Die Klasse Serialport, Teil 3

Ziel, Inhalt

- Wir führen die Arbeit an unserer SerialPort Klasse fort. Wir setzen heute die Eigenschaften des seriellen Ports wie Baud-Rate und andere. Wir lernen heute auch wie man Klassen definiert, die auf Ereignisse in einer anderen Klasse reagieren.

Die Klasse Serialport

Anforderungen/Analyse Klasse *SerialPort,* 3. Schritt

Eigenschaften des seriellen Ports

Der serielle Port überträgt Daten auf zwei Leitungen durch elektrische Spannungsimpulse. Dabei ist es möglich die zeitliche Dichte der Spannungsänderung über die Baud-Rate zu ändern. Die übertragenen Datenimpulse ergeben ein Bitmuster wobei auch hier definiert werden kann ob jeweils 8 Bit ein Byte ergeben und ob ein weiteres Bit als Stop-Bit verwenden wird. Es gibt noch mehr Einstellungen, die du über die Systemsteuerung ansehen kannst. Wir wollen dem Anwender unserer Klasse eine einfache Möglichkeit geben die wichtigsten Einstellungen zu setzen. Als nächstes wollen wir ein Interface beschreiben, das es ermöglicht, auf Ereignisse, also auf eintreffende Daten am seriellen Port zu reagieren und benachrichtigt zu werden. Wir werden dabei das Publisher/Subscriber-Idiom anwenden.

Eigenschaften des seriellen Ports

Der serielle Port wird zwar wie eine Datei geöffnet, gelesen und geschrieben, kennt aber noch mehr Einstellungen. Diese Einstellungen sind über weitere API's erreichbar.

Timeout-Einstellungen

Es ist möglich beim COM-Port Timeouts für das Lesen und das Schreiben zu definieren. Damit ist es möglich die sonst blockierenden ReadFile und WriteFile API's, nach einer bestimmbaren Zeit zu unterbrechen. Diese beiden Funktionen heissen:

```
BOOL GetCommTimeouts(HANDLE, LPCOMMTIMEOUTS);
BOOL SetCommTimeouts(HANDLE, LPCOMMTIMEOUTS);
```

Wobei LPCOMMTIMEOUTS ein Zeiger auf eine COMMTIMEOUTS-Struktur ist. Wir werden aber bald unsere Dateizugriffe auf dem seriellen Port so abändern, dass wir asynchron Lesen und Schreiben. Dazu später mehr. Für uns bedeutet das im Moment, dass wir nachdem wir den Port geöffnet haben, die Timmeouts auf 0 setzen, was dazu führt das eine Lese- oder eine Schreiboperation erst beendet wird, wenn sie erfolgreich war.

```
COMMTIMEOUTS ct = { 0 }; // Struktur mit 0 initialisieren
BOOL ok = SetCommTimeouts(handle, &ct);
```

Weitere Einstellungen mit der DCB-Struktur

Für weitere Einstellungen existiert die DCB-Struktur. Diese Struktur enthält viele Elemente, von denen die meisten von uns von zweitrangiger Bedeutung sind. Wir setzen nur einige davon, damit wir sicher mit einem PC kommunizieren können, der die gleich Klasse zur seriellen Kommunikation verwendet. Damit wir nicht alle Elemente von Hand setzen müssen, holen wir und die aktuellen Einstellungen.

```
DCB dcb = { 0 };
BOOL dcbOk = GetCommState(handle, &dcb);

// ändern der gewünschten dcb-Einstellungen
dcb.BaudRate = CBR_57600;     // set the baud rate
dcb.ByteSize = 8;             // data size, xmit, and rcv
dcb.Parity = NOPARITY;        // no parity bit
dcb.StopBits = ONESTOPBIT;    // one stop bit

dcbOk = SetCommState(handle, &dcb);
```

Asynchrones I/O

Motivation

Wie oben erwähnt werden wir in unserer Klasse asynchones I/O verwenden. Der Grund ist vor allem im Lese-Operation, die wir auf einem eigenen Thread auslagern werden. Um dieses Thread irgendwann zu beenden wollen wir irgen ein Event setzen können und der Thread muss sofort darauf reagieren. Wir werden auf diesem Thread eine asynchrone Lese-Operation starten und darauf warten, dass entweder die Lese-Operation Daten liefert oder der Thread beendet werden muss. Wir könnten das Problem auch auf andere Art lösen, insbesondere mit den Timeout-Einstellungen.

Asynchrones Lesen und Schreiben

Wenn wir eine Datei zum asynchronen Lesen und Schreiben öffnen wollen müssen wir das beim Aufruf von CreateFile angeben. Das zweitletzte Argument muss dann noch mit dem Flag FILE_FLAG_OVERLAPPED ver-oder-t werden.

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

Bei den ReadFile und der WriteFile aufrufen müssen wir jetzt für das letzte Argument einen Zeiger auf eine gültige OVERLAPPED-Struktur angeben. Diese Struktur enthält das Element hEvent, das ein HANDLE für ein Event ist.

Dieses Event wird signalisiert, wenn die asynchrone File-Operation beendet wird. Wir können also eine WriteFile oder ReadFile-Operation starten und danach darauf warten, dass das hEvent HANDLE in der OVERLAPPED-Struktur signalisiert wird.