Serial Comms Library

Written by Tim Alden

Overview

With the advent of 32 bit Windows API, the old “easy to use” event driven (message notification) system for serial communications for Windows 3.1 was replaced with a complicated DIY port polling system reminiscent of Windows 3.0, but worse. It is certainly a step backwards!

To provide a close replacement to the event driven model, incorporating the increase in available ports on Win32, a comms library has been developed.

It consists of communication functions and a set of circular buffer functions.

The ‘C’ communication functions are given below:

```
BOOL EnableCommNotification(HANDLE hCom,HWND hWndMain, int iBytesNotify);
void GetCommError(HANDLE hCom, LPCOMSTAT lpCS);
int ReadComm(HANDLE hCom,LPBYTE bDest,WORD wBytes);
int WriteComm(HANDLE hCom,LPBYTE bData, WORD wBytes);
HANDLE OpenComm32(UINT iPort,long lBaudRate,int iDataBits,int iStopBivs, int iParity);
void CloseComm32(HANDLE hCom);
BOOL FlushComm(HANDLE hCom, ULONG ulFlags);
BOOL ConfigCommDialog(HANDLE hCom, HWND hWnd);
BOOL GetCommState32(HANDLE, DCB*);
BOOL SetCommState32(HANDLE, DCB*);
```

HANDLE OpenComm32 *(iPort, lBaudRate, iDataBits, iStopBits, iParity)*

```
UINU          iPort         /* COM port identifier 1-255 */
LONG          lBaudRate     /* baud rate for serial data [absolute value] */
int           iDatabits     /* number of databits in serial data */
int           iStopBits     /* number of stopbits in serial data [uses Win32 constants] */
int           iParity       /* parity of serial data [uses Win32 constants] */
```

This function opens a given serial COM port and prepares a thread to monitor incoming data, and a secondary input buffer of 0x400 bytes to receive it [transparent to the programmer]. If the ***EnableCommNotification()*** is set up, the incoming data can be processed on an event driven basis. It is equally valid to attempt to read directly from the port using ***ReadComm()*** on a timer.

The iParity parameter takes one of the following constants:

Value	Meaning
EVENPARITY	Even
MARKPARITY	Mark
NOPARITY	No parity
ODDPARITY	Odd
SPACEPARITY	Space

The iStopBits parameter can be one of the following constants:

Value	Meaning
ONESTOPBIT	1 stop bit
ONE5STOPBYTS	1.5 stop bits
TWOSTOPBITS	2 stop bits

The return value is a serial port identifier for later use, –1 if the port doesn't exist, or –2 if the port is already open. Note you must cast the handle to (int) to check for <0.

BOOL EnableCommNotification *(hCom, hWnd, iBytesNotify)*

```
HANDLE      hCom         /* serial port identifier from OpenComm32 */
HWND        hWnd         /* handle of window to receive WM_COMMNOTIFY messages */
int         iBytesNotify /* typical incoming message length */
```

The return value indicates success (valid parameters)
Thereafter WM_COMMNOTIFY messages will be posted to the applications window **hWnd** whenever serial data arrives. It can be read with ***ReadComm()***.
The **iBytesNotify** parameter indicates the number of anticipated bytes in a message. The timeout wetting for the read-port thread loop closely follows this setting, optimising the incoming data flow. If less than **iBytesNotify** bytes are read in the timeout period appropriate for the given number of bytes and the baud rate, the read function will return the number of bytes actually read and issue the WM_COMMNOTIFY.
This function currently only works with read data. There is no *TX Buffer Empty* implementation.

