Windows Serial Port Programming

Robertson Bayer

March 30, 2008

Intro

This paper covers the basics of opening, reading from, writing to, and managing serial port communications using the Windows API. The overall goal here is to give you a basic overview of how serial communications programming works and to at least get you started along the right track. For a fully rigorous treatment of the subject, check out Allen Denver's "Serial Communications in Win32." Also, for a look at some actual code that uses this stuff, I'd recommend downloading the source code to two of my old programs RoboGUI and RoboCon, both of which are available at my website, www.robbayer.com.

This paper will assume you know some basic C/C++, are able to compile and run programs, and that your development environment is setup for using Windows API calls (Visual C++ does this automatically).

There are 6 sections to this document: opening the serial port, setting serial port properties, setting timeouts, reading and writing data, cleaning up after yourself, and advanced functions.

1 Opening the serial port

Opening the serial port is very easy, especially if you've ever done Windows file I/O before. First, make sure you include windows.h. You can then use the following code to open it:

```
HANDLE hSerial;

hSerial = CreateFile("COM1",
                     GENERIC_READ | GENERIC_WRITE,
                     0,
                     0,
                     OPEN_EXISTING,
                     FILE_ATTRIBUTE_NORMAL,
                     0);

if(hSerial==INVALID_HANDLE_VALUE){
    if(GetLastError()==ERROR_FILE_NOT_FOUND){
        //serial port does not exist.  Inform user.
    }
    //some other error occurred. Inform user.
}
```

So let's walk through this. First, we declare a variable of type `HANDLE` and initialize it with a call to `CreateFile`. The first argument to `CreateFile` is simply the name of the file you want to open. In this case, we want to open a serial port, so we use `"COM1"`, `"COM2"`, etc. The next argument tells Windows whether you want to read or write to the serial port. If you don't need to do one of these, just leave it out. Arguments 3 and 4 should pretty much always be 0. The next argument tells us that Windows should only open an existing file, and since serial ports already exist, this is what we want. After this comes `FILE_ATTRIBUTE_NORMAL`, which just tells Windows we don't want anything fancy here. The final argument should also be zero.

2 Setting Parameters

So now that we have a HANDLE to the serial port, we need to set the parameters for it (baud rate, stop bits, etc). Windows does this through a struct called DCB:

```
DCB dcbSerialParams = {0};

dcbSerial.DCBlength=sizeof(dcbSerialParams);

if (!GetCommState(hSerial, &dcbSerialParams)) {
    //error getting state
}

dcbSerialParams.BaudRate=CBR_19200;
dcbSerialParams.ByteSize=8;
dcbSerialParams.StopBits=ONESTOPBIT;
dcbSerialParams.Parity=NOPARITY;

if(!SetCommState(hSerial, &dcbSerialParams)){
    //error setting serial port state
}
```

Once again, we'll walk through this line-by-line. First, we create a variable of type `DCB`, clear all its fields, and set its size parameter (this is required due to a strange quirk of Windows). We then use the `GetCommState` function, which takes in our serial port handle and the `DCB` struct, to fill in the parameters currently in use by the serial port.

Anyway, once we have this, we just need to set the parameters we care about. Notably, baud rate, byte size, stop bits, and parity. For whatever reason, Windows requires that we use special constants to specify the baud rate. These constants are pretty straightforward: `CBR_19200` for 19200 baud, `CBR_9600` for 9600 baud, `CBR_57600` for 57600 baud, etc.

Byte size we can just specify directly, but stop bits and parity once again require special constants. The options for StopBits are `ONESTOPBIT, ONE5STOPBITS, TWOSTOPBITS`. Similarly, the most commonly used options for Parity are `EVENPARITY, NOPARITY, ODDPARITY`. There are others, but these are quite obscure. See the MSDN library entry (do search for DCB) for full details.

Similarly, there are loads of other fields in a DCB struct that you can use for some other more obscure serial parameters. Once again, see the MSDN library.

After we've set up the DCB struct how we want it, we need to apply these setting to the serial port. This is accomplished with the `SetCommState` function.

