

## Die Klasse Serialport

### Ziel, Inhalt

- Wir schreiben eine Klasse die den seriellen Port abstrahiert. Diese Klasse wird von der WIN32 API gebrauch machen und unter anderem mit Threads arbeiten. Wir werden in den folgenden Lektionen auch das Publisher-Subscriber Idiom/Technik kennenlernen

Die Klasse Serialport	1
Ziel, Inhalt	1
Die Klasse Serialport	2
Anforderungen/Analyse Klasse <i>SerialPort</i> , 1. Schritt	2
Erstellung und Zerstörung	2
Öffnen und Schliessen	2
Lesen und Schreiben	2
Zusammenfassung der Analyse	3
Design Klasse <i>SerialPort</i> , 1. Schritt	4
<i>SerialPort</i> , Design	4
Implementation Klasse <i>SerialPort</i> , 1. Schritt	5
Verwendete WIN32-API's	5
Öffnen und Schliessen des seriellen Ports	6
Fehlerbehandlung	7
Lesen und Schreiben	8
Übung	10
Öffnen des richtigen Ports	10
Ergänzen der Methoden <i>readByte</i> und <i>writeByte</i>	10

## Die Klasse Serialport

### Anforderungen/Analyse Klasse *SerialPort*, 1. Schritt

Wir stellen einige Anforderungen an unsere Klasse *SerialPort*. Diese Anforderungen muten einigermaßen willkürlich an. Wir versuchen einfach das zu finden, was ein Anwender von so einem *SerialPort*-Objekt erwartet, ohne dabei an die Implementation zu denken. Ihr werdet aber sehen, dass es bezüglich dieser Schnittstelle der Klasse *SerialPort* unterschiedliche Auffassungen geben kann. Vor allem besteht die Gefahr, dass wir/ich die Schnittstelle zu allgemein definieren und zu viele Möglichkeiten offenlassen, was die Implementation und auch die Anwendung erschwert.

### Erstellung und Zerstörung

Wir lassen zu, dass ein Objekt der Klasse *SerialPort* zu jeder Zeit als beliebige Variable erzeugt werden kann. Das heisst auch als globales Objekt oder auch als dynamisch erstelltes Objekt. Was wir vorwegnehmen können ist auch, dass ein solches Objekt irgendwelche Ressourcen anlegen und brauchen wird. Solche Ressourcen müssen einerseits initialisiert, andererseits auch wieder freigegeben werden. Das führt dazu, dass wir einen Konstruktor und einen Destruktor benötigen.

### Öffnen und Schliessen

Ein serieller Port wird im allgemeinen durch eine Port-Nummer identifiziert. Wir könnten diese Portnummer als Konstruktorparameter mitgeben. Vielleicht möchten wir aber das Objekt mehrmals für verschiedene Ports verwenden können. Um diese Anwendung zu ermöglichen stellen wir zwei Methoden zur Verfügung. Eine Methode *close* und eine Methode *open*.

### Lesen und Schreiben

Um einfaches Schreiben und Lesen zu ermöglichen stellen wir zwei Methoden *read* und *write* zur Verfügung. Hier stellt sich die Frage nach den Datentypen.

Wollen wir ganze string-Objekte lesen und schreiben?

```
void write(const string& text);  
string read();
```

Soll es möglich sein ein Array von Bytes zu lesen und schreiben und dabei die Länge anzugeben?

```
void write(unsigned char* data, unsigned int length);  
void read(unsigned char* buffer, unsigned int length);
```

Wobei bei der *read*-Methode der Aufrufer genügend Speicherplatz (mit der Variable *buffer*) bereitstellen muss, damit der Aufruf korrekt funktioniert.

