深入浅出VC 串口之第三方类（上） PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/140/2021_2022__E6_B7_B1_E5_85_A5_E6_B5_85_E5_c100_140213.htm 串口类 从本系列文章连载三、四可以看出，与通过WIN32 API进行串口访问相比，通过MScomm这个Activex控件进行串口访问要来的方便许多，它基本上可以向用户屏蔽多线程的细节，以事件（发出OnComm消息）方式实现串口的异步访问。 尽管如此，MScomm控件的使用仍有诸多不便，譬如其发送和接收数据都要进行VARIANT类型对象与字符串的转化等。因此，国内外许多优秀的程序员自己编写了一些串口类，使用这些类，我们将可以更方便的操作串口。在笔者的《深入浅出Win32多线程程序设计之综合实例》（网址：http://dev.yesky.com）一文中，曾向读者展示了由Remon Spekreijse编写的CSerialPort串口类，而本文将向您展示由程序员llbird编写的cnComm(中国串口？)串口类。 llbird是一位优秀的程序员，他的代码风格简洁而紧凑，类的声明和实现都被定义在一个头文件中，使用这个类的朋友只需要在工程中包含这一头文件即可：/*Comm Base Library(WIN98/NT/2000) ver 1.1Compile by： BC 5. C BUILDER 4, 5, 6, X. VC 5, 6. VC.NET. GCC.copyright(c) 2004.5 - 2005.8 llbird wushaojian@21cn.com*/ #ifndef _CN_COMM_H_#define _CN_COMM_H_#pragma warning(disable: 4530)#pragma warning(disable: 4786)#pragma warning(disable: 4800)#include #include #include //送到窗口的消息 WPARAM 端口号#define ON_COM_RECEIVE WM_USER 618#define ON_COM_CTS

WM_USER 619 //LPARAM 1 valid#define ON_COM_DSR WM_USER 621 //LPARAM 1 valid#define ON_COM_RING WM_USER 623#define ON_COM_RLSD WM_USER 624#define ON_COM_BREAK WM_USER 625#define ON_COM_TXEMPTY WM_USER 626#define ON_COM_ERROR WM_USER 627 //LPARAM save Error ID#define DEFAULT_COM_MASK_EVENT EV_RXCHAR | EV_ERR | EV_CTS | EV_DSR | EV_BREAK | EV_TXEMPTY | EV_RING | EV_RLSDclass cnComm{ public: //----------Construction---------- //第1个参数为是否在打开串口时启动监视线程, 第2个参数为IO方式 阻塞方式（0）/ 异步重叠方式（默认） cnComm(bool fAutoBeginThread = true, DWORD dwIOMode =FILE_FLAG_OVERLAPPED): _dwIOMode(dwIOMode), _fAutoBeginThread(fAutoBeginThread) { Init(). } virtual ~cnComm() { Close(). UnInit(). } //----------Attributes---------- //判断串口是否打开 inline bool IsOpen() { return _hCommHandle != INVALID_HANDLE_VALUE. } //判断串口是否打开 operator bool() { return _hCommHandle != INVALID_HANDLE_VALUE. } //获得串口句炳 inline HANDLE GetHandle() { return _hCommHandle. } //获得串口句炳 operator HANDLE() { return _hCommHandle. } //获得串口参数 DCB DCB *GetState() { return IsOpen() amp. ::GetCommState(_hCommHandle, amp._DCB: NULL. } //设置串口参数 DCB bool SetState(DCB *pdcb = NULL) { return IsOpen()

? ::SetCommState(_hCommHandle, pdcb == NULL ? amp._DCB) != TRUE) return false. if (::BuildCommDCB(szSetStr, amp._DCB) == TRUE. } return false. } //设置串口参数：波特率，停止位，等 bool SetState(DWORD dwBaudRate, DWORD dwByteSize = 8, DWORD dwParity =NOPARITY, DWORD dwStopBits = ONESTOPBIT) { if (IsOpen()) { if (::GetCommState(_hCommHandle, amp._DCB) == TRUE. } return false. } //获得超时结构 LPCOMMTIMEOUTS GetTimeouts(void) { return IsOpen() amp. ::GetCommTimeouts(_hCommHandle, amp._CO: NULL. } //设置超时 bool SetTimeouts(LPCOMMTIMEOUTS lpCO) { return IsOpen() ? ::SetCommTimeouts(_hCommHandle, lpCO) == TRUE:false. } //设置串口的I/O缓冲区大小 bool SetBufferSize(DWORD dwInputSize, DWORD dwOutputSize) { return IsOpen() ? ::SetupComm(_hCommHandle, dwInputSize, dwOutputSize)== TRUE: false. } //关联消息的窗口句柄 inline void SetWnd(HWND hWnd) { assert(::IsWindow(hWnd)). _hNotifyWnd = hWnd. } //设定发送通知, 接受字符最小值 inline void SetNotifyNum(DWORD dwNum) { _dwNotifyNum = dwNum. } //线程是否运行 inline bool IsThreadRunning() { return _hThreadHandle != NULL. } //获得线程句柄 inline HANDLE GetThread() { return _hThreadHandle. } //设置要监视的事件, 打开前设置有效 void SetMaskEvent(DWORD dwEvent = DEFAULT_COM_MASK_EVENT) { _dwMaskEvent = dwEvent. } //获得读缓冲区的字符数 int GetInputSize() { COMSTAT Stat. DWORD dwError. return ::ClearCommError(_hCommHandle,

