Die Klasse Serialport, asynchrones Lesen und Schreiben

Ziel, Inhalt

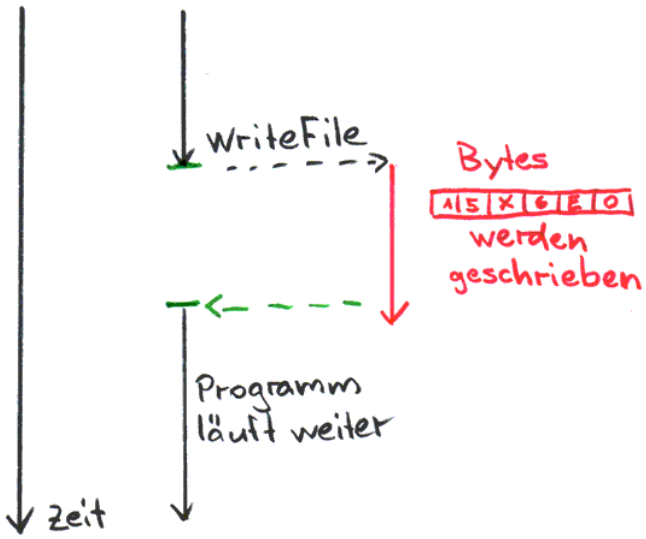
- Wir führen die Arbeit an unserer SerialPort Klasse fort. Wir bauen das Lesen und Schreiben so um, dass es auf einem getrennten Thread und asynchron ausgeführt wird

Die Klasse Serialport

Asynchrones Lesen und Schreiben

Synchrones Lesen und Schreiben

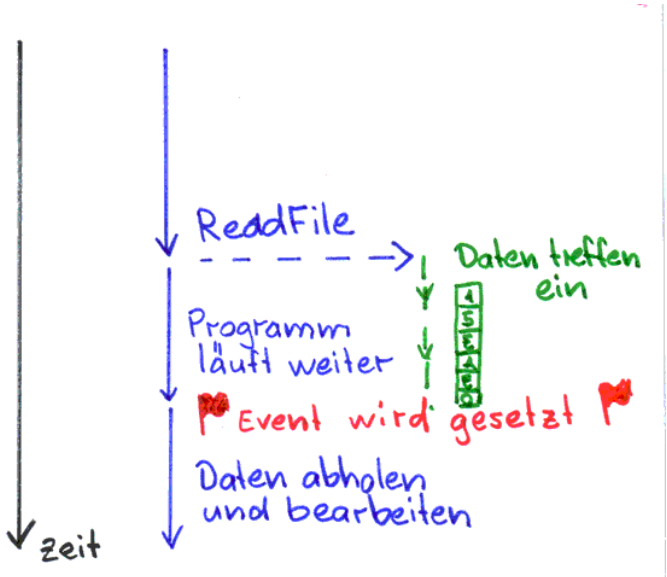
Beim synchronen Lesen und Schreiben dauern die WriteFile und die ReadFile API's so lange, bis die geforderte Anzahl Bytes entweder geschrieben oder gelesen wurde. Das kann beim seriellen Port manchmal lange dauern oder beim Lesen gar nie der Fall sein, wenn nichts am seriellen Port angeschlossen ist.



Das Schreiben kann je nach Medium eine gewisse Zeit dauern

Asynchrones Lesen und Schreiben

Beim asynchronen Lesen und Schreiben läuft das Programm nach dem Aufruf von WriteFile oder ReadFile weiter, ohne darauf zu warten, dass die angegebene Anzahl Bytes bearbeitet wurde. Wenn dann die Daten schlussendlich gelesen oder geschrieben wurden, kann ein Event, das vorher mit CreateEvent erzeugt wurde gesetzt werden. Siehe Abend 9: http://www.devmentor.ch/teaching/additional/01I/Semester5/Abend9/Threads.pdf



Sobald die angeforderten Daten angekommen sind, wird ein Event gesetzt

Asynchrones Lesen und Schreiben in der Praxis

Öffnen des Ports für asynchrones Lesen und Schreiben

Beim Öffnen des Ports mit CreateFile genügt es bei den Attributen zusätzlich das Flag FILE_FLAG_OVERLAPPED zu setzen.

```
_handle = CreateFile(comPort.c_str(),
           GENERIC_READ | GENERIC_WRITE,
           0,
           0,
           OPEN_EXISTING,
           FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
           0);
```

Das Flag wird mit dem bitweisen oder-Operator mit dem bereits vorhandenen Flag verknüpft.

