Beuth Hochschule für Technik Berlin

Bachelorarbeit

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN

University of Applied Sciences

WN Serial COM Port Tester

Erstprüfer: Prof. Dr. Voß
Zweitprüfer: Prof. Dr. Görlich

Eingereicht am
23. September 2013

Eingereicht von
Matthias Hansert 764369

Inhaltsverzeichnis

Kapitel 1

Einleitung

Gefordert ist ein Programm, das auf Wincor Nixdorf Mainboards und auf Erweiterungskarten(COM-Karten) implementierte RS-232/COM-Port-Hardware, die zugehörigen Hardware-Treiber und das BIOS, indirekt testet. Das Testtool wird in der Programmiersprache „C / C++“ entwickelt und wird auf folgenden Betriebssystemen implementiert:

- Windows XP SP3 x86, x64
- Windows 7 x86, x64
- WinPE

Unter den möglichen Fehlerfällen wird das Tool Kabel- und Stecker-Fehler(Wackelkontakt durch schlechten mechanischen Kontakt der Stecker und der Buchsen) erkennen. Elektrische Fehler, die sich bei der Übertragung über ein langes Nullmodemkabel in Form von Paritätsfehlern zeigen oder die durch Windows-interne Ressourcenprobleme(Shared Interrupt) hervorgerufen werden, werden erkannt.

Das Programm soll die Schwächen anderer Serial-Ports-Tools ergänzen, wie zum Beispiel die eingeschränkte Skriptfähigkeit und die Testautomatisierung. Diese aktuell verwendeten Tools unterstützen keine Zähler, keine Uhrzeit mit aktuellem Datum als Sendedaten und keinen Zufallsgenerator. Zudem können sie nicht von einen schreibgeschütztem Medium gestartet werden. Das Tool soll direkt über die Windows API die COM Ports testen, da die Kunden von der Firma Wincor Nixdorf und die Applikationen die COM Ports über diese API steuern. Ein weiterer Vorteil der Nutzung der Windows API ist, dass die Scripting- und Automatisierungsoptionen besser implementierbar sind. Durch die Voraussetzung der Nutzung der Windows API, ist die Installation oder Implementierung des Tools auf einem Linux-System nicht möglich.

Das Programm besteht aus einer GUI, in der alle Programm- und Schnittstellenparameter (Baudrate, Stoppbits, Paritätsbit, Datenbits, Hardware und Software) eingestellt werden können. Wenn das Programm über die Kommandozeile oder ein Skript gestartet wird, muss eine Konfigurationsdatei oder die Konfigurationsparameter übergeben werden. Wie diese Datei strukturiert sein muss wird im Kapitel 6 behandelt.

Kapitel 2

Aufgabenstellung

Das Tool soll die RS-232 Schnittstelle testen und die Schwächen der vorhandenen Lösungen ergänzen. Die Firma Wincor Nixdorf besitzt derzeit schon ein Tool welches die Druckertestumgebung testet, dies ist aber auf Ruby basierend. Das Programm „WN Serial COM Port Tester“ soll mit Funktionsaufrufen der WinAPI programmiert werden. Die aktuell vorhandenen Lösungen, wie zum Beispiel das Programm „AGGsoft Serial Port Tester“ haben eingeschränkte Einstellungen und sind nicht ausführlich genug für die Voraussetzungen der Qualitätssicherung.

2.1 Module

Das Programm bietet dem Nutzer folgende Schnittstellen an um es zu bedienen:

- Windows-GUI(Graphical User Interface) mit allen Einstellmöglichkeiten und Start-Button.
- Kommandozeilen User Interface mit passenden Einstellmöglichkeiten, wie auch per GUI und Rückgabewerte (Exitcodes).
- Funktionskern, der aus beiden UI angesprochen wird und die eigentlichen Tests durchführt.

2.2 Laufzeit

Das Com-Port-Test Programm kann von einem schreibgeschütztem Medium (z.B. Windows-PE-CD) laufen, d.h. alle Schreibfunktionen (Log-Datei) sind per Option abschaltbar. Die Testparameter können in einer Konfigurationsdatei gespeichert und wieder geladen werden. Alle Ereignisse und Meldungen können in einer Log-Datei eingetragen werden.