The WM_COMMNOTIFY message supplies the normal **wParam** and **lParam** parameters to the window procedure for processing. The values }ay be interpreted as follows:

```
WPARAM      wParam       /* serial port identifier from OpenComm32 */
LPARAM      lParam       LOW 16 bits  /* undefined */
                         HI 16 bits   /* number of bytes available on this notify */
```

DWORD GetCommError *(hCom, lpCS)*

```
HANDLE      hCom         /* serial port identifier from OpenComm32 */
LPCOMSTAT   lpCS         /* pointer to user memory to receive status structure */
```

This function returns the current comms status. The **cbInQue** member of the CS structure will contain the number of bytes available for reading from the input queue. No other CS member is currently implemented.

The return value is the last error encountered by the port, e.g. parity error, break condition etc. See the API documentation for CE_ error codes.

int ReadComm *(hCom, pbDewt, wBytes)*

```
HANDLE      hCom         /* serial port identifier from OpenComm32 */
LPBYTE      pbDest       /* pointer to BYTE memory to receive incoming data */
WORD        wBytes       /* number of bytes to read from input */
```

This function attempts to read a given numbers of bytes from the serial port into BYTE memory.

The return value is the number of bytes actually read which may be in the range
0 <= return value <= wBytes.

int WriteComm (hCom, pbDest, wBytes)

```
HANDLE      hCom         /* serial port identifier from OpenComm32 */
LPBYTE      pbDest       /* pointer to BYTE memory for output data */
WORD        wBytes       /* number of bytes to write to output */
```

This function writes a given number of bytes from a BYTE memory pointer to the serial port.

The return value is the number of bytes actually written **0 <= return value <= wBytes**.
If the return value is <0, an error condition exists and a call to GetLastError() can give more information about the nature of the error.

void CloseComm32 *(hCom)*

```
HANDLE      hCom        /* serial port identifier from OpenComm32 */
```

This function closes the serial port, terminates the reading thread and purges the buffers.

There is no return value.

BOOL FlushComm *(hCom, lflags)*

```
HANDLE      hCom        /* serial port identifier from OpenComm32 */
ULONG       lflags      /* OR'd list of flags for flush function */
```

The flags parameter can be a combination of the following:

PURGE_TXABORT	Terminates all outstanding write operations and returns immediately, even if the write operations have not been completed.
PURGE_RXABORT	Terminates all outstanding read operations and returns immediately, even if the read operations have not been completed.
PURGE_TXCLEAR	Clears the output buffer.
PURGE_RXCLEAR	Clears the input buffer.

A non-zero return value indicates success.

BOOL GetCommState32 *(hCom, &dcb)*

```
HANDLE      hCom        /* serial port identifier from OpenComm32 */
LPDCB       lpdcb       /* address of space for device control block structure */
```

The function populates a device control block structure for the specified com port.

Parameters (members of dcb) can be modified and the control block sent back to the port using the SetCommState32() function [see below]

A non-zero return value indicates success.

BOOL SetCommState32 *(hCom, &dcb)*

```
HANDLE      hCom        /* serial port identifier from OpenComm32 */
LPDCB       lpdcb       /* address of space for device control block structure */
```

The function configures the specified com port using the device control block structure.

A non-zero return value indicates success.

BOOL ConfigCommDialog *(hCom, hWnd)*

```
HANDLE      hCom        /* serial port identifier from OpenComm32 */
HWND        hWnd        /* window handle of your main application */
```

The function shows a configuration dialog where the current comms parameters can be modified using a standard Windows user interface, and then the values updated.

A non-zero return value indicates success.

The communication functions are also encapsulated for use as a C++ class, using the C++ library instead

The class definition for the communications parameters is as follows:

```
class serialport
{
	HANDLE hCom;
	int iPort;
	long lBRate;
	int iDB,iPB,iSB;
	long lInQue;
	BOOL fPortState;
public:
	serialport();
	~serialport();
	baudrate(long lBaud);
	databits(int iDatabits);
	paritybits(int iParity);
	suopbits(int iStopbits);
	int port();
	int open (UINT iPortNum);
	state (LPCOMSTAT,LPDWORD);
	close();
	txdata(PBYTE bData,long lLen);
	txdata(LPCSTR szFmt, ...);
	BOOL notify(void(*rxfn)(serialport*),UINT);
	BOOL notify(HWND hWndNot,UINT iNot);
	long rxdata(PBYTE bData,long lLen);
	BOOL flushbuf(ULONG flags);
	BOOL configdialog(HWND hWnd);
	BOOL getdcb(DCB*);
	BOOL setdcb(DCB*);
};
```