Vorbereitungen zum Lesen und Schreiben

Beim Lesen und Schreiben müssen wir nun eine OVERLAPPED Struktur abfüllen und beim Aufruf mitgeben. In dieser OVERLAPPED Struktur verwenden wir nur das hEvent Element. Wir erzeugen ein vorher im Code eine Event mit CreateEvent. Dieses Flag wird nun gesetzt, wenn die Daten geschrieben oder gelesen wurden. Es genügt für uns auf dieses Event zu achten. Wir ergänzen am besten in unserer Klasse SerialPort die beiden Event-Handles als private Datenelemente.

```
private:
    HANDLE _handle;
    HANDLE _readFinished;
    HANDLE _writeFinished;
};
```

Als erstes werden die beiden im Konstruktor von SerialPort mit 0 initialisiert.

```
SerialPort::SerialPort()
:_handle(INVALID_HANDLE_VALUE),
 _readFinished(0),
 _writeFinished(0)
{
}
```

Wir ergänzen zwei Methoden, mit denen wir die Events erzeugen und wieder freigeben und schliessen.

```
void SerialPort::createEvents()
{
    // diese Methode nur aufrufen
    // wenn vorher die closeEvent
    // Methode aufgerufen wurde
    _ASSERT(_readFinished == 0);
    _ASSERT(_writeFinished == 0);

    _readFinished = CreateEvent(0, FALSE, FALSE, 0);
    _writeFinished = CreateEvent(0, FALSE, FALSE, 0);
}

void SerialPort::closeEvents()
{
    CloseHandle(_readFinished);
    CloseHandle(_writeFinished);
}
```

Für das _ASSERT Makro muss man <crtdbg.h> einbinden (#include). Die Methoden rufen wir nun in open und close auf.

Das Schreiben

Das Schreiben sieht nun so aus:

```
// unbedingt vorher createEvents aufrufen!
_ASSERT(_writeFinished);
OVERLAPPED ol = { 0 };

ol.hEvent = _writeFinished;

result = WriteFile(_handle,
                   &data,
                   1,
                   &bytesWritten,
                   &ol);

// wir warten hier darauf,
// dass die Daten geschrieben wurden.
WaitForSingleObject(_writeFinished, INFINITE);
```

Wir warten also nach dem Aufruf von WriteFile darauf, dass alle Daten geschrieben werden. Das wäre nicht nötig, dafür brauchen wir ja asynchrone Operationen nicht.
Asynchron ist für uns vor allem beim Lesen wichtig, denn so können wir eine bestehende Leseoperation unterbrechen und einen lesenden Thread, den wir noch erzeugen beenden.

Das Lesen

Das Lesen ist einiges aufwändiger. Zuerst bereiten wir alle Variablen und Strukturen vor. Danach senden wir den Lesen-Befehl. Wir prüfen das Ergebnis, das im Normalfall besagt, dass das Lesen **misslungen** ist. Das ist Normal, denn wenn keine Daten am Port anliegen, werden auch keine gelesen. GetLastError muss dann aber den Fehlercode ERROR_IO_PENDING zurückliefern. Darauf warten wir einfach, bis die gewünschten Daten (hier 1 Byte) bereit sind (mit getOverlappedResult).

```
unsigned char SerialPort::readByte()
{
    if(INVALID_HANDLE_VALUE == _handle)
    {
        throw std::string("Port nicht offen");
    }

    DWORD bytesRead = 0;
    BOOL result = FALSE;

    // unbedingt vorher createEvents aufrufen
    _ASSERT(_readFinished);
    OVERLAPPED ol = { 0 };

    ol.hEvent = _readFinished;
    unsigned char data = 0;

    result = ReadFile(_handle,
                      &data,
                      1,
                      &bytesRead,
                      &ol);
    if(FALSE == result)
    {
        // es kann sein, dass es einen Fehler
        // gab, nur weil keine Daten zum Lesen
        // bereit liegen
        DWORD error = GetLastError();
        if(ERROR_IO_PENDING != error)
        {
            std::stringstream stream;
            stream << "ReadFile misslungen mit";
            stream << " error Code : " << error;
            throw stream.str();
        }
    }
    // Wir warten bis die Daten hier sind
    WaitForSingleObject(_readFinished, INFINITE);
    // Hole das Ergebnis der overlapped (asynchronen)
    // Operation
    if(GetOverlappedResult(_handle, &ol, &bytesRead, FALSE))
    {
        if(1 != bytesRead)
        {
            std::string errorMessage("Unerwartete Anzahl Bytes");
            throw errorMessage;
        }
    }
    else
    {
        // das wäre wirklich unerwartet
        std::string errorMessage("GetOverlappedResult misslungen");
        throw errorMessage;
    }
    return data;
}
```

Ein lesender Thread

Thread in der Klasse SerialPort

Wie man einen Thread erzeugt wissen wir ja bereits vom Abend 9:
http://www.devmentor.ch/teaching/additional/01I/Semester5/Abend9/Threads.pdf
Unsere SerialPort Klasse sieht nun so aus:

```
class SerialPort
{
public:
    ~SerialPort();

    static SerialPort& getSerialPort(unsigned char portNumber);

    void writeByte(unsigned char data);

private:
    SerialPort();
    SerialPort(const SerialPort& c);
    void open(unsigned char portNumber);
    void close();

    void createEvents();
    void closeEvents();
    void startReadingThread();
    void stopReadingThread();

    bool readByte(unsigned char& data);
    static DWORD WINAPI ThreadFunction(LPVOID parameter);

private:
    HANDLE _handle;
    HANDLE _readFinished;
    HANDLE _writeFinished;
    HANDLE _thread;
    HANDLE _stopThread;
};
```

Die ReadByte Methode hat sich verändert, sie wird nun aus der ThreadFunction aufgerufen und darf sonst nicht mehr verwendet werden, da wir sie so umrüsten, dass sie entweder Daten liest oder beendet wird, falls das _stopThread Event gesetzt wird.

