSE;  
                    dcb.fOutxDsrFlow = FALSE;  
                    dcb.fOutX = FALSE;  
                    dcb.fInX = FALSE;  
                    break;  
                }  
                case CtsRtsFlowControl:  
                {  
                    dcb.fOutxCtsFlow = TRUE;  
                    dcb.fOutxDsrFlow = FALSE;  
                    dcb.fRtsControl = RTS\_CONTROL\_HANDSHAKE;  
                    dcb.fOutX = FALSE;  
                    dcb.fInX = FALSE;  
                    break;  
                }  
                case CtsDtrFlowControl:  
                {  
                    dcb.fOutxCtsFlow = TRUE;  
                    dcb.fOutxDsrFlow = FALSE;  
                    dcb.fDtrControl = DTR\_CONTROL\_HANDSHAKE;  
                    dcb.fOutX = FALSE;  
                    dcb.fInX = FALSE;  
                    break;  
                }  
                case DsrRtsFlowControl:  
                {  
                    dcb.fOutxCtsFlow = FALSE;  
                    dcb.fOutxDsrFlow = TRUE;  
                    dcb.fRtsControl = RTS\_CONTROL\_HANDSHAKE;  
                    dcb.fOutX = FALSE;  
                    dcb.fInX = FALSE;  
                    break;  
                }  
                case DsrDtrFlowControl:  
                {  
                    dcb.fOutxCtsFlow = FALSE;  
                    dcb.fOutxDsrFlow = TRUE;  
                    dcb.fDtrControl = DTR\_CONTROL\_HANDSHAKE;  
                    dcb.fOutX = FALSE;  
                    dcb.fInX = FALSE;  
                    break;  
                }  
                case XonXoffFlowControl:  
                {  
                    dcb.fOutxCtsFlow = FALSE;  
                    dcb.fOutxDsrFlow = FALSE;  
                    dcb.fOutX = TRUE;  
                    dcb.fInX = TRUE;  
                    dcb.XonChar = 0x11;  
                    dcb.XoffChar = 0x13;  
                    dcb.XoffLim = 100;  
                    dcb.XonLim = 100;  
                    break;  
                }

Win32

下串口通信与

16

位串口通信有很大的区别。在

Win32

下，可以使用两种编程方式

实现串口通信，

其一是调用的

Windows

的

API

函数，

其二是使用

ActiveX

控件。

使用

API

调用，

可以清楚地掌握串口通信的机制，熟悉各种配置和自由灵活采用不同的流控进行串口通信。下

面介绍串口操作的基本知识。

打开串口：使用

CreateFile()

函数，可以打开串口。有两种方法可以打开串口，一种是同

步方式（

NonOverlapped

）

,

另外一种异步方式（

Overlapped

）。使用

Overlapped

打开时，适

当的方法是：

  HANDLE hComm;

  hComm = CreateFile( gszPort,

  GENERIC\_READ | GENERIC\_WRITE,

  0,

  0,

  OPEN\_EXISTING,

  FILE\_FLAG\_OVERLAPPED,

  0);

  if (hComm == INVALID\_HANDLE\_VALUE)

  // error opening port; abort

配置串口：

1.DCB

配置

 DCB

（

Device Control Block

）结构定义了串口通信设备的控制设置。许多重要设置都是

在

DCB

结构中设置的，有三种方式可以初始化

DCB

。

(1)

通过

GetCommState()

函数得

DCB

的初始值，其使用方式为：

DCB dcb = {0};

if (!GetCommState(hComm,

＆

dcb))

// Error getting current DCB settings

else

// DCB is ready for use.

(2)

用

BuildCommDCB()

函数初始化

DCB

结构，该函数填充

 DCB

的波特率、奇偶校验类型、

数据位、停止位。对于流控成员函数设置了缺省值。其用法是：

DCB dcb;

FillMemory(

＆

dcb, sizeof(dcb), 0);

dcb.DCBlength = sizeof(dcb);

if (!BuildCommDCB(“9600,n,8,

1",

＆

dcb)) {

// Couldn't build the DCB. Usually a problem

// with the communications specification string.

return FALSE;

}

else

// DCB is ready for use.