Wenn das Programm mit einem Skript ausgeführt wird startet das Com-Port-Test Programm und testet die Ports automatisch. Unter Ports sind die im die BEETLE[1] Systeme eingebaute COM Ports zu verstehen. Solche Ports sind Legacy oder durch PCI erweiterbare Ports.

2.3 Hardware

Folgende Hardware-Kombinationen werden unterstützt (die Reihenfolge hier gibt die Priorität der Implementierung wieder):

[1] Wincor Nixdorf Kassensystem

1. Test einer Schnittstelle mit Kurzschlussstecker auf einem System
2. Test zweier Schnittstelle mit Null-MODEM-Kabel (RxD/TxD gekreuzt; RTS/CTS gekreuzt, etc.) auf einem System
3. Test zweier Schnittstelle mit Null-MODEM-Kabel auf unterschiedlichen Systemen! Siehe dazu auch „Synchronisierungsprotokoll“...

2.4 Betriebssystem

Das Programm "WN Serial COM Port Testerïst für folgende Zielbetriebssysteme entwickelt worden:

- Windows XP SP3 x86, x64; Windows 7 x86, x64; WinPE 2.x (WinXP) / 3.x (Win7)
- Windows 8 nur ein Test, der aussagt ob es kompatibel ist oder noch etwas zu tun wäre

Linux - Distributionen werden vom Programm nicht unterstützt.

2.5 Logbuch

Alle Ereignisse und Testschritte können zu Dokumentations- und Analysezwecken in einer Log-Datei gespeichert werden. Auch Fehler, die der Parser in der Test-Datei(.ini) findet, werden in der Log-Datei eingetragen. Der Name der Log-Datei ist für jede Instanz und jedes Testsystem eindeutig zu wählen, wie im folgenden Beispiel zu sehen ist:

<Programmname>_<Computername>_<Master/Slave>_<Port>.csv

Jeder Eintrag bekommt einen Uhrzeit und Datums-Stempel. Eine Zeile sieht dann wie folgt aus:

<Datum>; <Uhrzeit>;<Ereignis>;<Wert>;<Kommentar>

Die ersten beiden Einträge der Log-Datei:

<Datum>; <Uhrzeit>;START;RS232-Port-Nummer; INI-Dateiname als Kommentar

2.6 Programminstanzen

Das Programm kann in mehrfachen Programm-Instanzen gestartet werden, in der Regel läuft bei umfangreichen Tests für jeden zu testenden RS-232-Port eine Programm-Instanz. Das vermeidet aufwändige Thread-Programmierung eines einzigen Programms für alle Ports. Das ist keine feste Vorgabe, der Entwickler kann entscheiden welcher Lösungsweg als geeignet befindet.

2.7 Master / Slave

Jede Programminstanz wird als Master, Slave oder mit beiden Konfigurationen gestartet.

2.7.1 Master-Funktion

Im Master-Modus schickt die Schnittstelle die vorgegebenen Texte / Dateien mit den eingestellten Parametern an den Slave. Der Master wertet die dem Test zugeordnete INI-Datei aus, der Slave folgt nur dem Master. Der Master wartet auf das Echo oder auf eine Fehlermeldung und bewertet den Testschritt sobald der Test komplett und fehlerfrei empfangen worden ist. Zugleich bewertet er ob der Test erfolgreich oder fehlerhaft war. In beiden Fällen werden Fehler, Warnungen und Erfolgsmeldungen geloggt.

2.7.2 Slave-Funktion

Im Slave-Modus (Programminstanz auf gleichem System oder zweiten BEETLE/PC)werden die Daten komplett empfangen, wertet eventuelle Fehler aus und schickt entweder die Daten als Echo oder eine Fehlermeldung wieder zurück.

2.7.3 Master / Slave Kombination mit Kurzschlussstecker

Im kombinierten Master / Slave-Modus (Kurzschlussstecker) übernimmt eine Instanz des Programms beides, das heißt, sie schickt sich selber Daten und antwortet ebenso. Programm intern laufen aber die beiden Module (Master / Slave) weitestgehend getrennt ab.