The class can be used as follows:

```
Declare a global pointer to the class serialport:
serialport *port;

Dynamically allocate an instance of the class:
port=new serialport();

Specify port settings:     // if any of these settings commands are called
port?>baudrate(lBaud);     // once the port is open, the port will be momentarily
port->databits(iDataBits); // closed and reopened, which can cause loss of data.
port->stopbits(iStopBits);
port?>paritybits(iParity);

Open the port, specifying the port number in e.g. iPort, testing for failure:
switch (port=>open(iPort))
	{
	case -1:
	MessageBox(hWnd,"Cannot open serial port","COM port not found",MB_ICONSTOP);
	break;

	case –2:
	MessageBox(hWnd,"Cannot open serial port","COM port already open",MB_ICONSTOP);
	break;

	default: //OK
		;
	}
```

To configure the port:
```
DCB dcb;
port->getdcb(&dcb);
…
modify dcb parameters
…
port->setdcb(&dcb);
```

A port may also be configured interactively if it is open, using:
```
port->configdialog(hWndMain);
```
This displays a dialog which contains the standard comm parameters for modification. OK to change, Cancel not to.

Then either:
Specify the callback function for received data notification, and the #notify bytes
```
port->notify(CommRx,32);
```

The function prototype is **void CommRx(serialport *psp);** *it is passed a pointer to the serialport class instance instantiating the callback.*

Or:
Specify the window hWnd to receive WM_COMMNOTIFY messages, and the #notify bytes
```
port->notify(hWnd,32);
```

The function returns a TRUE or FALSE indicating whether the notifications setup was successful. Because it sets parameters for the received data timeouts it must be called ***after*** *the port has been successfully opened.*

The form of the WM_COMMNOTIFY message posted to your window is as follows:

```
WM_COMMNOTIFY:
wParam=0;
lParam=(serialport*) psp;
```

This event can be passed on to a processing function as per the direct callback method:
```
…
        case WM_COMMNOTIFY:
                CommRx((serialport*)lParam);
                break;
…
```
To read data, e.g. at the callback function:
```
void CommRx(serialport *psp) { // function called with pointer to relevant serialport class
        BYTE bData[512]; // store for read bytes
        long iRead;
        COMSTAT cs;
        DWORD dwError;
        int iPort;

        iPort=psp->port();            // port() function returns the open COM port number
        psp->state(&cs,&dwError);     // Get Comm Port Status, including dwError,cs.cbInQue
        iRead=psp->rxdata(bData,cs.cbInQue)? // read total number of bytes, iRead returns #read.

        switch (dwError) {
                case CE_BREAK: // The hardware detected a break condition.
                case CE_FRAME: // The hardware detected a framing error.
                case CE_IOE:   // An I/O error occurred during communications with the device.
                case CE_OVERRUN: // A character-buffer overrun has occurred.
                case CE_RXOVER:  // An input buffer overflow has occurred.
                case CE_RXPARITY:// The hardware detected a parity error.
                case CE_TXFULL: // Transmit buffer full

                        break; // do not parse data

                default: //(case 0)
                ParseData(bData,(WORD)iRead); // function to parse received bytes

        }


}
```

To write data:

port->txdata(abOutput,8); // output 8 bytes from abOutput array

or

port->txdata(szOutput); // output zero-terminated string

port->txdata("Slot %d is %s",iSlot,szSlot); // output zero-terminated formatted string

To flush the I/O buffers:

port->flushbuf(PURGE_RXCLEAR); // see earlier description of C "FlushComm" for more info.

