Serial Port Programming in Windows and Linux

Maxwell Walter

November 7, 2003

Abstract

While devices that use RS−232 and the serial port to communicate are becoming increasingly rare, it is still an important skill to have. Serial port programming, at its most basic level, consists of a series of steps regardless of the operating system that one is operating on. These steps are opening the port, configuring the port, reading and writing to the port, and finally, closing the port. It is possible then to create an API that contains the functions necessary to successfully communicate with the serial port. With a single header file that contains all the functions necessary to communicate over a serial port, the implementation details can remain platform dependent as a library that can then be compiled and maintained separately for each operating system. The actual application can then use the common header file as its interface to the serial port. This creates a cross-platform serial interface allowing the creation of code that is more easily portable across operating systems. In this document the functions and steps necessary to use the serial port are detailed, and at the end of this document an example cross-platform header file is provided.

Keywords: Serial Port, Programming, Linux, Windows

1 Introduction

The advent of USB has caused a significant decline in the number of devices that communicate using RS−232, and many motherboards today ship without serial ports. This does not mean, however, that serial port programming is no longer a necessary or relevant skill to posses. Many important and useful devices still in use today date back to pre-USB times, and there are still some devices that continue to use the RS−232 protocol for communication. Also, because there are far fewer devices that communicate using RS−232, fewer people are exposed to it, and thus fewer people learn how to use it, making it an even more valuable skill.

Serial port programming requires API's provided by the underlying operating system to communicate with the hardware, making serial port programming operating system dependent and not very portable. However, because communicating with the serial port itself requires few steps, a library with a common API can be created and separate versions maintained for different operating systems. The goal of this document is to develop an operating system independent class allowing the creation of portable code that utilizes the serial port.

2 RS−232 Overview

The RS−232 protocol was originally designed in 1969 by the Electronic Industries Association (EIA) as a general method of data transmission between two devices. They wanted to create a specification that would be free of transmission errors while being simple enough that it would be adopted by many device manufactures. The protocol that they developed specified signal levels, timing, mechanical plugs, and information control protocols. The RS−232 specification has gone through three major milestones, namely:

- 1969: The third version of the RS−232 protocol (originally called RS−232−C) was adopted as a standard by the industry.
- 1987: EIA created a new version of the standard called EIA-232-D.
- 1991: EIA and Telecommunications Industry Association created a new version of the standard that they called EIA/TIA-232-E. Most called the standard RS−232−C or simply RS−232.

Other specifications have been created that aim to remove defects inherent in the original RS−232 specification including:

- RS−422: Allows a line impedance of up to 50Ω

- RS−423: Minimum line impedance of 450Ω
- RS−449: High rate of communication speed using a 37 pin plug

[2] [3]

3 Serial Port Programming

There are four operations that must happen in proper order to successfully communicate over the serial port. These are to open the serial port, configure the serial port, read and write bytes to and from the serial port, and close the serial port when the task has been completed. It is important to check the return codes at each stage to ensure that the task has been completed, because failure at one stage means that the next stage will fail as well. Sanity checking the return codes is also good coding practice, and throwing away important information can lead to program errors that are difficult to trace.

The serial port is a bi-directional port meaning that it can send data in both directions, and, because it has separate pins for transmit and receive, a device can send and receive data at the same time. Sending and receiving data at the same time is called overlapped communication as opposed to non-overlapped, or blocking communication. This paper will deal exclusively with non-overlapped communication as overlapped communication requires very complex objects such as threads, mutexes, and semaphores. Also, non-overlapped communication is adequate in many situations.

The two operating systems discussed here, Windows and Linux, both have simple communication API's that facilitate communication over the serial port. While certain tasks, such as opening and closing the serial port, are very similar, other tasks such a configuring the serial port are very different. It is our goal to create a single header file that provides a consistent interface across multiple operating systems, requiring only that the programmer link against the object library for the operating system his program is compiling on.