amp.Stat) == TRUE ? Stat.cbInQue : (DWORD) - 1L. } //--------------------------------Operations---------------------------------- //打开串口 缺省 9600, 8, n, 1 bool Open(DWORD dwPort) { return Open(dwPort, 9600). } //打开串口 缺省 baud_rate, 8, n, 1 bool Open(DWORD dwPort, DWORD dwBaudRate) { if (dwPort 1024) return false. BindCommPort(dwPort). if (!OpenCommPort()) return false. if (!SetupPort()) return false. return SetState(dwBaudRate). } //打开串口, 使用类似"9600, 8, n, 1"的设置字符串设置串口 bool Open(DWORD dwPort, char *szSetStr) { if (dwPort 1024) return false. BindCommPort(dwPort). if (!OpenCommPort()) return false. if (!SetupPort()) return false. return SetState(szSetStr). } //读取串口 dwBufferLength个字符到 Buffer 返回实际读到的字符数 可读任意数据 DWORD Read(LPVOID Buffer, DWORD dwBufferLength, DWORD dwWaitTime = 10) { if (!IsOpen()) return 0. COMSTAT Stat. DWORD dwError. if (::ClearCommError(_hCommHandle, amp.Stat) amp. dwError > 0) { ::PurgeComm(_hCommHandle,PURGE_RXABORT | PURGE_RXCLEAR). return 0. } if (!Stat.cbInQue) // 缓冲区无数据 return 0. unsigned long uReadLength = 0. dwBufferLength = dwBufferLength > Stat.cbInQue ? Stat.cbInQue :dwBufferLength. if (!::ReadFile(_hCommHandle, Buffer, dwBufferLength, amp._ReadOverlapped)) { if (::GetLastError() == ERROR_IO_PENDING) { WaitForSingleObject(_ReadOverlapped.hEvent, dwWaitTime). // 结束异步I/O if (!::GetOverlappedResult(_hCommHandle,

amp.uReadLength, false)) { if (::GetLastError() != ERROR_IO_INCOMPLETE) uReadLength = 0. } } else uReadLength = 0. } return uReadLength. } //读取串口 dwBufferLength - 1 个字符到 szBuffer 返回ANSI C 模式字符串指针 适合一般字符通讯 char *ReadString(char *szBuffer, DWORD dwBufferLength, DWORD dwWaitTime =20) { unsigned long uReadLength = Read(szBuffer, dwBufferLength - 1,dwWaitTime). szBuffer[uReadLength] = \0. return szBuffer. } //写串口 可写任意数据 "abcd" or "\x0\x1\x2" DWORD Write(LPVOID Buffer, DWORD dwBufferLength) { if (!IsOpen()) return 0. DWORD dwError. if (::ClearCommError(_hCommHandle, amp.amp.uWriteLength,amp.dwError, NULL) amp. dwError > 0) { ::PurgeComm(_hCommHandle,PURGE_RXABORT | PURGE_RXCLEAR). return 0. } DWORD uReadLength = 0. ::ReadFile(_hCommHandle, Buffer, dwBufferLength, amp.dwError, NULL) amp. dwError > 0) ::PurgeComm(_hCommHandle, PURGE_TXABORT | PURGE_TXCLEAR). unsigned long uWriteLength = 0. ::WriteFile(_hCommHandle, Buffer, dwBufferLength, amp.id). return (_hThreadHandle != NULL). } return false. } //暂停监视线程 inline bool SuspendThread() { return IsThreadRunning() ? ::SuspendThread(_hThreadHandle) !=0xFFFFFFFF: false. } //恢复监视线程 inline bool ResumeThread() { return IsThreadRunning() ? ::ResumeThread(_hThreadHandle) !=0xFFFFFFFF: false. } //终止线程 bool EndThread(DWORD dwWaitTime = 100) { if (IsThreadRunning()) { _fRunFlag = false.