Hier besteht die Gefahr, dass der Aufrufer mit unserer Klasse das Programm zum Absturz bringt. was sehr unangenehm ist („He, Deine Sch....-Klasse bringt unser Programm immer zum Absturz!“). Wir wollen aber Methoden schreiben die niemals schiefgehen können. Zuallerletzt sollen sie das Programm zum Absturz bringen. Natürlich gibt es noch andere Möglichkeiten diesen Buffer bereitzustellen, aber die Klasse und die Anwendung wird aufwendiger. Wir beschränken uns in diesem ersten Schritt mit der einfachsten aber auch universellsten Variante:

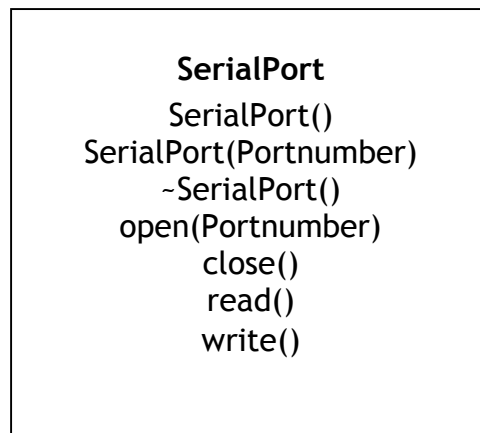
```
void write(unsigned char data);  
unsigned char read();
```

Es ist also nur möglich ein Byte zu lesen und eines zu schreiben. Die Verantwortung, oder die Aufgabe mehr zu tun überlassen wir vorerst dem client, dem Anwender der Klasse *SerialPort*.

### Zusammenfassung der Analyse

Unsere erste Version der Klasse *SerialPort* kann nun grob skizziert werden. Wobei wir bei der Analyse ein wenig auf das Design vorgegriffen haben, denn wir haben bereits über Datentypen gesprochen, was man eigentlich während einer Analysephase vermeiden sollte.

Hier die Klasse *SerialPort*:



Die Klasse wird in diesem ersten Schritt nur synchrones Lesen und Schreiben ermöglichen. Das heisst der Methodenaufruf zum Schreiben so lange dauert, bis die Daten geschrieben sind. Insbesondere der Aufruf zum Lesen dauert so lange bis die Daten eintreffen, das kann unter Umständen nie geschehen. Dadurch würde der Programmfluss unterbrochen und unser Programm würde nicht mehr auf Benutzereingaben reagieren können. Wir werden aber in einem zweiten Schritt auch asynchrones Lesen und Schreiben ermöglichen.

## Design Klasse SerialPort, 1. Schritt

Das Design haben wir zum Teil vorweggenommen. Ihr werdet in anderen Fächern später ein strengeres Vorgehen lernen. Schlussendlich muss jeder für sich den besten Weg finden um:

- fehlerfreie
- wartbare
- erweiterbare
- verständliche
- leicht verwendbare

Klassen und Software zu schreiben.

Hier werden wir einfach die vorhandenen Methoden mit Datentypen ergänzen, wobei ich direkt die Headerdatei zur Klasse *SerialPort* hinschreiben werde, aber noch die eine oder andere Methode oder das eine oder andere Datenelement später, bei der Implementation hinzukommen werden.

### SerialPort, Design

```
#ifndef SERIALPORT_H
#define SERIALPORT_H

class SerialPort
{
public:
    SerialPort();
    SerialPort(unsigned char portNumber);
    ~SerialPort();

    void open(unsigned char portNumber);
    void close();

    unsigned char readByte();
    void writeByte(unsigned char data);
};

#endif
```

Wir haben hier das Analysemodell einfach um Datentypen ergänzt und die Methoden zum lesen und schreiben von einzelnen Bytes umbenannt.

## Implementation Klasse SerialPort, 1. Schritt

### Verwendete WIN32-API's

Für den Zugriff auf den seriellen Port verwenden wir die WIN32-API. Für den Programmierer ist die Anwendung kaum von normalen Datei-Handling zu unterscheiden, denn es werden folgende API's verwendet:

```
////////////////////////////////////

HANDLE CreateFile(
    LPCTSTR lpFileName,           // file name
    DWORD dwDesiredAccess,       // access mode
    DWORD dwShareMode,           // share mode
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // SD
    DWORD dwCreationDisposition, // how to create
    DWORD dwFlagsAndAttributes,  // file attributes
    HANDLE hTemplateFile         // handle to template file
);

////////////////////////////////////

BOOL ReadFile(
    HANDLE hFile,                // handle to file
    LPVOID lpBuffer,            // data buffer
    DWORD nNumberOfBytesToRead, // number of bytes to read
    LPDWORD lpNumberOfBytesRead, // number of bytes read
    LPOVERLAPPED lpOverlapped   // overlapped buffer
);

////////////////////////////////////

BOOL WriteFile(
    HANDLE hFile,                // handle to file
    LPCVOID lpBuffer,           // data buffer
    DWORD nNumberOfBytesToWrite, // number of bytes to write
    LPDWORD lpNumberOfBytesWritten, // number of bytes written
    LPOVERLAPPED lpOverlapped   // overlapped buffer
);

////////////////////////////////////

BOOL CloseHandle(
    HANDLE hObject // handle to object
);

////////////////////////////////////
```