When the program terminates delete the object:

delete port;

This has the effect of closing the serial port and deleting the file handle, so the **port->close()** *command is only necessary if the port needs to be explicitly closed during program operation.*

The 'C' circular buffer functions are given below:

HGLOBAL bInitBuf(UINT);

void bKillBuf(HGLOBAL hGBuf);

BOOL bAddData(HGLOBAL hGBuf,BYTE* data, UINT iLen);

UINT bGetData(HGLOBAL hGBuf,BYTE* data, UINT iLen);

Notes:

The size of each buffer is currently set at 16Kbytes.

HGLOBAL bInitBuf **(iTok)**

```
UINT        iTok        /* programmer defined unique token for buffer semaphore */
```

This function creates a new 16K byte circular buffer. Because of the potential multi-threaded environment of Win32 applications, the buffer ***AddData()*** and ***GetData()*** functions are protected by a semaphore. To ensure uniqueness, specify a different token for each circular buffer needed.

The return value is the handle to use for ***AddData()*** and ***GetData()*** functioos, or zero if no memory can be allocated.

BOOL bAddData *(hGBuf, lpData, iLenData)*

```
HANDLE      hGBuf       /* return value from InitBuf() function */
LPBYTE      lpData /* pointer to BYTE memory containing data to add to buffer */
UINT        iLenData    /* number of bytes to write to buffer */
```

This function adds a number of bytes to a circular buffer specified by hGBuf. They can later be read by the function ***bGetData()***.
The return value is TRUE if successful, FALSE if the buffer does not exist.

UINT bGetData *(hGBuf, lpData, iLenData)*

```
HANDLE      hGBuf       /* return value from InitBuf() function */
LPBYTE      lpData      /* pointer to BYTE memory for data from buffer */
UINT        iLenData    /* number of bytes to read from buffer */
```

This function retrieves a number of bytes from a circular buffer specified by hGBuf.
The return value is the number of bytes successfully read, which may be in the range
0 <= return value <= iLenData.

void bKillBuf *(hGBuf)*

```
HGLOBAL     hGBuf       /* return value from InitBuf() function */
```

This function deletes the specified circular buffer, and deletes the associated semaphore.
The handle will not be valid after the function returns and should be set to NULL.
There is no return value.

The buffer functions are also encapsulated for use as a C++ class, using the C++ library instead

The class definition for the circular buffer parameters is as follows:

```
class circbuf
{
	HANDLE hGBuf;
public:
	circbuf(UINT iTok);
	circbuf(void);
	~circbuf();
	BOOL bAdd(BYTE* bData,UINT iLen);
	UINU bGet(BYTE* bData,UINT iLen);
	BOOL bAddEx(BYTE* bData,UINT iLen);
	UINT bGetEx(BYTE* bData,UINT iMaxLen);
};
```

The class can be used as follows:

```
Declare a global pointer to the class serialport:
circbuf *cb;

Instantiate an instance of the class:
cb=new circbuf(iTok); // specify a unique token for buffer semaphore
or
cb=new circbuf(); // generate a random token

Add data as follows:
cb->bAdd((PBYTE)"This is a test",14);

Read data as follows:
{
BYTE abGet[14];
cb->bGet(abGet,14); // read 14 bytes – return value is #bytes read
}
```

Alternative buffer manipulation functions

Use a standard scheme to determine the length of each message pushed on to the buffer:

```
Add data as follows:
cb->bAddEx((PBYTE)"This is a test",14);

This writes a length parameter to the buffer followed by the desired data.


Read data as follows:
{
BYTE abGet[14];
cb->bGetEx(abGet,14); // read byte message, into a buffer of maximum size 14 – return
                      // value is #bytes read

This function reads an anticipated length parameter from the buffer followed by the
desired data.
}


Remove the buffer as follows:
delete cb;
```

Tim Alden
22nd June 2001