(3)

用

SetCommState()

函数手动设置

DCB

初值。用法如下：

DCB dcb;

FillMemory(

＆

dcb, sizeof(dcb), 0);

if (!GetCommState(hComm,

＆

dcb)) // get current DCB

// Error in GetCommState

return FALSE;

// Update DCB rate.

dcb.BaudRate = CBR\_9600

// Set new state.

if (!SetCommState(hComm,

＆

dcb))

// Error in SetCommState.

  Possibly a problem with the communications

// port handle or a problem with the DCB structure itself.

手动设置

DCB

值时，

DCB

的结构的各成员的含义，可以参看

MSDN

帮助。

 2.

流控设置

硬件流控：串口通信中的硬件流控有两种，

DTE/DSR

方式和

RTS/CTS

方式，这与

DCB

结构

的初始化有关系，

DCB

结构中的

OutxCtsFlow

、

 fOutxDsrFlow

、

fDsrSensitivity

、

fRtsControl

、

fDtrControl

几个成员的初始值很关键，不同的值代表不同流控，也可以自己设置流控，但建

议采用标准流行的流控方式。采用硬件流控时，

DTE

、

DSR

、

RTS

、

CTS

的逻辑位直接影响到数据

的读写及收发数据的缓冲区控制。

软件流控：串口通信中采用特殊字符

XON

和

XOFF

作为控制串口数据的收发。与此相关的

DCB

成员是：

fOut

、

fInX

、

XoffChar

、

XonChar

、

 XoffLim

和

XonLim

。

具体含义参见

MSDN

帮助。

串口读写操作：串口读写有两种方式：同步方式（

NonOverlapped

）和异步方式

（

Overlapped

）。同步方式是指必须完成了读写操作，函数才返回，这可能造成程序死掉，因

为如果在读写时发生了错误，永远不返回就会出错，可能线程将永远等待在那儿。而异步方式

则灵活得多，一旦读写不成功，就将读写挂起，函数直接返回，可以通过

GetLastError

函数

得知读写未成功的原因，所以常常采用异步方式操作。

读操作：

ReadFile()

函数用于完成读操作。异步方式的读操作为：

  DWORD dwRead;

  BOOL fWaitingOnRead = FALSE;

  OVERLAPPED osReader = {0};

  // Create the overlapped event. Must be closed before exiting

  // to avoid a handle leak.

  osReader.hEvent = CreateEvent

  (NULL, TRUE, FALSE, NULL);

  if (osReader.hEvent == NULL)

  // Error creating overlapped event; abort.

  if (!fWaitingOnRead) {

  // Issue read operation.

  if (!ReadFile(hComm, lpBuf, READ\_BUF\_SIZE,

＆

dwRead,

＆

osReader)) {

  if (GetLastError() != ERROR\_IO\_PENDING)

    // read not delayed?

  // Error in communications; report it.

  else

  fWaitingOnRead = TRUE;

  }

  else {

  // read completed immediately

  HandleASuccessfulRead(lpBuf, dwRead);

  }

  }

如果读操作被挂起，可以调用

WaitForSingleObject()

函数或

WaitForMuntilpleObjects()

函数等待读操作完成或者超时发生，再调用

GetOverlappedResult()

得到想要的信息。

写操作：与读操作相似，故不详述，调用的

API

函数是：

 WriteFile

函数。

串口状态：

（

1

）通信事件：用

SetCommMask()

函数设置想要得到的通信事件的掩码，再调用

WaitCommEvent()

函数检测通信事件的发生。

可设置的通信事件标志

（即

SetCommMask()

函数所

设置的掩码）可以有

EV\_BREAK

、

EV\_CTS

、

EV\_DSR

、

 EV\_ERR

、

EV\_RING

、

EV\_RLSD

、

EV\_RXCHAR

、

EV\_RXFLAG

、

EV\_TXEMPTY

。

注意：

1

对于

EV\_RING

标志的设置，

WIN95

是不会返回

EV\_RING

事件的，因为

WIN95

不检

测该事件。

2

设置

EV\_RXCHAR

，可以检测到字符到达，但是在绑定此事件和

ReadFile()