Wir werden diese auf eine sehr beschränkte Art verwenden und nur die wenigsten der Möglichkeiten brauchen. Die Anzahl der Argumente zeigt aber, dass diese API's sehr mächtig sind und die Beschreibung dazu viele Seiten füllt. Die Beschreibung ist als SDK-Dokumentation frei verfügbar, aber häufig nicht installiert, da diese viel Platz auf der Harddisk braucht.

## Öffnen und Schliessen des seriellen Ports

Das Betriebssystem betrachtet den seriellen Port als eine Art Datei, in die man schreiben kann und aus der man lesen kann. Um eine Datei zu öffnen rufen wir `CreateFile` auf. Betrachten wir das Öffnen anhand unserer `SerialPort::open(unsigned char portNumber)` Methode.

```
void SerialPort::open(unsigned char portNumber)
{
    m_handle = CreateFile("COM1",
                        GENERIC_READ | GENERIC_WRITE,
                        0,
                        0,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        0);

    if(INVALID_HANDLE_VALUE == m_handle)
    {
        throw std::string("Konnte Port nicht öffnen");
    }
}
```

In dieser Version ignorieren wir das Argument `portNumber` und öffnen den ersten COM-Port.

Das erste Argument zur `CreateFile` API ist der Name der Datei als LPCTSTR (`const char*`). Der Com-Port 1 hat den Namen „COM1“! Das zweite Argument gibt an, wofür wir den Port öffnen möchten. Die Konstanten `GENERIC_READ` und `GENERIC_WRITE` sind vordefiniert und werden hier einfach mit dem logischen ODER kombiniert. Diese Werte sind so definiert, dass bei ihnen jeweils nur ein Bit der 32 möglichen Bits gesetzt ist. Diese Bits lassen sich einfach abfragen, so dass als Optionsflags ausgewertet werden können. Das nächste Argument gibt an ob die Datei mehrmals geöffnet werden kann, also ob mehrere Programme oder Programmteile die Datei gleichzeitig öffnen können, was aber beim seriellen Port nicht möglich ist. Das vierte Argument ist wieder ein Zeiger auf Sicherheitsattribute, die wir aber nicht verwenden und darum eine 0 übergeben. Das nächste Argument gibt an, dass wir eine bereits bestehende Datei öffnen möchten. Man könnte damit Dateien nach Bedarf erzeugen, was aber beim seriellen Port nicht möglich ist. Das zweitletzte Argument gibt dem Betriebssystem einen Hinweis, wie wir die Datei verwenden wollen. Auch hier verwenden wir das Attribut, das normalen Zugriff erlaubt. Das letzte Argument wäre ein Handle auf ein File-Template, genaueres steht in der SDK-Dokumentation.

Das Schliessen ist ziemlich einfach:

```
void SerialPort::close()
{
    if(INVALID_HANDLE_VALUE != m_handle)
    {
        CloseHandle(m_handle);
        m_handle = INVALID_HANDLE_VALUE;
    }
}
```

Wichtig ist nur, dass im Konstruktor das *m\_handle* Datenelement mit dem Wert `INVALID_HANDLE_VALUE` initialisiert wird:

```
SerialPort::SerialPort()
    :m_handle(INVALID_HANDLE_VALUE)
{
}
```

### Fehlerbehandlung

Zur Fehlerbehandlung verwenden wir Exceptions. Der Einfachheit halber hier nur strings, was nicht ganz optimal ist, aber immerhin ist es eine Klasse, die im Standard bekannt ist. Man sollte eigentlich mehr Gedanken an die Fehlerbehandlung verschwenden, denn aufschieben kann man das schlecht. Am Ende fehlt immer die Zeit den ganzen Code nach schlechter Fehlerbehandlung zu durchforsten. Wir werden versuchen das wenige das wir machen konsequent durchzuziehen und werfen im Fehlerfall einen *std::string*.