3.1 Opening the Serial Port

The first step that is necessary to communicate over the serial port is to actually open the device. Under both Windows and Linux, the serial port is treated as a file, so to open the serial port one calls the functions used to open a file. There are restrictions on the parameters allowed when opening the file however, and those will be discussed in the appropriate section below. The options are operating system dependent.

3.1.1 Windows

The function used to open the serial port in the Windows operating system is CreateFile() which is used as follows:

```
HANDLE fileHandle;
fileHandle =
CreateFile(
    //the name of the port (as a string)
    //eg. COM1, COM2, COM4
    gszPort,
    //must have read AND write access to
    //port
    GENERIC_READ | GENERIC_WRITE,
    //sharing mode
    //ports CANNOT be shared
    //(hence the 0)
    0,
    //security attributes
    //0 here means that this file handle
    //cannot be inherited
    0,
    //The port MUST exist before-hand
    //we cannot create serial ports
    OPEN_EXISTING,
    //Overlapped/Non-Overlapped Mode.
    //This paper will deal with
    //non-overlapped communication.
    //To use overlapped communication
    //replace 0 with
    //FFILE_FLAG_OVERLAPPED
    0,
    //HANDLE of a template file
    //which will supply attributes and
    //permissions.  Not used with
    //port access.
    0);
```

CreateFile() returns a HANDLE object that can then be used to access the port. If CreateFile() fails, the HANDLE object that is returned is invalid and validity can be tested using the following code:

```
if (fileHandle == INVALID_HANDLE_VALUE) {
   //error handling code here
}
```

Provided that the serial port is successfully opened the next step is to configure it to the specific application.

3.1.2 Linux

Opening the serial port in Linux is performed in much the same way as it is in Windows, using the open() command. One caveat to Linux is that the user ID that the program is running under must be allowed to access the serial port, either by giving the user permission to the serial port, or by changing the permissions of the program to allow access to the serial port. Code to open the serial port under Linux can is as follows.

```
int fd =
    open(
        //the name of the serial port
        //as a c-string (char *)
        //eg. /dev/ttys0
        serialPortName,
        //configuration options
        //O_RDWR - we need read
        //    and write access
        //O_CTTY - prevent other
        //    input (like keyboard)
        //    from affecting what we read
        //O_NDELAY - We don't care if
        //    the other side is
        //    connected (some devices
        //    don't explicitly connect)
        O_RDWR | O_NOCTTY | O_NDELAY
    );
if(fd == -1) {
    //error code goes here
}
```

The serial port is now opened and, in this case, fd is a handle to the opened device file. As can be seen, if the open() function call fails, the device handle is set to -1 and by checking the handle against this value one can determine if an error occurred.

3.2 Configuring the Serial Port

The next action that must take place before data can be written to or read from the serial port is to appropriately configure the port. Port configuration options include the communication speed, data word size, parity, flow control, number of stop bits. There are other settings that can be used but will not be discussed here. The communication speed is the speed at which the serial port can send and receive data and is specified in units of BAUD. Common speeds are 9600 BAUD, 19200 BAUD, and the current maximum of standard serial ports is 115200 BAUD. Data word size is the size of a piece of the data, usually a word, or eight bits. Parity is the type of parity used, either even, odd, or none, and flow control and the number of stop bits are used to synchronize the communication.

It is also necessary to set the read and write timeouts for non-overlapped communication, because in non-overlapped communication the read and write operations block, or do not return, until there is data available to return. So if a read or write function is called and there is no data available or no device listening, the program will hang. Setting the read and write timeouts ensure that, if there is no device to communicate with the read and write operations will return a failure instead of blocking. It is important to note that while it is possible to set the wait timeout, unless the host is expecting acknowledgment from the device there will be no write timeouts. Non-overlapped I/O with no flow-control uses no acknowledgment, so for the purpose of this paper the wait timeouts are not needed.