2.7.4 Master / Slave Ablauf mit „Wobbeln“-Funktion

Die Master-Instanz übernimmt die Steuerung des Ablaufes, der Slave antwortet dem Master. In diesem Testmodus werden durch den Benutzer eingegebene Baudraten(Unter und Obergrenze) gewobbelt. Der Wobbeln-Ablauf kann über die GUI oder in der INI-Datei beschrieben werden. Der automatische Test erfolgt dann nur für die dort eingetragenen Parameter, z.B.:

BAUD = 9600 - 115200

Hier werden die Baudrate von 9600 bis 115200 für den Test berücksichtigt, aber 2400, 300, etc. weiter nicht. Wenn mehrere Parameter flexibel gesetzt / programmiert sind, werden alle Kombinationen getestet. Das kann dann durchaus eine nicht unerhebliche Testzeit beanspruchen.

Die Programminstanz öffnet die zugehörige Schnittstelle automatisch mit der ersten von allen der programmierten Optionen (Baudrate, Parität, etc.) und startet dann einen Testdurchlauf mit diesen Einstellungen. Wenn alle Daten fehlerfrei gesendet und danach auch das Echo fehlerfrei empfangen wurden, wird der Port geschlossen, mit der nächsten Kombination geöffnet und wieder Daten gesendet und empfangen. Wenn Fehler (Parität, Offline, etc.) vom Master erkannt oder vom Slave gesendet wurden, dann wird der letzte Schritt wiederholt. Jedes Lese- oder Schreibvorgang wird fünf mal versucht. Wenn der aktuelle Test schon die x-te Fehler-Wiederholung war, dann stoppt das Programm und zeigt den Fehler an. Dieser Fehler-Zustand darf durch das Programm nicht verändert werden. Der Tester / Entwickler muss die Fehler analysieren.

2.7.5 Synchronisierungsprotokoll

Für den Wobbelmodus ist ein minimales Protokoll zur Synchronisation nötig. Wenn eine Programminstanz als Master gestartet wird, erhält sie die nötigen Schnittstellenparameter per GUI

oder aus der Konfigurationsdatei. Sie sendet diese in From eines Konfigurationspakets dem Slave. Solche Konfigurationspakete werden immer mit den Standardschnittstellenparametern (96,o,8,1) gesendet. Der Slave synchronisiert sich dann, d.h. programmiert -per WIN-API- seinen zu testenden Port mit den empfangenen Parametern und wartet auf Nutzdaten.

Wenn der erste Test erfolgreich war (Master hat gesendet und der Slave hat ein Echo geschickt), dann schickt der Master an den Slave die Parameter des nächsten Testschritts. Das ist nötig, weil im Fehlerfall (Echo fehlerhaft empfangen) der Slave nicht mehr synchron wäre, er weiß nur ob seine Daten fehlerfrei gesendet wurden, aber nicht ob der Empfang richtig angekommen ist. Der Slave kann so einfach nur mit Standard-Parametern gestartet werden und folgt dann dem Master.

2.8 Teststruktur

Gefordert ist eine einfache Trennung von Testablauf / Testoptionen und Testobjekt(RS-232-Port). Die Teststeuerung wird daher sowohl im Programm selber, als auch über die Kommandozeile implementiert. Der Testablauf für jeden Port selber ist in der zugehörigen Konfigurationsdatei vollständig, aber unabhängig vom zu testenden Port (RS-232-Port / V.24-Schnittstelle), beschrieben. Es können mehrere Konfigurationsdateien auf der Kommandozeile übergeben werden. Die Schnittstellenummer wird über die Kommandozeile festgelegt. So kann der gleiche Testablauf (INI-Datei) für jeden Port einzeln oder parallel durchgeführt werden.

2.9 Benutzerschnittstelle

2.9.1 GUI

In der GUI sind alle Programm- und Schnittstellenparameter einstellbar und können jederzeit aus der GUI in eine Konfigurationsdatei gespeichert werden. Das Windows-Fenster bietet also implizit auch die Funktion eines Konfigurationseditors mit eingebauter Syntaxüberprüfung.