## Lesen und Schreiben

Das Lesen und Schreiben eines einzelnen Bytes wird uns nicht stark herausfordern, da wir die ReadFile und WriteFile API jetzt noch in der einfachsten Art verwenden.

Hier der Code zur Methode *SerialPort::writeByte()*

```
void SerialPort::writeByte(unsigned char data)
{
    if(INVALID_HANDLE_VALUE == m_handle)
    {
        throw std::string("writeByte : Port nicht offen");
    }

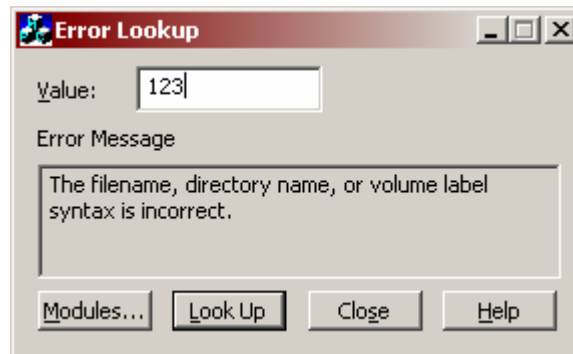
    DWORD bytesWritten = 0;
    BOOL result = FALSE;
    result = WriteFile(m_handle, // unser handle
                      &data,    // Zeiger auf Speicher
                      1,        // Anzahl Bytes
                      &bytesWritten, // Zeiger auf DWORD
                      0);      // Overlapped Struktur

    if(FALSE == result)
    {
        DWORD error = GetLastError();
        std::stringstream stream;
        stream << "WriteFile mislungen mit error code : ";
        stream << error;
        throw stream.str();
    }
}
```

Man übergibt der Funktion einen Zeiger auf den Speicher in dem sich die Daten befinden und gibt an wieviele Bytes geschrieben werden sollen. Die Funktion gibt als Rückgabewert einen BOOL zurück, der nur Erfolg oder Misserfolg anzeigt. In das zweitletzte Argument schreibt sie aber noch die Anzahl geschriebene Bytes, was wir hier aber nicht auswerten. Das Lesen sieht sehr ähnlich aus. Ich möchte darum auf die [Lösung](#) hinweisen, wo du den Code selber genauer untersuchen kannst.



Falls etwas schief geht setzt diese API-Funktion wie viele andere auch einen Fehlercode im System, der mit der GetLastError()-API ausgelesen werden kann. Ein Fehlercode lässt sich im allgemeinen mit dem Error-Lookup Tool (siehe Tools-Menu) nachsehen. Beispiel:



## Übung

### Öffnen des richtigen Ports

Das Beispiel oben öffnet nur den ersten Com-Port („COM1“). Wir wollen aber das Argument korrekt benutzen und müssen einen Weg finden, den `unsigned char` in einen `string` zu bringen. Ergänze den Code also, so dass bevor die `CreateFile` Funktion aufgerufen wird, ein `string` zusammengesetzt („COM“ + „X“) wird. Tipp : Das geht sehr einfach mit einem `std::stringstream`! Schau sonst den Code der Klasse `Zahl` an, der auch einen solchen `std::stringstream` verwendet. Siehe [www.devmentor.ch/teaching/additional/001/Semester5/Abend2/Zahl.zip](http://www.devmentor.ch/teaching/additional/001/Semester5/Abend2/Zahl.zip). Ein kleines Problem ergibt sich dadurch, dass wir als Argument einen `unsigned char` haben. Dieser wird nicht umgewandelt. Aber weise vorher diesen Wert einem `long` zu, und schicke diesen `long` in den stream.

### Ergänzen der Methoden `readByte` und `writeByte`

Ergänze die Klasse `SerialPort`, so dass sie ungefähr dem Zustand der [Musterlösung Version 1](#) entspricht.