3.2.1 Windows

In Windows, setting up the serial port requires three steps.

1. Create a DCB object and initialize it using the function BuildCommDCB().

2. Set the serial port settings using the initialized DCB object using the function SetCommState().

3. Set the size of the serial port read and write buffers using SetupComm().

Code to accomplish this can be found below.

```
DCB dcb; //create the dcb

//first, set every field
//of the DCB to 0
//to make sure there are
//no invalid values
FillMemory(&dcb, sizeof(dcb), 0);

//set the length of the DCB
dcb.DCBlength = sizeof(dcb);

//try to build the DCB
```

```
//The function takes a string that
//is formatted as
//speed,parity,data size,stop bits
//the speed is the speed
//    of the device in BAUD
//the parity is the
//    type or parity used
//--n for none
//--e for even
//--o for odd
//the data size is the number of bits
//  that make up a work (typically 8)
//the stop bits is the
//    number of stop bits used
//  typically 1 or 2
if(!BuildCommDCB("9600,n,8,1", &dcb)) {
    return false;
}

//set the state of fileHandle to be dcb
//returns a boolean indicating success
//or failure
if(!SetCommState(filehandle, &dcb)) {
    return false;
}

//set the buffers to be size 1024
//of fileHandle
//Also returns a boolean indicating
//success or failure
if(!SetupComm(fileHandle,
              //in queue
              1024,
              //out queue
              1024))
{
    return false;
}
```

Next, the timeouts of the port must be set. It is important to note that if the timeouts of a port are not set in windows, the API states that undefined results will occur. This leads to difficulty debugging because the errors are inconsistent and sporadic. To set the timeouts of the port an object of type COMMTIMEOUTS must be created and initialized and then applied to the port using the function SetCommTimeouts(), as the code below demonstrates.

```
COMMTIMEOUTS cmt; //create the object

//the maximum amount of time
//allowed to pass between
//the arrival of two bytes on
//the read line (in ms)
cmt.ReadIntervalTimeout = 1000;

//value used to calculate the total
//time needed for a read operation
//which is
//  (num bytes to read) * (timeout)
// in ms
cmt.ReadTotalTimeoutMultiplier = 1000;

//This value is added to
//the previous one to generate
//the timeout value
//for a single read operation (in ms)
cmt.ReadTotalTimeoutConstant = 1000;

//the next two values are the same
//as their read counterparts, only
//applying to write operations
cmt.WriteTotalTimeoutConstant = 1000;
cmt.WriteTotalTimeoutMultiplier = 1000;

//set the timeouts of fileHandle to be
//what is contained in cmt
//returns boolean success or failure
if(!SetCommTimeouts(fileHandle, &cmt)) {
    //error code goes here
}
```

Provided all the configuration functions returned success, the serial port is now ready to be used to send and receive data. It may be necessary, depending on the application, to re-configure the serial port. It may be necessary, for instance, to change the speed of the port, or the timeout values in the middle of an application.

3.2.2 Linux

Configuration of the serial port under Linux takes place through the use the termios struct, and consists of four steps:

1. Create the struct and initialize it to the current port settings.
2. Set the speed attribute of the struct to the desired port speed using the functions cfsetispeed() and cfsetospeed(). While these functions allow different reading and writing speed, most hardware and system implementations do not allow these speeds to be different.
3. Set the timeouts of the port.