2.9.2 Kommandozeile

Wenn die Kommandozeile keine Parameter bekommt, wird automatisch die GUI dargestellt. Die Kommandozeile nimmt folgende Optionen entgegen:

- -f INI_Datei
- -f INI_Datei /Port

In der INI Datei ist der Testvorgang beschrieben. Wenn kein "/Port“ angegeben wurde, dann wird der Port in der INI Datei getestet. So sieht ein Programmaufruf aus:

- SerialPortTester.exe KOMPLETT.INI
- SerialPortTester.exe KOMPLETT.INI /COM3

2.10 Skriptfähigkeit

Per Kommandozeile gestartet, ist das Programm BATCH-/Script-fähig, d.h. es startet ohne weitere Benutzereingaben, loggt alle Ereignisse und auch Hinweise -zur Analyse nach dem Test- in die zugehörige Datei und beendet sich ebenso selbstständig. Fehler, Warnungen und Hinweise werden auf der Konsole ausgegeben. Als Option kann es auch im Hintergrund ohne jede Anzeige laufen.

2.11 Programmierung der Testparameter

Die Testparameter können direkt per GUI oder per INI-Datei eingestellt werden. Die aktuelle Testkonfiguration und der Ablauf kann in einer INI-Datei gespeichert und wieder geladen werden. Eine mit GUI erstellte INI-Datei kann über die Kommandozeile geladen werden. Diese INI-Datei hat die Windows-übliche Struktur mit „[“ und „]“ und ist per Notepad (oder anderen Editor) editierbar.
Sie enthält alle Informationen zur Steuerung der Programminstanz, dass heißt alle einstellbaren Schnittstellenparameter inklusive Wobbelnoption (von-bis):

Baudrate = 1200-9600; Stopbbits = 1; Datenbits = 8; RTS_CTS=yes; etc.
Baudrate = min-max; Stopbbits = min-max; etc.

Für MIN/MAX siehe Schnittstellenoptionen. Timeout-Werte (für Echo-Antworten auf Datenpakete) sollten keine festen Zeiten programmiert sein, das Programm berechnet die Wartezeiten automatisch aus der aktuellen Baudrate. Verzögerungszeiten zwischen zwei Datenblöcken/Testschritten inklusive Zufallswert ist wünschenswert.

2.11.1 Schnittstellenoptionen zu Test

Folgende Werte sollen für den Benutzer einstellbar sein:

- Baudrate
- Parität
- Datenbits
- Stopbits
- Flusssteuerung

Das Programm kann hier mit Variablen MIN und MAX umgehen. Wenn beispielsweise die Zeile „BAUDRATE = MAX“ oder „BAUDRATE = MIN“ programmiert wurde, sucht das Programm die maximale bzw. minimale Baudrate, die angegebene RS-232-Port unterstützt und nutzt sie für den Test. Diese Werte werden aus „dem System“ –beispielsweise Registry oder MSPORTS.DLL direkt ausgelesen. Manuell können diese MIN/MAX-Werte über den Gerätemanager ermittelt werden. Wenn keine Parameter übergeben wurden, startet das Programm mit diesen Standardparametern: 9600,odd,8,1,RTS/CTS (von der Wincor Nixdorf Anzeige BA66).

Die Wobbelnoption wird in Form einer VON-BIS-Zeile implementiert:

BAUDRATE=300;2400;9600;115200	diese vier Werte werden getestet
BAUDRATE=MIN-MAX	alle möglichen Werte
BAUDRATE=300-9600;115200	alle Werte von 300 bis 9600 und 115200 im Test

2.11.2 Sendedaten aus Datei

Als Testdaten werden feste Texte (in der Regel ASCII Zeichen) oder auch Pattern aus einer Datei gesendet. Die Dateipfade / Sendetexte sind als Testparameter in der GUI oder Test-Datei) einstellbar. Wenn keine Sendedaten übergeben wurde, dann sendet das Programm zufällige, aber auf einem Protokollanalysator lesbare, ASCII-Zeichen: 0x20 ... 0x7F .

