在工业控制中，工控机（一般都基于Windows平台）经常需要与智能仪表通过串口进行通信。串口通信方便易行，应用广泛。

一般情况下，工控机和各智能仪表通过RS485总线进行通信。RS485的通信方式是半双工的，只能由作为主节点的工控PC机依次轮询网络上的各智能控制单元子节点。每次通信都是由PC机通过串口向智能控制单元发布命令，智能控制单元在接收到正确的命令后作出应答。

在Win32下，可以使用两种编程方式实现串口通信，其一是使用ActiveX控件，这种方法程序简单，但欠灵活。其二是调用Windows的API函数，这种方法可以清楚地掌握串口通信的机制，并且自由灵活。本文我们只介绍API串口通信部分。

串口的操作可以有两种操作方式：同步操作方式和重叠操作方式（又称为异步操作方式）。

同步操作时，API函数会阻塞直到操作完成以后才能返回（在多线程方式中，虽然不会阻塞主线程，但是仍然会阻塞监听线程）；而重叠操作方式，API函数会立即返回，操作在后台进行，避免线程的阻塞。

无论那种操作方式，一般都通过四个步骤来完成：

* （1） 打开串口
* （2） 配置串口
* （3） 读写串口
* （4） 关闭串口

**1、打开串口**

Win32系统把文件的概念进行了扩展。无论是文件、通信设备、命名管道、邮件槽、磁盘、还是控制台，都是用API函数CreateFile来打开或创建的。该函数的原型为：

**C++代码**



|  |  |
| --- | --- |
| 1 | HANDLE CreateFile( LPCTSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode, LPSECURITY\_ATTRIBUTES lpSecurityAttributes, DWORD dwCreationDistribution, DWORD dwFlagsAndAttributes, HANDLE hTemplateFile); |

* lpFileName：将要打开的串口逻辑名，如"COM1"；
* dwDesiredAccess：指定串口访问的类型，可以是读取、写入或二者并列；
* dwShareMode：指定共享属性，由于串口不能共享，该参数必须置为0；
* lpSecurityAttributes：引用安全性属性结构，缺省值为NULL；
* dwCreationDistribution：创建标志，对串口操作该参数必须置为OPEN\_EXISTING；
* dwFlagsAndAttributes：属性描述，用于指定该串口是否进行异步操作，该值为FILE\_FLAG\_OVERLAPPED，表示使用异步的I/O；该值为0，表示同步I/O操作；
* hTemplateFile：对串口而言该参数必须置为NULL。

**同步I/O方式打开串口的示例代码：C++代码**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | HANDLE hCom; //全局变量，串口句柄  hCom=CreateFile("COM1",//COM1口  GENERIC\_READ|GENERIC\_WRITE, //允许读和写  0, //独占方式  NULL,  OPEN\_EXISTING, //打开而不是创建  0, //同步方式  NULL);  if(hCom==(HANDLE)-1)  {  AfxMessageBox("打开COM失败!");  return FALSE;  }  return TRUE; |

**重叠I/O打开串口的示例代码：**

C++代码



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | HANDLE hCom; //全局变量，串口句柄  hCom =CreateFile("COM1", //COM1口  GENERIC\_READ|GENERIC\_WRITE, //允许读和写  0, //独占方式  NULL,  OPEN\_EXISTING, //打开而不是创建  FILE\_ATTRIBUTE\_NORMAL|FILE\_FLAG\_OVERLAPPED, //重叠方式  NULL);  if(hCom ==INVALID\_HANDLE\_VALUE)  {  AfxMessageBox("打开COM失败!");  return FALSE;  }  return TRUE; |

**2、配置串口**

在打开通讯设备句柄后，常常需要对串口进行一些初始化配置工作。这需要通过一个DCB结构来进行。DCB结构包含了诸如波特率、数据位数、奇偶校验和停止位数等信息。在查询或配置串口的属性时，都要用DCB结构来作为缓冲区。