4. Apply the settings to the serial port.

While opening the serial port is generally easier in Linux than it is in Windows, configuring it is harder as there is more bit-masking involved to change a single option. While there are convenient functions that can be used to set the speed of the port, other options, like parity and number of stop bits, are set using the c_cflag member of the termios struct, and require bitwise operations to set the various settings. Linux is also only capable of setting the read timeout values. This is set using the c_cc member of the termios struct which is actually an array indexed by defined values. Please see the code below for an example.

```
//create the struct
struct termios options;

//get the current settings of the
//    serial port
tcgetattr(fd, &options);

//set the read and write speed to
//19200 BAUD
//All speeds can be prefixed with B
//as a settings.
cfsetispeed(&options, B19200);
cfsetospeed(&options, B19200);

//now to set the other settings
//here we will have two examples.
//The first will be no parity,
//the second will be odd parity.
//Both will assume 8-bit words

/**********************************
*
*    no parity example
*
**********************************/
//PARENB is enable parity bit
//so this disables the parity bit
options.c_cflag &= ~PARENB
//CSTOPB means 2 stop bits
//otherwise (in this case)
//only one stop bit
options.c_cflag &= ~CSTOPB
//CSIZE is a mask for all the
//data size bits, so anding
//with the negation clears out
//the current data size setting
options.c_cflag &= ~CSIZE;
//CS8 means 8-bits per work
options.c_cflag |= CS8;

/**********************************
*
*    odd parity example
*
**********************************/
//enable parity with PARENB
options.c_cflag |= PARENB
//PARAODD enables odd parity
//and with ~PARAODD for even parity
options.c_cflag |= PARODD
//Only one stop bit
options.c_cflag &= ~CSTOPB
//clear out the current word size
options.c_cflag &= ~CSIZE;
//we only have 7-bit words here
//because one bit is taken up
//by the parity bit
options.c_cflag |= CS7;

//Set the timeouts
//VMIN is the minimum amount
//of characters to read.
options.c_cc[VMIN] = 0;
//The amount of time to wait
//for the amount of data
//specified by VMIN in tenths
//of a second.
optiont.c_cc[VTIME] = 1;

//CLOCAL means don't allow
//control of the port to be changed
//CREAD says to enable the receiver
options.c_cflag |= (CLOCAL | CREAD);

//apply the settings to the serial port
//TCSNOW means apply the changes now
//other valid options include:
//    TCSADRAIN - wait until every
//        thing has been transmitted
//    TCSAFLUSH - flush buffers
//        and apply changes
if(tcsetattr(fd, TCSANOW, &options)!= 0) {
    //error code goes here
}
```

3.3 Reading and Writing

After the port is open and configured, a program can send and receive data through it. In both Windows and Linux the serial port is treated as a file, and the file read and write operations are used to send data to and from the port. These operations differ from standard file input/output operations in that the number of bytes to be read or written and the number of bytes actually read or written are very important.

It is a good idea to compare the number of bytes that were actually read or written to the number of bytes that were supposed to have been read or written to insure the correctness of the program and the accuracy of the data.

3.3.1 Windows

Reading and writing to a serial port in Windows is very simple and similar to reading and writing to a file. In fact, the functions used to read and write to a serial port are called ReadFile() and WriteFile respectively. When reading or writing to the serial port, the programmer provides a pointer to a buffer containing the number of words to be written and the size of the buffer. The system then returns the actual number of words that have been read or written. The read and write functions return a boolean value, but this value does not relate to the success or failure of the operation. Only by checking the actual number of bytes read or written can the actual success of the operation be ascertained. Code to read and write to a serial port can be found below.

```
/******************************
*
* Reading from a file
*
*******************************/
//the amount of the data actually
//read will be returned in
//this variable
DWORD read = -1;
ReadFile(
    //the HANDLE that we
    //are reading from
    fileHandle,
    //a pointer to an array
    //of words that we
    //want to read
    data,
    //the size of the
    //array of values to
    //be read
    size,
    //the address of a DWORD
    //that the number of words
    //actually read will
    //be stored in
    &read,
    //a pointer to an
    //overlapped_reader struct
    //that is used in overlapped
    //reading.  NULL in out case
    NULL);

/******************************
*
* Writing to a file
*
*******************************/
//the amount of the data actually
//written will be returned in
//this variable
DWORD write = -1;
ReadFile(
    //the HANDLE that we
    //are writing to
    fileHandle,
    //a pointer to an array
    //of words that we
    //want to write
    data,
    //the size of the
    //array of values to
    //be written
    size,
    //the address of a DWORD
    //that the number of words
    //actually written will
    //be stored in
    &read,
    //a pointer to an
    //overlapped_reader struct
    //that is used in overlapped
    //writing.  NULL in out case
    NULL);
```

Because we are using non-overlapped input/output operations the ReadFile() and WriteFile() operations will block until the requested data is either received or the operation times out.