2.12 Fehlererkennung und Behandlung

Folgende mögliche Fehlerfälle muss ein solches Tool erkennen:

- Kabel-/Stecker-Fehler („Wackelkontakt“ durch schlechten mechanischen Kontakt Stecker/-Buchse)
- Elektrische Fehler, die sich bei Übertragung über langes Null-MODEM-Kabel im Form von Paritäts-, Rahmenfehlern oder ähnliches zeigen
- Dauerhaft gezogene Kabel / Kurzschlussstecker
- Windows-interne Ressourcenprobleme (Shared Interrupts)

Bei Erkennung eines Übertragungsfehlers wird dieser in die Logdatei eingetragen und / oder auf der Konsole angezeigt und danach der letzte Block wiederholt. Nach der x-ten erfolglosen Wiederholung (Wert aus Test-Datei) erfolgt Testabbruch

2.13 Neue Anforderungen

2.13.1 Logbuch

Der Name der Logdatei soll um eine ID-Nummer erweitert werden. Wenn eine Logdatei auf dem Zielsystem schon vorhanden ist, wird diese Nummer hochgezählt.

<Programmname>_<Computername>_<Master/Slave>_<Port>_<Nummer>.ini

Wenn der Test direkt über die GUI konfiguriert wurde, dann werden diese Werte in einer INI-Datei vor dem Teststart gesichert und diese in die Logdatei eingetragen.

2.13.2 Synchronisierungsprotokoll

Das erste solcher Konfigurationspakete wird immer mit den Standardschnittstellenparametern (96,o,8,1) gesendet. Ein solches Paket beginnt mit einem Fluchtzeichen (in ASCII wäre das ESCAPE 0x1B) und wird vom Slave per Bestätigungszeichen (in ASCII ACK) beantwortet.

2.13.3 Anhalten eines Tests

Möchte der Benutzer ein Test vorzeitig beenden, gibt es eine Schaltfläche in der GUI die den Test anhält. Wird per Kommandozeile das Programm ausgeführt, kann der Benutzer durch drücken der „ESC“ Taste ein Test stoppen.

Kapitel 3

Fachlichesumfeld

Die Quellen dieses Kapitel sind aus dem Wikipedia Artikel [Wik13], aus der MSDN Online Bibliothek [Mic95], aus den Büchern „C Programmers Guide to Serial Communications“ [Cam94], „Programming Windows“ [Pet98], „Visual C++ 2010“[Lou10] und „V 24 / RS-232 Kommunikation“ [Cam90].

3.1 RS-232 (Radio Sector 232)

Der RS-232 ist ein Standard der in den 60er Jahren von dem US-amerikanischen Standardisierungskomitee Electronic Industries Association(EIA) bearbeitet?? (entwickelt) und definiert wurde. Bei diesem Standard handelt es sich um eine serielle Schnittstelle, die für die serielle Kommunikation zwischen Rechnern und Modems für eine Punkt-zu-Punkt Verbindung (über Telefonleitungen) entwickelt wurde.

3.1.1 Definition

Der RS-232 Standard definiert eine Verbindung zwischen einer Datenendeinrichtung (DEE, zum Beispiel das Terminal) und einer Datenübertragungseinrichtung (DÜE, zum Beispiel einem Modem) und dessen Parameter. Unter den einzelnen Parametern sind folgende Eigenschaften zu verstehen:

1. Spannungspegel
2. Übertragungsprotokoll (Handshake)
3. Stecker
4. Timing

Die Übertragung ist Bit-seriell, das heißt, die Bits werden hintereinander in einer Datenleitung verschickt. Eine bestimmte Menge an Bits entspricht einem Zeichen. Ein Zeichen besteht aus einem Startbit, aus vier bis acht Datenbits (Nutzdaten), aus ein oder zwei Stopbits und der Parität. Die Parität ist in dem Standard nicht definiert, aber wird benutzt um Fehler zu erkennen und zu beheben. Im Allgemeinen ist die Parität nebensächlich, da sie aber in diesem Fall der Arbeit relevant ist wird sie im späteren Verlauf des Kapitels näher erklärt.