一般用CreateFile打开串口后，可以调用GetCommState函数来获取串口的初始配置。要修改串口的配置，应该先修改DCB结构，然后再调用SetCommState函数设置串口。  
DCB结构包含了串口的各项参数设置，下面仅介绍几个该结构常用的变量：

typedef struct \_DCB{ ………

DWORD BaudRate;//波特率，指定通信设备的传输速率。这个成员可以是实际波特率值或者下面的常量值之一：  CBR\_110，CBR\_300，CBR\_600，CBR\_1200，CBR\_2400，CBR\_4800，CBR\_9600，CBR\_19200， CBR\_38400， CBR\_56000， CBR\_57600， CBR\_115200， CBR\_128000， CBR\_256000， CBR\_14400

DWORD fParity; // 指定奇偶校验使能。若此成员为1，允许奇偶校验检查 …

BYTE ByteSize; // 通信字节位数，4—8

BYTE Parity; //指定奇偶校验方法。此成员可以有下列值： EVENPARITY 偶校验 NOPARITY 无校验 MARKPARITY 标记校验 ODDPARITY 奇校验

BYTE StopBits; //指定停止位的位数。此成员可以有下列值： ONESTOPBIT 1位停止位 TWOSTOPBITS 2位停止位  
ON 5STOPBITS   1.5位停止位  
GetCommState函数可以获得COM口的设备控制块，从而获得相关参数：



|  |  |
| --- | --- |
| 1  2  3  4  5 | BOOLGetCommState(  HANDLEhFile,//标识通讯端口的句柄  LPDCBlpDCB//指向一个设备控制块（DCB结构）的指针 );  SetCommState函数设置COM口的设备控制块：  BOOLSetCommState(HANDLEhFile,LPDCBlpDCB); |

除了在BCD中的设置外，程序一般还需要设置I/O缓冲区的大小和超时。Windows用I/O缓冲区来暂存串口输入和输出的数据。如果通信的速率较高，则应该设置较大的缓冲区。调用SetupComm函数可以设置串行口的输入和输出缓冲区的大小。  
BOOL SetupComm( HANDLE hFile, // 通信设备的句柄

DWORD dwInQueue, // 输入缓冲区的大小（字节数）

DWORD dwOutQueue // 输出缓冲区的大小（字节数） );  
在用ReadFile和WriteFile读写串行口时，需要考虑超时问题。超时的作用是在指定的时间内没有读入或发送指定数量的字符，ReadFile或WriteFile的操作仍然会结束。  
要查询当前的超时设置应调用GetCommTimeouts函数，该函数会填充一个COMMTIMEOUTS结构。调用SetCommTimeouts可以用某一个COMMTIMEOUTS结构的内容来设置超时。  
读写串口的超时有两种：间隔超时和总超时。间隔超时是指在接收时两个字符之间的最大时延。总超时是指读写操作总共花费的最大时间。写操作只支持总超时，而读操作两种超时均支持。用COMMTIMEOUTS结构可以规定读写操作的超时。  
COMMTIMEOUTS结构的定义为：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | typedef struct \_COMMTIMEOUTS {  DWORD ReadIntervalTimeout; //读间隔超时  DWORD ReadTotalTimeoutMultiplier; //读时间系数  DWORD ReadTotalTimeoutConstant; //读时间常量  DWORD WriteTotalTimeoutMultiplier; // 写时间系数  DWORD WriteTotalTimeoutConstant; //写时间常量  } COMMTIMEOUTS,\*LPCOMMTIMEOUTS; |

COMMTIMEOUTS结构的成员都以毫秒为单位。

总超时的计算公式是：总超时＝时间系数×要求读/写的字符数＋时间常量  
例如，要读入10个字符，那么读操作的总超时的计算公式为：  
读总超时＝ReadTotalTimeoutMultiplier×10＋ReadTotalTimeoutConstant  
可以看出：间隔超时和总超时的设置是不相关的，这可以方便通信程序灵活地设置各种超时。

如果所有写超时参数均为0，那么就不使用写超时。如果ReadIntervalTimeout为0，那么就不使用读间隔超时。如果ReadTotalTimeoutMultiplier 和 ReadTotalTimeoutConstant 都为0，则不使用读总超时。如果读间隔超时被设置成MAXDWORD并且读时间系数和读时间常量都为0，那么在读一次输入缓冲区的内容后读操作就立即返回，而不管是否读入了要求的字符。  
在用重叠方式读写串口时，虽然ReadFile和WriteFile在完成操作以前就可能返回，但超时仍然是起作用的。在这种情况下，超时规定的是操作的完成时间，而不是ReadFile和WriteFile的返回时间。

