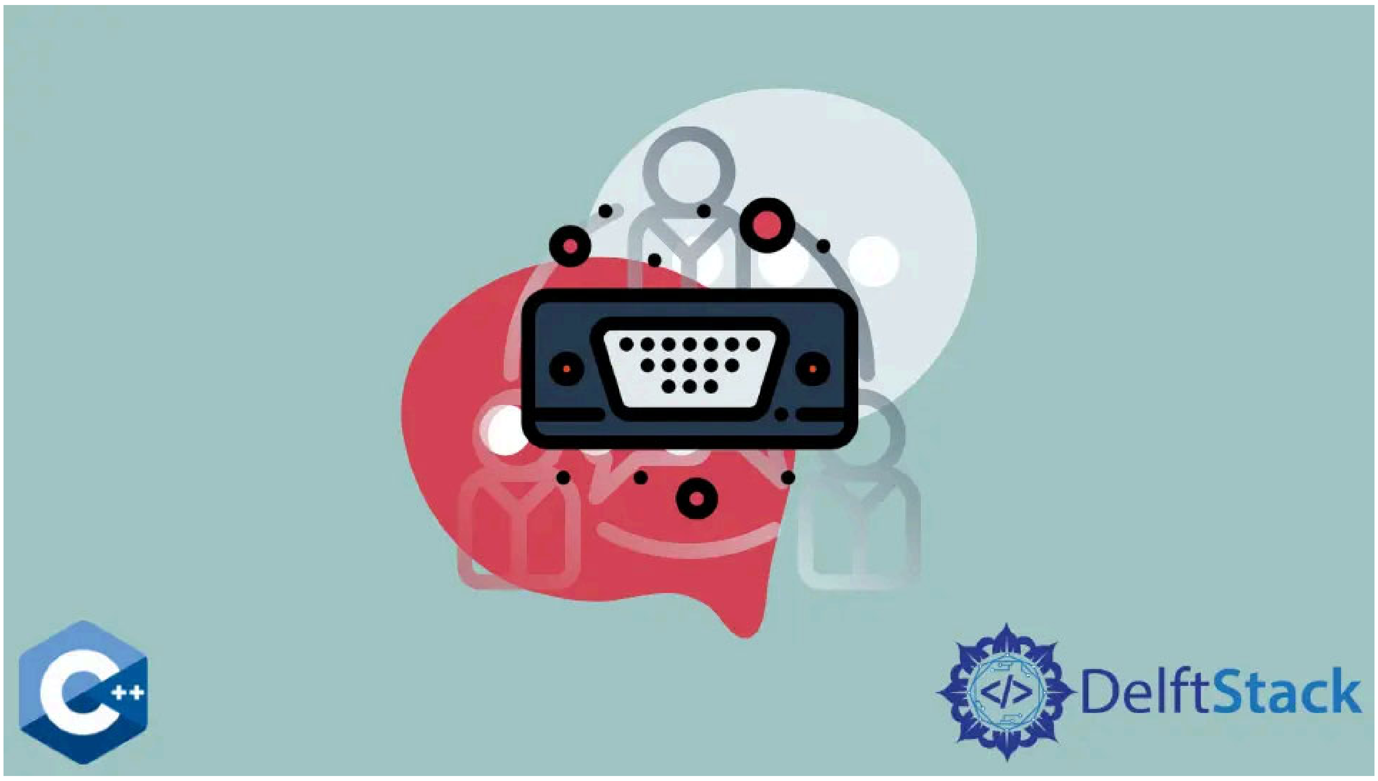

The basics of using the Windows API to open, read from, write to, and manage serial port connections are covered in this article.

The main purpose of this tutorial is to provide you with a basic understanding of how serial communications in programming work and get you started in the correct direction.

This article will presume that you have a basic understanding of C/C++, that you can compile and run programs, and that your development environment is set up to use Windows API calls.

Serial Port Connection in C++

There are six steps for reading data or writing to serial ports in C++:

1. Open the serial ports
2. Set some basic properties
3. Set the timeout
4. Read or write the data
5. Clean up the ports
6. Some advanced functions

Open the Serial Port

The first step is the opening of the port. This is one of the easiest and simple steps, especially if you already know Windows file I/O.

First, you must ensure that you have included the required header files, i.e., `windows.h` (/howto/cpp/windows.h-cpp/) in your file. Then, use the following code:

```
HANDLE h_Serial;
h_Serial = CreateFile("COM1", GENERIC_READ | GENERIC_WRITE, 0, 0, OPEN_EXISTING,
                      FILE_ATTRIBUTE_NORMAL, 0);
if (h_Serial == INVALID_HANDLE_VALUE) {
  if (GetLastError() == ERROR_FILE_NOT_FOUND) {
    // serial port not found. Handle error here.
  }
  // any other error. Handle error here.
}
```

We made a type `HANDLE` variable in the first line and then called the function `CreateFile` to initialize it. The first argument for this function is the name of the file you need to open; in this case, we want to open a serial port, that's why we used the name of that port, i.e., `COM1`.

The second argument is to specify whether we need to read or write the data. If none of the tasks is to be done, you can leave this argument.

The next two arguments are always kept at zero. The next argument is to specify whether one needs to open an existing file or create a new file.

In this case, it is the port, so we used `OPEN_EXISTING`. The next argument is to specify to Windows that we don't need anything fancy but regular reading or writing.

The last argument is also always kept at zero.