3.1.2 Übertragung

Das Startbit meldet dem Empfänger das die Übertragung anfängt. Die Start- und Stopbits haben inversen Pegeln. Ist eine Leitung im Ruhezustand oder hat die Stopbits erhalten, wird durch die Ankunft einer inversen Signalflanke der Empfänger aufmerksam gemacht, dass Nutzdaten ankommen werden. Bei der Ankunft eines Startbits tastet der Empfänger die Nutzdaten mit seiner Bitrate (Bits pro Sekunde) ab. Die Nutzdaten sind die Bitdarstellung von einem ASCII Zeichnen. Die Länge der Daten sind einstellbar, dabei kann die Übertragung von vier bis acht Bits pro Zeichen eingestellt werden. Im Anschluss kommt mindestens ein Stopbit, es können auch 1,5 Bits oder zwei Bits eingestellt werden. 1,5 Bits sind sehr ungewöhnlich, aber damit ist gemeint, dass die Mindestdauer der Pause zwischen zwei ankommenden Zeichen 1,5 Bitzeichen entspricht.

Im Folgenden wird die Übertragung von einem Zeichen näher erläutert: Wird ein Zeichen 'z' übertragen, mit einem Startbit, acht Datenbits und zwei Stopbits, ist die Gesamtlänge der Übertragung elf Bits lang. 'z' entspricht laut ASCII Kodierung den Wert 122 dezimal und „7A“ hexadezimal(0x7A). Die Bitdarstellung für 0x7 lautet „0111“ und für 0xA „1010“, zusammengesetzt ergibt sich für 0x7A = „01111010“. Die Übertragung in diesem Standard folgt dem LSB(Less significant Bit) zuerst, also werden die Bits vertauscht, demnach ergibt sich „0101 1110“. Davor wird eine Null hinzugefügt als Startbit und am Ende zwei Einsen als Stopbits. Demnach ergibt sich für die Übertragung des Zeichen 'z' folgenden Bitreihenfolge „0 0101 1110 11“, die versendet wird.

Die Datenübertragung unter RS-232 ist asynchron, das bedeutet es existiert kein gemeinsamer Takt. Die Bitraten zwischen Sender und Empfänger dürfen um wenige Prozent von einander abweichen, sonst wird der Empfänger das Wort zu schnell / langsam abtasten und falsch interpretieren. Dagegen muss die Baudrate bei Sender und Empfänger genau gleich sein. Beide Begriffe sind nicht zu verwechseln. Eine Bitrate definiert die übertragenen Bits pro Sekunde und die Baudrate die übertragenen Symbole pro Sekunde, wo jedes Symbol als definierte messbare Signaländerung im physischen Übertragungsmedium definiert ist. Da die Übertragung im diesem Standard binär ist, ist ein Symbol als ein Bit definiert. Dies hat zu Folge, dass im diesem Spezialfall der Übertragung die Bitrate und die Baudrate gleich sind. Bei anderen Übertragungsarten unterscheiden sie sich jedoch.

Die Baudrate ist vom Benutzer frei wählbar. Jedoch hat der Anwender die Kabellänge, den Leitungswiderstand und die Kapazität des Kabels zu beachtet. Je länger das Kabel, desto starker nimmt die Spannung ab. Nach Erfahrungswerten von Texas Instruments ist bei einer Baudrate von 9600, eine maximale Kabellänge von 152 Meter möglich. Bei 115200 muss das Kabel kürzer als 2 Meter sein.

Der Empfänger muss die Datenübertragung anhalten können, wenn er keine Daten mehr verarbeiten kann. Dieser Handshake wird softwaretechnisch oder über die Hardware mit Steuerleitungen realisiert. Bei der Softwarelösung werden am Sender spezielle Steuerzeichen gesendet. Dieses Protokoll ist als „Xon/Xoff“ bekannt. Es ist nur möglich dieses Protokoll zu benutzen wenn die Steuerzeichen(Xon = 0x11 und Xoff = 0x13) nicht in den Nutzdaten vorkommen. Bei der Hardwarelösung signalisieren Sender und Empfänger sich gegenseitig ihren Status. Solche Protokoll besteht zum Beispiel aus fünf Steuerleitungen(TxD, RxD, GND, RTS und CTS). Damit triviale Fehler vermieden werden, müssen Sender und Empfänger die gleichen Einstellungen haben. Das heißt dass die Baudrate, Stopbits, Parität, Handshake und Kabellänge übereinstimmen müssen.

3.1.3 UART

Universal Asynchronous Receiver Transmitter(UART) ist eine elektronische Schaltung zur Realisierung von digitalen seriellen Schnittstellen(für diese Arbeit der sogenannte COM Port). Der UART ist zum Senden und Empfangen von Daten über eine Datenleitung vorgesehen. Im industriellen Bereich ist der UART unter dem RS-232 Standard sehr verbreitet. Auch bei Wincor Nixdorf wird diese Schnittstelle zum steuern der COM Ports benutzt.