::SetCommMask(_hCommHandle, 0). ::SetEvent(_WaitOverlapped.hEvent). if (::WaitForSingleObject(_hThreadHandle, dwWaitTime) !=WAIT_OBJECT_0) if (!::TerminateThread(_hThreadHandle, 0)) return false. ::CloseHandle(_hThreadHandle). ::ResetEvent(_WaitOverlapped.hEvent). _hThreadHandle = NULL. return true. } return false. } protected: volatile DWORD _dwPort. //串口号 volatile HANDLE _hCommHandle. //串口句柄 char _szCommStr[20]. //保存COM1类似的字符串 DCB _DCB. //波特率，停止位，等 COMMTIMEOUTS _CO. //超时结构 DWORD _dwIOMode. // 0 同步 默认 FILE_FLAG_OVERLAPPED重叠I/O异步 OVERLAPPED _ReadOverlapped, _WriteOverlapped. // 重叠I/O volatile HANDLE _hThreadHandle. //辅助线程 volatile HWND _hNotifyWnd. // 通知窗口 volatile DWORD _dwNotifyNum. //接受多少字节(>=_dwNotifyNum)发送通知消息 volatile DWORD _dwMaskEvent. //监视的事件 volatile bool _fRunFlag. //线程运行循环标志 bool _fAutoBeginThread. //Open() 自动 BeginThread(). OVERLAPPED _WaitOverlapped. //WaitCommEvent use //初始化 void Init() { memset(_szCommStr, 0, 20). memset(amp._ReadOverlapped, 0, sizeof(_ReadOverlapped)). memset(amp._WaitOverlapped, 0, sizeof(_WaitOverlapped)). _WaitOverlapped.hEvent = ::CreateEvent(NULL, true, false, NULL). assert(_WaitOverlapped.hEvent != INVALID_HANDLE_VALUE). } //析构 void UnInit() { if (_ReadOverlapped.hEvent != INVALID_HANDLE_VALUE)

CloseHandle(_ReadOverlapped.hEvent). if (_WriteOverlapped.hEvent != INVALID_HANDLE_VALUE) CloseHandle(_WriteOverlapped.hEvent). if (_WaitOverlapped.hEvent != INVALID_HANDLE_VALUE) CloseHandle(_WaitOverlapped.hEvent). } //绑定串口 void BindCommPort(DWORD dwPort) { assert(dwPort >= 1 amp. dwPort char p[5]. _dwPort = dwPort. strcpy(_szCommStr, "\\\\.\\COM"). ltoa(_dwPort, p, 10). strcat(_szCommStr, p). } //打开串口 virtual bool OpenCommPort() { if (IsOpen()) Close(). _hCommHandle = ::CreateFile(_szCommStr, GENERIC_READ | GENERIC_WRITE, 0, NULL,OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL | _dwIOMode,NULL). if (_fAutoBeginThread) { if (IsOpen() amp. BeginThread()) return true. else { Close(). //创建线程失败 return false. } } return IsOpen(). } //设置串口 virtual bool SetupPort() { if (!IsOpen()) return false. if (!::SetupComm(_hCommHandle, 4096, 4096)) return false. if (!::GetCommTimeouts(_hCommHandle, amp._CO)) return false. if (!::PurgeComm(_hCommHandle, PURGE_TXABORT | PURGE_RXABORT |PURGE_TXCLEAR | PURGE_RXCLEAR)) return false. return true. } //----------------------------------threads callback---------------------------------- //线程收到消息自动调用, 如窗口句柄有效, 送出消息, 包含串口编号， 均为虚函数可以在基层类中扩展 virtual void OnReceive() //EV_RXCHAR { if (::IsWindow(_hNotifyWnd)) ::PostMessage(_hNotifyWnd, ON_COM_RECEIVE, WPARAM(_dwPort), LPARAM (0)). }

virtual void OnDSR() { if (::IsWindow(_hNotifyWnd)) { DWORD Status. if (GetCommModemStatus(_hCommHandle, amp.MS_DSR_ON) ? 1 : 0)). } } virtual void OnCTS() { if (::IsWindow(_hNotifyWnd)) { DWORD Status. if (GetCommModemStatus(_hCommHandle, amp.MS_CTS_ON) ? 1 : 0)). } } virtual void OnBreak() { if (::IsWindow(_hNotifyWnd)) { ::PostMessage(_hNotifyWnd, ON_COM_BREAK, WPARAM(_dwPort), LPARAM(0)). } } virtual void OnTXEmpty() { if (::IsWindow(_hNotifyWnd)) ::PostMessage(_hNotifyWnd, ON_COM_TXEMPTY, WPARAM(_dwPort), LPARAM (0)). } virtual void OnError() { DWORD dwError. ::ClearCommError(_hCommHandle, amp.amp.dwMask, amp._WaitOverlapped,amp.dwError, amp.). cnComm amp.). //base function for thread static DWORD WINAPI CommThreadProc(LPVOID lpPara) { return ((cnComm*)lpPara)->ThreadFunc(). } }.#endif //_CN_COMM_H_