Die geänderte Methode ReadByte

```
bool SerialPort::readByte(unsigned char& data)
{
    if(INVALID_HANDLE_VALUE == _handle)
    {
        throw std::string("Port nicht offen");
    }

    DWORD bytesRead = 0;
    BOOL result = FALSE;
    // unbedingt vorher createEvents aufrufen
    _ASSERT(_readFinished);
    OVERLAPPED ol = { 0 };
    ol.hEvent = _readFinished;

    result = ReadFile(_handle, &data, 1, &bytesRead, &ol);
    if(FALSE == result)
    {
        // es kann sein, dass es einen Fehler
        // gab, nur weil keine Daten zum Lesen
        // bereit liegen
        DWORD error = GetLastError();
        if(ERROR_IO_PENDING != error)
        {
            std::stringstream stream;
            stream << "ReadFile misslungen mit";
            stream << " error Code : " << error;
            throw stream.str();
        }
    }
    // Array mit Events auf die wir warten
    HANDLE wartenAuf[] = { _readFinished, _stopThread };
    // Wir warten bis die Daten hier sind
    // oder der Thread beendet werden muss
    DWORD wait = WaitForMultipleObjects(2, // wir warten auf 2 Events
                                        wartenAuf, // array mit vents
                                        FALSE, // ein Event reicht
                                        INFINITE); // ewig warten
    if(WAIT_OBJECT_0 == wait)
    {
        // das erste Event ist eingetreten
        // es sind also Daten vorhanden
        // Hole das Ergebnis der overlapped (asynchronen)
        // Operation
        if(GetOverlappedResult(_handle, &ol, &bytesRead, FALSE))
        {
            if(1 != bytesRead)
            {
                std::string error ("Unerwartete Anzahl Bytes”);
                throw error;
            }
        }
        else
        {
            // das wäre wirklich unerwartet
            std::string error ("GetOverlappedResult misslungen");
            throw error;
        }
    }
    else
    {
        // etwas anderes ist eingetreten
        return false;
    }

    return true;
}
```

Die Methode ist schon sehr lang und gefällt mir nicht besonders, da sie das Lesen des Bytes nicht sauber vom Beenden der Leseoperation trennt. Vielleicht gelingt es uns zusammen die Sache sauberer zu machen.

Thread starten und beenden

Hier die Methode um den Thread zu erzeugen und zu beenden:

```
void SerialPort::startReadingThread()
{
    // zuerst den thread beenden, bevor
    // er erneut gestartet wird.
    _ASSERT(0 == _thread);
    _ASSERT(0 == _stopThread);

    _stopThread = CreateEvent(0,
                              FALSE,
                              FALSE,
                              0);
    DWORD threadId = 0;
    _thread = CreateThread(0,
                           0,
                           ThreadFunction,
                           this,
                           0,
                           &threadId);
}

void SerialPort::stopReadingThread()
{
    // rufe vorher startReadingThread auf!
    _ASSERT(0 != _thread);
    _ASSERT(0 != _stopThread);

    // wir setzen das Event, dass der Thread
    // beendet werden soll
    SetEvent(_stopThread);

    WaitForSingleObject(_thread, INFINITE);

    CloseHandle(_thread);
    _thread = 0;
    CloseHandle(_stopThread);
    _stopThread = 0;
}
```

Beachte auch wie ich versuche mit den _ASSERT Aufrufen den Programmierer und mich selber zu korrekter Anwendung der Methoden zu zwingen.

Die Thread Funktion

Als letztes nun die Thread Funktion.

```
DWORD SerialPort::ThreadFunction(LPVOID parameter)
{
    // beim Erzeugen des Threads
    // sollte ein gültiger Zeiger
    // auf ein SerialPort Objekt mitgegeben
    // werden.
    _ASSERT(0 != parameter);

    SerialPort* serialPort = (SerialPort*)parameter;

    unsigned char data = 0;

    try
    {
        while(serialPort->readByte(data))
        {
            // solange Daten gelesen werden können
        }
    }
    catch(std::string& error)
    {
    }

    return 0;
}
```

Wir müssen hier die ReadByte Methode mit einem try-catch Block umrahmen, denn falls eine Exception geschieht, müssen wir diese auf dem aufrufenden Thread fangen.
Exceptions bleiben immer auf dem Thread, auf dem sie erzeugt wurden. Werden sie nicht auf dem Thread selber gefangen, stürzt das Programm ab.