函数一

起读取串口接收数据时，可能会出现错误，造成少读字节数，具体原因查看

MSDN

帮助。可以

采用循环读的办法，

另外一个比较好的解决办法是调用

ClearCommError()

函数，

确定在一次读

操作中在缓冲区中等待被读的字节数。

（

2

）

错误处理和通信状态：

在串口通信中，

可能会产生很多的错误，

使用

ClearCommError()

函数可以检测错误并且清除错误条件。

（

3

）

Modem

状态：用

SetcommMask()

可以包含很多事件标志，但是这些事件标志只指示

在串口线路上的电压变化情况。而调用

 GetCommModemStatus()

函数可以获得线路上真正的电

压状态。

扩展函数：

如果应用程序想用自己的流控，

可以使用

 EscapeCommFunction()

函数设置

DTR

和

RTS

线路的电平。

通信超时：在通信中，超时是个很重要的考虑因素，因为如果在数据接收过程中由于某

种原因突然中断或停止，如果不采取超时控制机制，将会使得

I/O

线程被挂起或无限阻塞。串

口通信中的超时设置分为两步，首先设置

 COMMTIMEOUTS

结构的五个变量，然后调用

SetcommTimeouts()

设置超时值。对于使用异步方式读写的操作，如果操作挂起后，异步成功

完成了读写，

WaitForSingleObject()

或

 WaitForMultipleObjects()

函数将返回

WAIT\_OBJECT\_0

，

GetOverlappedResult()

返回

TRUE

。其实还可以用

GetCommTimeouts()

得到系

统初始值。

关闭串口：程序结束或需要释放串口资源时，应该正确关闭串口，关闭串口比较简单，

使用

API

调用

CloseHandle()

关闭串口的句柄就可以了。

调用方法为：

CloseHandle(hComm);

但是值得注意的是在关闭串口之前必须保证读写串口线程已经退出，否则会引起误操作，

一般采用的办法是使用事件驱动机制，启动一事件，通知串口读写线程强制退出，在线程退出

之前，通知主线程可以关闭串口。

**二、实现**

1.

程序设计思路

对于不同的应用程序，虽然界面不同，但是如果采用串口与主机之间的通信，对串口的

处理方式大致相似，无非就是通过串口收发数据，对于通过串口接收到的数据，交给上层软件

处理显示，对于上层要发给串口的数据，进行转发。但在实际编程中，由于采用的通信方式和

流控不同，串口设置也不同，这就涉及到

 DCB

的初始化问题和读写串口等细节问题。串口通信

应用程序设计的总体思路（即操作过程）是：首先，确定要打开的串口名、波特率、奇偶校验

方式、数据位、停止位，传递给

CreateFile()

函数打开特定串口；其次，为了保护系统对串口

的初始设置，调用

 GetCommTimeouts()

得到串口的原始超时设置；然后，初始化

DCB

对象，调

用

SetCommState()

设置

DCB

，调用

SetCommTimeouts()

设置串口超时控制；再次，调用

SetupComm()

设置串口接收发送数据的缓冲区大小，串口的设置就基本完成，之后就可以启动

读写线程了。

一般来说，串口的读写由串口读写线程完成，这样可以避免读写阻塞时主程序死锁。对于

全双工的串口读写，应该分别开启读线程和写线程；对于半双工和单工的，建议只需开启一个

线程即可。在线程中，按照预定好的通信握手方式，正确检测串口状态，读取发送串口数据。

2.

实现细节

在半双工的情况下，首先完成必要的串口配置，成功打开串口、

DCB

设置、超时设置；然

后开启线程，如：

 CwinThread hSerialThread = (CWinThread

＊

)

AfxBeginThread(SerialOperation

，

hWnd

，

THREAD\_PRIORITY\_NORMAL);

其中开启之线程为

SerialOperation

，优先级为普通。

全双工情况下的串口编程，与单工差不多，区别仅仅在于启动双线程，分别为读线程和

写线程，读线程根据不同的事件或消息，通过不断查询串口所收到的有效数据，完成读操作；

写线程通过接收主线程的发送数据事件和要发送的数据，向串口发送。