**配置串口的示例代码：**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | SetupComm(hCom,1024,1024);//输入缓冲区和输出缓冲区的大小都是1024  COMMTIMEOUTSTimeOuts;//设定读超时  TimeOuts.ReadIntervalTimeout=1000;  TimeOuts.ReadTotalTimeoutMultiplier=500;  TimeOuts.ReadTotalTimeoutConstant=5000;//设定写超时  TimeOuts.WriteTotalTimeoutMultiplier=500;  TimeOuts.WriteTotalTimeoutConstant=2000;  SetCommTimeouts(hCom,&TimeOuts);//设置超时  DCBdcb;  GetCommState(hCom,&dcb);  dcb.BaudRate=9600;//波特率为9600  dcb.ByteSize=8;//每个字节有8位  dcb.Parity=NOPARITY;//无奇偶校验位  dcb.StopBits=TWOSTOPBITS;//两个停止位  SetCommState(hCom,&dcb);  PurgeComm(hCom,PURGE\_TXCLEAR|PURGE\_RXCLEAR); |

在读写串口之前，还要用PurgeComm()函数清空缓冲区，该函数原型：  
BOOL PurgeComm( HANDLE hFile, //串口句柄

DWORD dwFlags // 需要完成的操作 );  
参数dwFlags指定要完成的操作，可以是下列值的组合：  
PURGE\_TXABORT 中断所有写操作并立即返回，即使写操作还没有完成。

PURGE\_RXABORT 中断所有读操作并立即返回，即使读操作还没有完成。

PURGE\_TXCLEAR 清除输出缓冲区

PURGE\_RXCLEAR 清除输入缓冲区

**3、读写串口**

我们使用ReadFile和WriteFile读写串口，下面是两个函数的声明：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | BOOL ReadFile( HANDLE hFile, //串口的句柄  // 读入的数据存储的地址，  // 即读入的数据将存储在以该指针的值为首地址的一片内存区  LPVOID lpBuffer,  // 要读入的数据的字节数  DWORD nNumberOfBytesToRead,  // 指向一个DWORD数值，该数值返回读操作实际读入的字节数  LPDWORD lpNumberOfBytesRead,  // 重叠操作时，该参数指向一个OVERLAPPED结构，同步操作时，该参数为NULL。  LPOVERLAPPED lpOverlapped );  BOOL WriteFile( HANDLE hFile, //串口的句柄  // 写入的数据存储的地址，  // 即以该指针的值为首地址的  LPCVOID lpBuffer,  //要写入的数据的字节数  DWORD nNumberOfBytesToWrite,  // 指向指向一个DWORD数值，该数值返回实际写入的字节数  LPDWORD lpNumberOfBytesWritten,  // 重叠操作时，该参数指向一个OVERLAPPED结构，  // 同步操作时，该参数为NULL。  LPOVERLAPPED lpOverlapped ); |

在用ReadFile和WriteFile读写串口时，既可以同步执行，也可以重叠执行。在同步执行时，函数直到操作完成后才返回。这意味着同步执行时线程会被阻塞，从而导致效率下降。在重叠执行时，即使操作还未完成，这两个函数也会立即返回，费时的I/O操作在后台进行。  
ReadFile和WriteFile函数是同步还是异步由CreateFile函数决定，如果在调用CreateFile创建句柄时指定了FILE\_FLAG\_OVERLAPPED标志，那么调用ReadFile和WriteFile对该句柄进行的操作就应该是重叠的；如果未指定重叠标志，则读写操作应该是同步的。ReadFile和WriteFile函数的同步或者异步应该和CreateFile函数相一致。  
ReadFile函数只要在串口输入缓冲区中读入指定数量的字符，就算完成操作。而WriteFile函数不但要把指定数量的字符拷入到输出缓冲区，而且要等这些字符从串行口送出去后才算完成操作。  
如果操作成功，这两个函数都返回TRUE。需要注意的是，当ReadFile和WriteFile返回FALSE时，不一定就是操作失败，线程应该调用GetLastError函数分析返回的结果。例如，在重叠操作时如果操作还未完成函数就返回，那么函数就返回FALSE，而且GetLastError函数返回ERROR\_IO\_PENDING。这说明重叠操作还未完成。