3 Setting timeouts

One of the big problems with serial port communication is that if there isn't any data coming into the serial port (if the serial port device gets disconnected or turned off, for example) then attempting to read from the port can cause your application to hang while waiting for data to show up. There are two ways of going about fixing this. First, you can use multithreading in your application with one thread dealing with the serial port stuff and the other doing the actual processing. This can get very messy and complex and really isn't necessary. The other way is much simpler: just tell Windows not to wait for data to show up! This is accomplished quite simply:

```
COMMTIMEOUTS timeouts={0};

timeouts.ReadIntervalTimeout=50;
timeouts.ReadTotalTimeoutConstant=50;
timeouts.ReadTotalTimeoutMultiplier=10;
```

```
timeouts.WriteTotalTimeoutConstant=50;
timeouts.WriteTotalTimeoutMultiplier=10;

if(!SetCommTimeouts(hSerial, &timeouts)){
  //error occureed.  Inform user
}
```

The `COMMTIMEOUTS` structure is pretty straightforward, and the above fields are the only ones it has. A quick rundown:

- `ReadIntervalTimeout` specifies how long (in milliseconds) to wait between receiving characters before timing out.
- `ReadTotalTimeoutConstant` specifies how long to wait (in milliseconds) before returning.
- `ReadTotalTimeoutMultiplier` specifies how much additional time to wait (in milliseconds) before returning for each byte that was requested in the read operation.
- `WriteTotalTimeoutConstant` and `WriteTotalTimeoutMultiplier` do the same thing, just for writes instead of reads.

One special case that comes up pretty often: Setting `ReadIntervalTimeout` to `MAXDWORD` and both `ReadTotalTimeoutConstant` and `ReadTotalTimeoutMultiplier` to zero will cause any read operations to return immediately with whatever characters are in the buffer (ie, have already been received), even if there aren't any.

After we've setup the `COMMTIMEOUTS` structure how we want it, we need to apply the settings to the serial port using the `SetCommTimeouts` function.

4 Reading/Writing data

Alright, so once you have an open serial port with the correct parameters and timeouts, you can start doing the actual reads. These are extremely simple. Suppose we want to read n bytes from the serial port. Then we just do:

```
char szBuff[n + 1] = {0};
DWORD dwBytesRead = 0;

if(!ReadFile(hSerial, szBuff, n, &dwBytesRead, NULL)){
    //error occurred.  Report to user.
}
```

`ReadFile` takes in a `HANDLE` to a file (our serial port in this case), a buffer to store the data in, the number of bytes to read, a pointer to an integer that will be set to the number of bytes actually read, and NULL.

Note that `dwBytesRead` will contain the number of bytes actually read by the `ReadFile` operation.

Writing data is exactly the same, except the function is called `WriteFile`.

5 Closing down

Once you're done using the serial port, make sure you close the handle. If you don't weird things can happen, like nobody else being able to access the serial port until you reboot. In any event, it's really simple to do, so just remember to do it:

```
CloseHandle(hSerial);
```

6 Errors

As you've probably noticed, after each system-call, I've included a comment saying you should handle errors. This is ALWAYS good programming practice, but is especially important with I/O functions, since they do tend to fail more often that most functions. In any event, all the above functions return 0 whenever they fail and something other than 0 on success. To find out exactly what the error was, you can call the function `GetLastError()`, which returns the error code as a DWORD. To convert this into a string that makes sense, you can use the `FormatMessage` function as follows:

```
char lastError[1024];
FormatMessage(
            FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
            NULL,
            GetLastError(),
            MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
            lastError,
            1024,
            NULL);
```

7 Advanced functions

For the majority of people, this section will be completely irrelevant, so feel free to ignore it unless you know you need some advanced stuff (setting DTR lines, breaks, etc).

Nearly all advanced functions are accomplished using the `EscapeCommFunction` function, which has prototype `BOOL EscapeCommFunction(HANDLE, DWORD)`. In any event, the first parameter is just the HANDLE to the serial port you're interested in, and the second is one of the following (self-explanatory) constants:

1. CLRDTR
2. SETDTR
3. SETRTS
4. CLRRTS
5. SETBREAK
6. CLRBREAK

About the only other advanced thing I can think of is clearing the read/write buffers, which is done with a simple call to `FlushFileBuffers`, which takes just one argument, the `HANDLE` to the file whose buffers you want to flush.

Contacting me

If you have any questions, suggestions, or corrections for this document, please let me know via email at rob.bayer@gmail.com. Also, the latest version of this document (and any others I've written) can be found at my website, www.robbayer.com.

Sources

1. Petzold, Charles. Programming Windows. 5th edition. Redmond, WA: Microsoft Press, 1999.
2. Denver, Allen. "Serial Communications in Win32." MSDN Online Library. 11 December 1995. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwbgen/html/msdn_serial.asp. 3 Jun 2002.