3.3.2 Linux

Reading and writing data under Linux is just as easy as it is under Windows and involves calling the read() and write() functions. The same rule of thumb of checking the number of requested words to read or write and the actual number of words that were read or written applies.

```
/*********************************
    Reading from the Serial Port
**********************************/
//fd is the file descriptor to the
//    serial port
//buf is a pointer the array we want to
//    read data into
//bufSize is the amount of data that we
//    want to read in
int wordsRead = read(fd, buf, bufSize);

/*********************************
    Writing to the Serial Port
**********************************/
//write data to the serial port
//fd is the file descriptor of
//    the serial port
//buf is a pointer to the data that
//    we want to write to the serial
//    port
//bufSize is the amount of data that we
//    want to write
int wordsWritten = write(fd, buf, bufSize);
```

3.4 Closing the Serial Port

Closing the port is very similar to opening it, and the system call used to close a file is also used to close the port. It is not strictly necessary to close the port before closing the application because modern operating systems like Microsoft Windows and Linux reclaim all resources used by the application. It is, however, good practice to explicitly free all resources used so as to not rely on methods that can't be controlled. Also, if an application will continue to reside in memory after it is finished using the port, like a daemon, then it is necessary to close the port. Otherwise no other application would be able to use the port while the other application has control of it. For this reason, explicitly closing the port when it is no longer in use is a good habit to use.

3.4.1 Windows

Closing the serial port is the easiest task under the Windows operating system. It consists of calling a single function, CloseFile(), and checking the return value. Code that demonstrates this can be found below.

```
//Close the fileHandle, thus
//releasing the device.
if(!CloseFile(fileHandle)) {
    //error code goes here
}
```

3.4.2 Linux

Closing the serial port on Linux involves a single call to the close() system function, as the following code demonstrates.

```
//close the serial port
if(close(fd) == -1) {
    //error code goes here
}
```