同步方式读写串口比较简单，下面先例举同步方式读写串口的代码：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | //同步读串口    charstr[100];    DWORDwCount;//读取的字节数    BOOLbReadStat;    bReadStat=ReadFile(hCom,str,100,&wCount,NULL);    if(!bReadStat){AfxMessageBox("读串口失败!");returnFALSE;}returnTRUE;//同步写串口    charlpOutBuffer[100];    DWORDdwBytesWrite=100;    COMSTATComStat;    DWORDdwErrorFlags;    BOOLbWriteStat;    ClearCommError(hCom,&dwErrorFlags,&ComStat);    bWriteStat=WriteFile(hCom,lpOutBuffer,dwBytesWrite,&dwBytesWrite,NULL);    if(!bWriteStat){AfxMessageBox("写串口失败!");}    PurgeComm(hCom,PURGE\_TXABORT|PURGE\_RXABORT|PURGE\_TXCLEAR|PURGE\_RXCLEAR); |

在重叠操作时,操作还未完成函数就返回。

重叠I/O非常灵活，它也可以实现阻塞（例如我们可以设置一定要读取到一个数据才能进行到下一步操作）。有两种方法可以等待操作完成：一种方法是用象WaitForSingleObject这样的等待函数来等待OVERLAPPED结构的hEvent成员；另一种方法是调用GetOverlappedResult函数等待，后面将演示说明。  
下面我们先简单说一下OVERLAPPED结构和GetOverlappedResult函数：  
OVERLAPPED结构  
OVERLAPPED结构包含了重叠I/O的一些信息，定义如下：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | typedef struct \_OVERLAPPED { // o    DWORD Internal;    DWORD InternalHigh;    DWORD Offset;    DWORD OffsetHigh;    HANDLE hEvent;    } OVERLAPPED; |

在使用ReadFile和WriteFile重叠操作时，线程需要创建OVERLAPPED结构以供这两个函数使用。线程通过OVERLAPPED结构获得当前的操作状态，该结构最重要的成员是hEvent。hEvent是读写事件。当串口使用异步通讯时，函数返回时操作可能还没有完成，程序可以通过检查该事件得知是否读写完毕。  
当调用ReadFile, WriteFile 函数的时候，该成员会自动被置为无信号状态；当重叠操作完成后，该成员变量会自动被置为有信号状态。  
GetOverlappedResult函数 BOOL GetOverlappedResult( HANDLE hFile, // 串口的句柄 // 指向重叠操作开始时指定的OVERLAPPED结构 LPOVERLAPPED lpOverlapped, // 指向一个32位变量，该变量的值返回实际读写操作传输的字节数。 LPDWORD lpNumberOfBytesTransferred, // 该参数用于指定函数是否一直等到重叠操作结束。 // 如果该参数为TRUE，函数直到操作结束才返回。 // 如果该参数为FALSE，函数直接返回，这时如果操作没有完成， // 通过调用GetLastError()函数会返回ERROR\_IO\_INCOMPLETE。 BOOL bWait );  
该函数返回重叠操作的结果，用来判断异步操作是否完成，它是通过判断OVERLAPPED结构中的hEvent是否被置位来实现的。

**异步读串口的示例代码：**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | charlpInBuffer[1024];    DWORDdwBytesRead=1024;    COMSTATComStat;    DWORDdwErrorFlags;    OVERLAPPEDm\_osRead;    memset(&m\_osRead,0,sizeof(OVERLAPPED));    m\_osRead.hEvent=CreateEvent(NULL,TRUE,FALSE,NULL);    ClearCommError(hCom,&dwErrorFlags,&ComStat);    dwBytesRead=min(dwBytesRead,(DWORD)ComStat.cbInQue);    if(!dwBytesRead)returnFALSE;    BOOLbReadStatus;    bReadStatus=ReadFile(hCom,lpInBuffer,dwBytesRead,&dwBytesRead,&m\_osRead);    if(!bReadStatus)    //如果ReadFile函数返回FALSE    {    if(GetLastError()==ERROR\_IO\_PENDING)    //GetLastError()函数返回ERROR\_IO\_PENDING,表明串口正在进行读操作    {    WaitForSingleObject(m\_osRead.hEvent,2000);    //使用WaitForSingleObject函数等待，直到读操作完成或延时已达到2秒钟    //当串口读操作进行完毕后，m\_osRead的hEvent事件会变为有信号    PurgeComm(hCom,PURGE\_TXABORT|PURGE\_RXABORT|PURGE\_TXCLEAR|PURGE\_RXCLEAR);    returndwBytesRead;    }    return0;    }    PurgeComm(hCom,PURGE\_TXABORT|PURGE\_RXABORT|PURGE\_TXCLEAR|PURGE\_RXCLEAR);    returndwBytesRead; |