3.1.4 Paritätsbit

Wie schon erwähnt, ist die Parität nicht im RS-232 Standard definiert, aber sie ist für diese Arbeit relevant. Das Paritätsbit dient zur Erkennung fehlerhafte Übertragungen. Die Parität kann gerade(even) oder ungerade(odd) sein und wird durch die Anzahl an Einsen in einer Bitfolge bestimmt. Ist eine gerade Parität festgelegt, wird bei einer gerade Anzahl an Einsen eine Null angehängt, bei ungerade Anzahl eine Eins. Genau das Gegenteil geschieht bei ungerader Parität. Nach unserem Beispiel mit 0x7A wird bei gerader Parität eine Eins („0 0101 1110 **1** 11“) angehängt.

3.1.5 Leitungen und Stecker

Am Anfang wurde der 25-polige D-Sub-Stecker verwendet. Viele dieser 25 Leitungen sind reine Drucker und Terminal-Steuerleitungen aus der elektromechanischen Zeit, somit sind sie für die modernere Peripherie überflüssig. So hat sich der 9-polige D-Sub-Stecker(COM Port) etabliert. Dieser Stecker war nicht ursprünglich für diesen Standard gedacht, sondern wurde von IBM als Notlösung in einem anderen Standard entwickelt, um Platz zu sparen. Der Stecker ist daher unter EIA/TIA-574 zu finden. Für die EIA-232-Datenübertragung werden selten andere Stecker benutzt. Der 9-polige D-Sub-Stecker besteht aus folgenden Leitungen:

TxD, TX, TD	: Transmit Data, Leitung für ausgehende Daten
RxD, RX, RD	: Recieve Data, Leitung für ankommende Daten
RTS	: Request To Send, Sendanforderung
CTS	: Clear To Send, Sendeerlaubnis
DSR	: Data Set Ready, Einsatzbereitschaft
GND	: Ground, Signalmasse
DCD, CD, RLSD	: Data Carrier Detect, Erkennung einlaufende Daten
DTR	: Data Terminal Ready, Datenendeinrichtung ist bereit
RI	: Ring Indicator, Datenverbindungsaufbau

EIA-232 ist eine Spannungsschnittstelle, also werden die logische Null und Eins durch positive oder negative Spannungen vertreten. Die Datenleitungen (TxD und RxD) benutzt eine negative Logik. Spannungen zwischen -3V und -15V repräsentieren eine logische Eins. Signale zwischen -3V und +3V gelten als nicht definierte Signale. Die logische Null wird als eine Spannung zwischen +3V und +15V interpretiert. Beim Empfänger, wird ein Signal zwischen +3V und +15V auf den Steuerleitungen als aktiv betrachtet, und inaktiv zwischen -3V und -15V. Beim Sender üblicherweise ±12V.

Um Sender und Empfänger zu verbinden, gibt es verschiedene Kabelvarianten, abhängig vom Sender und vom Empfänger. Verbindet man ein Rechner (in der Regel mit einem Stecker) zu einem Modem (mit einer Buchse) ist ein 1:1 Kabel nötig. Sind zwei Rechner mit einander Verbunden, so ist die Rede von einem Nullmodemkabel, bei dem die Leitungen gekreuzt sind. Durch ein

Loopback-Stecker bzw. Kurzschlussstecker wird die Sendeleitung direkt and die Empfangsleitung der gleichen Schnittstelle umgeleitet. So ein Stecker wird für die Entwicklung und deren Tests von Kommunikationsanwendungen und Hardware (UART) benutzt.

3.1.6 Verwendung der RS-232 Schnittstelle bei Wincor Nixdorf

Die aktuellen BEETLE Systeme haben ein integrierten UART im Chipsatz. Diese COM Ports werden „Onboard Ports“ genannt und sind „Legacy Devices“. Je nach System sind zwei bis sechs COM Ports eingebaut. Wenn ein Kunde weitere Schnittstellen benötigt, kann er diese über den PCI Bus die Anzahl an Ports erweitern. Diese Erweiterung findet durch eine Sunix PCI Karte mit vier oder acht COM Ports statt. Der Kunde kann auch zwei Sunix PCI Karten einbauen.