4 Putting it all Together

Now that we have an understanding of the necessary steps that must be completed to communicate with the serial port in both Windows and Linux, it is time to create our cross-platform API to facilitate easily portable serial port code. Based on the steps outlined above, our API will require, at minimum, six functions. Two functions to open and to close the serial port, one function to set the serial port parameters, another to set the timeout values, and two functions to read and write to the serial port. The header file of our proposed class can be found below.

```
#ifndef __CAPSTONE_CROSS_SERIAL_PORT__
#define __CAPSTONE_CROSS_SERIAL_PORT__

/***************************************************************************
This header defines a cross-platform serial interface class
that was developed over the course of the ECE capstone class.

It is intended for demonstration purposes only and, while useful,
is not sufficient for most large applications.  While we use these
functions in our project, they most likely do not apply
to all applications, and as always, Your Mileage May Vary (YMMV)

With that in mind, I hope that at the very least, you find this useful.
***************************************************************************/

class crossPlatformSerial {
    /**********************************************************************
        Default constructor: create an instance of our serial interface
            object.
        Parameters: None, though it could be convenient to have default
            settings specified here as well as possibly a default port name.
        Returns: Nothing (it is a constructor)
    **********************************************************************/
    crossPlatformSerial();

    /**********************************************************************
        Default deconstructor: delete an instance of out serial interface
            object.
        Parameters: None (it is a de-constructor)
        Returns: Noting (it is a de-constructor)
    **********************************************************************/
    ~crossPlatformSerial();

    /**********************************************************************
        open: opens the serial port using port-name provided.
        Parameters:
            portName: an array of chars containing a NULL terminated string
                that is the port name that we want to open.
                eg. COM1, COM2, COM4
        Returns: a boolean indicating success or failure
    **********************************************************************/
    bool open(char * portName);

    /**********************************************************************
        close: closes the serial port
        Parameters: None.  None are needed.
        Returns: a boolean indicating success or failure
    **********************************************************************/
    bool close();

    /**********************************************************************
        setSettings: Set the settings of the serial port interface.  Should
            be called after the port is opened because otherwise settings
            could be lost.
        Parameters:
            speed: the desired speed of the serial port
            wordSize: the size of the data word in bits
            stopBits: the number of stop bits
            parity: parity to be used.  One of
                'n' - None
                'e' - Even
                'o' - Odd
        Returns: boolean indicating success or failure
    ***********************************************************************/
    bool setSettings(int speed, int wordSize,
                     int stopBits, char parity);


    /**********************************************************************
        setTimeouts: set the read and write timeouts of the serial port
        Parameters:
            readTimeout: The amount of time to wait for a single word
                read to timeout
            writeTimeout: The amount of time to wait for a single word
                write to timeout
        Returns: boolean indicating success or failure
    **********************************************************************/
    bool setTimeouts(int readTimeout, int writeTimeout);


    //NOTE:
    //    These readBuf and writeBuf functions assume word
    //    sizes of 8 bits.  If that is not the case char *
    //    cannot be used as the storage medium as chars are
    //    8-bits in size.
    /***********************************************************************
        readBuf: read in a buffer of words
        Parameters:
            buf: The buffer to read words into
            bfSize: the number of words to read. This should be less than
                or equal to the size of buf
        Returns: the number of words actually read
    ***********************************************************************/
    int readBuf(char * buf, int bufSize);
    /***********************************************************************
        writeBuf: write out a buffer of words
        Parameters:
            buf: The buffer to write words from
            bfSize: the number of words to write. This should be less than
                or equal to the size of buf
        Returns: the number of words actually written
    ***********************************************************************/
    int writeBuf(char * buf, int bufSize);
};

#endif
```

This class demonstrates the minimum necessary to create a useful serial interface. These function stubs can be filled in with the code provided previously in the paper under the appropriate sections. A separate binary library would need to be maintained for each operating system that the code is intended to run on, though maintaining a library is significantly easier than maintaining the large code-base of an application.

In practice, there are other things that are needed such as a boolean flag indicating whether or not the serial port is currently open, and some method of getting the current settings of the serial port device. It is also important to realize that this API applies only to non-overlapped communication because overlapped communication requires complicated operating system features such as threads, mutexes, and semaphores making cross-platform operation much harder.

5 Conclusion

This document has demonstrated the code necessary to communicate using a serial port on two different operating systems, namely Microsoft Windows and Linux. It has also provided a small amount of information on the history of RS−232 as well as a design for a simple cross-platform serial port interface API.

This API does have limitations in that it only supports non-overlapped I/O and does not support the advanced features that the operating systems provide. It would be possible to create a common header file that includes overlapped communications by also creating cross-platform libraries of the other components that one needs such as threads, mutexes, and semaphores. Again, there would always be something that the operating system is capable of that the library will not be able to support. In gaining the convenience of a cross-platform interface, one loses the power associated with interfacing with the operating system directly.

References

[1] Michael R. Sweet, Serial Programming Guide for POSIX Operating Systems, 1994, http://www.easysw.com/∼mike/serial/serial.html.

[2] fCoder Group International, RS-232-C History, http://www.lookrs232.com/rs232/history_rs232.htm.

[3] Wikipedia, RS-232, http://en.wikipedia.org/wiki/RS-232.

[4] Allen Denver, Serial Communications in Win32, Dec. 11, 1995, MSDN April 2003.