Set the Basic Properties

After getting the `HANDLE` of the port, we need to set some of the basic properties like band rate, byte size, stop bits, etc. This is done using a struct `DCB`.

```
DCB dcbSerialParam = {0};
dcbSerial.DCBlength = sizeof(dcbSerialParam);

if (!GetCommState(h_Serial, &dcbSerialParam)) {
  // handle error here
}

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

if (!SetCommState(h_Serial, &dcbSerialParam)) {
  // handle error here
}
```

In the above code snippet, we created an object of `DCB` in the first line of code and initialized it with zero to clear the values. In the following line, we have set the length of this struct which is a mandatory step by Windows.

After that, we called the function `GetCommState` and passed two parameters to it, i.e., our port `HANDLE` and `DCB` object, to fill in the parameters currently in use.

Once we have this, we must set the critical parameters, e.i., `BaudRate`, `ByteSize`, `StopBits`, and `Parity`.

Windows needs us to provide the `BaudRate` using special constants. For example, `CBR 19200` represents 19200 baud, `CBR 9600` represents 9600 baud, `CBR 57600` represents 57600 baud, etc.

We can define the `ByteSize` directly but `StopBits` and `Parity` require extra variables. `ONESTOPBIT`, `ONE5STOPBITS`, and `TWOSTOPBITS` are the `StopBits` possibilities.

`EVENPARITY`, `NOPARITY`, and `ODDPARITY` are the most widely used alternatives for `Parity`. Others exist, but they are less well-known.

See the MSDN library item (search for DCB) for more information.

We need to apply these settings to the serial port after configuring the DCB struct to our priorities. The `SetCommState` function is used to accomplish this.

Set the Timeout

Reading from the serial port can cause your application to halt while waiting for data to appear if no data is coming into the serial port (for instance, the serial port device is switched off or is disconnected). There are two ways to deal with this situation.

First, multithreading can be used in your application, with one thread dealing with serial port difficulties and the other with actual processing. This can become cumbersome and complicated, and it isn't required.

The alternative is much simpler: tell Windows to stop waiting for data to appear! This is accomplished through the following:

```
COMMTIMEOUTS timeout = {0};
timeout.ReadIntervalTimeout = 60;
timeout.ReadTotalTimeoutConstant = 60;
timeout.ReadTotalTimeoutMultiplier = 15;
timeout.WriteTotalTimeoutConstant = 60;
timeout.WriteTotalTimeoutMultiplier = 8;
if (!SetCommTimeouts(h_Serial, &timeout)) {
  // handle error here
}
```

The `COMMTIMEOUTS` structure is rather plain, with simply the fields listed above. A quick recap:

- `ReadIntervalTimeout` - specifies the time that must pass between receiving characters before timing out (in milliseconds).
- `ReadTotalTimeoutConstant` - provides the amount of time to wait before returning (in milliseconds).
- `ReadTotalTimeoutMultiplier` - specifies the length of time to wait before responding for each byte requested in the read operation (in milliseconds).
- `WriteTotalTimeoutConstant` and `WriteTotalTimeoutMultiplier` - both accomplish the same thing as `ReadTotalTimeoutConstant` and `WriteTotalTimeoutMultiplier`, but for writes instead of reads.

Setting `ReadIntervalTimeout` to `MAXDWORD` and both `ReadTotalTimeoutConstant` and `ReadTotalTimeoutMultiplier` to `0` causes any read operations to return instantly with whatever characters in the buffer (i.e., have already been received), even if none are present.

After configuring the `COMMTIMEOUTS` structure, we'll need to use the `SetCommTimeouts` method to apply the changes to the serial port.

Read/Write the Data

You can start reading data after an open serial port with the necessary parameters and timeouts. These are easy to understand.

Consider the scenario of reading `n` bytes from the serial port. Then, we simply follow these steps:

```
char sBuff[n + 1] = {0};
DWORD dwRead = 0;
if (!ReadFile(h_Serial, sBuff, n, &dwRead, NULL)) {
  // handle error here
}
```

A `HANDLE` to a file (serial port), buffer where data is stored, the number of bytes to read, reference to an integer to be set to the number of bytes read, and `NULL` are all passed to `ReadFile`. The number of bytes read by the `ReadFile` operation will be stored in `dwRead`.

Writing the data is also the same procedure. The only difference is that the `WriteFile` function can write data.

Close the Port

When you've finished using the serial port, close the handle. If you don't, strange things might happen, including no one else being able to use the serial port until you reboot.

In any case, it's rather easy to accomplish, so keep the following in mind:

```
CloseHandle(h_Serial);
```

Handle the Errors

As you may have seen, we've inserted a note after each system call stating that you should handle errors.

This is always an excellent programming practice, but it's especially crucial with I/O functions, which fail more frequently. All of the following functions, in any case, return `0` when they fail and anything other than `0` when they succeed.

To figure out what went wrong, use the `GetLastError()` function, which returns the error code as a `DWORD` (/howto/cpp/dword-cpp/). You can use the `FormatMessage` method to turn this into a string that makes sense:

```
char lastErr[1020];
FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS, NULL,
              GetLastError(), MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
              lastErr, 1020, NULL);
```

Author: **Muhammad Husnain** (/author/muhammad-husnain/)

(/author/muhammad-husnain/)

(/author/muhammad-husnain/)

Husnain is a professional Software Engineer and a researcher who loves to learn, build, write, and teach. Having worked various jobs in the IT industry, he especially enjoys finding ways to express complex ideas in simple ways through his content. In his free time, Husnain unwinds by thinking about tech fiction to solve problems around him.

LinkedIn (https://www.linkedin.com/in/muhammad-husnain-b4a08a144/)