**对以上代码再作简要说明：**

在使用ReadFile 函数进行读操作前，应先使用ClearCommError函数清除错误。

ClearCommError函数的原型如下：



|  |  |
| --- | --- |
| 1  2  3  4  5 | BOOL ClearCommError( HANDLE hFile,    // 串口句柄LPDWORD lpErrors, // 指向接收错误码的变量    LPCOMSTAT lpStat // 指向通讯状态缓冲区 ); |

该函数获得通信错误并报告串口的当前状态，同时，该函数清除串口的错误标志以便继续输入、输出操作。

参数lpStat指向一个COMSTAT结构，该结构返回串口状态信息。

COMSTAT结构 COMSTAT结构包含串口的信息，

结构定义如下：  
typedef struct \_COMSTAT { // cst DWORD fCtsHold : 1; // Tx waiting for CTS signal DWORD fDsrHold : 1; // Tx waiting for DSR signal DWORD fRlsdHold : 1; // Tx waiting for RLSD signal DWORD fXoffHold : 1; // Tx waiting, XOFF char rec''d DWORD fXoffSent : 1; // Tx waiting, XOFF char sent DWORD fEof : 1; // EOF character sent DWORD fTxim : 1; // character waiting for Tx DWORD fReserved : 25; // reserved DWORD cbInQue; // bytes in input buffer DWORD cbOutQue; // bytes in output buffer } COMSTAT, \*LPCOMSTAT;  
本文只用到了cbInQue成员变量，该成员变量的值代表输入缓冲区的字节数。

最后用PurgeComm函数清空串口的输入输出缓冲区。  
这段代码用WaitForSingleObject函数来等待OVERLAPPED结构的hEvent成员，下面我们再演示一段调用GetOverlappedResult函数等待的

**异步读串口示例代码：**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | charlpInBuffer[1024];  DWORDdwBytesRead=1024;  BOOLbReadStatus;  DWORDdwErrorFlags;  COMSTATComStat;  OVERLAPPEDm\_osRead;  ClearCommError(hCom,&dwErrorFlags,&ComStat);  if(!ComStat.cbInQue)return0;  dwBytesRead=min(dwBytesRead,(DWORD)ComStat.cbInQue);  bReadStatus=ReadFile(hCom,lpInBuffer,dwBytesRead,&dwBytesRead,&m\_osRead);  if(!bReadStatus)//如果ReadFile函数返回FALSE  {if(GetLastError()==ERROR\_IO\_PENDING)  {GetOverlappedResult(hCom,&m\_osRead,&dwBytesRead,TRUE);  // GetOverlappedResult函数的最后一个参数设为TRUE，  //函数会一直等待，直到读操作完成或由于错误而返回。  returndwBytesRead;}  return0;}  returndwBytesRead; |

**异步写串口的示例代码：**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | char buffer[1024];  DWORD dwBytesWritten=1024;  DWORD dwErrorFlags;  COMSTAT ComStat;  OVERLAPPED m\_osWrite;  BOOL bWriteStat;  bWriteStat=WriteFile(hCom,buffer,dwBytesWritten, &dwBytesWritten,&m\_OsWrite);  if(!bWriteStat)  { if(GetLastError()==ERROR\_IO\_PENDING)  { WaitForSingleObject(m\_osWrite.hEvent,1000);  return dwBytesWritten; }  return 0; }  return dwBytesWritten; |

**4、关闭串口**

利用API函数关闭串口非常简单，只需使用CreateFile函数返回的句柄作为参数调用CloseHandle即可：



|  |  |
| --- | --- |
| 1  2  3 | BOOLCloseHandle(  HANDLEhObject;//handle to object to close  ); |