Eine andere Erweiterungsmöglichkeit sind „ITE“ COM Ports. Diese Ports sind direkt am PCI Bus angeschlossen und im Mainboard eingebaut. Die ITE COM Ports wurden eingebaut, weil der Super I/O in älteren Mainboards nur zwei Legacy COM Ports verwalten konnten. Um mehrere „On Board“ Ports dem Benutzer zur Verfügung zu stellen wurde der ITE Chip eingebaut. Modernere Mainboards sind in der Lage mehr als zwei COM Ports zu verwalten, und deswegen gibt es jetzt bis zu sechs Legacy Ports plus Erweiterungsmöglichkeiten.

Die vorgestellten Schnittstellen sind für das Test Tools relevant. Es gibt noch ein anderen COM Port, der nicht in allen BEETLE Systemen vorhanden ist. Einige BEETLE Systeme haben ein iAMT (Intel Active Managment Technologie)[1] Chip. Dieser Port ist ein virtueller Port. Durch ein Treiber (SOL, Serial Over LAN) wird der Datenverkehr von der seriellen Schnittstelle auf das LAN umgeleitet. Da es ein virtueller Port ist, wird es nicht möglich sein, diesen Port zu testen. Die genauere Verwendung von iAMT und SOL ist für diese Arbeit nicht relevant, für mehrere Informationen wenden Sie sich an die angegebene Quelle.

Die BEETLE Systeme benutzen die COM Ports für verschiedene Zwecke. Für die Kassensysteme, werden an den COM Ports die Scanner und Kundenanzeigen angeschlossen. So kann das BEETLE System die beiden Peripheriegeräte mit Strom versorgen und Daten senden und empfangen. Es werden auch noch Drucker an die COM Ports angeschlossen. Bei ältere BEETLE Systeme wurden die Touchkomponenten für Bildschirme an die RS-232 Schnittstelle angeschlossen. Der Bildschirm war an eine Videoschnittstelle angeschlossen und die Touchsignale wurden über ein COM Port empfangen. Im Banking Bereich werden viele verschiedene Peripheriegeräte pro System über die RS-232 Schnittstelle gesteuert und mit Strom versorgt.

[1] http://software.intel.com/en-us/articles/using-intel-amt-serial-over-lan-to-the-fullest; 30.08.2013

3.2 Microsoft Windows API

3.2.1 Definition

Die Microsoft Windows API „Application Programming Interface“(Schnittstelle zur Anwendungsprogrammierung) ist ein Programmteil, dass vom Windows Betriebssystem den Benutzern und vor allem Entwicklern angeboten wird, um Programme an das Betriebssystem anbinden zu können. Ein Betriebssystem (Microsoft Windows, Mac OS, Linux, etc.) ist für Entwickler und Programmierer durch die API definiert. Somit kann eine Applikation über die API alle Funktionsaufrufe ausführen, die ein Betriebssystem anbietet. Nicht nur Funktionen sind in einer API definiert, sondern bestimmte Datenstrukturen und Datentypen durch das Kommando *typedef* wie *LRESULT* oder *CALLBACK*.

Mit fast jedem neuen Microsoft Betriebssystem wird die Windows API erweitert und abgeändert. Die erste API, bekannt als *Win16*, für die 16-Bit Versionen von Microsoft Windows. Für Windows 1.0 hatte die API etwa 450 Funktionsaufrufe. Bei der Zeit von Windows 98 wurde die API auf 32-Bit und mehrere tausende Funktionsaufrufe. Ab Windows XP „x64 Edition“ und Windows Server 2003 wurde die API auch auf 64-Bit erweitert.

Der hauptsächlichen Unterschiede zwischen den 16, 32 und 64 Bit Versionen von der API entstanden durch die verschiedenen Speicher und Prozessorarchitekturen. Unter der 16-Bit Architektur war die Registergröße 16 Bit groß, diese wurde bei den Prozessoren von Intel 8086 und 8088 eingesetzt. In der 32-Bit Architektur, 32 Bit groß bzw. in der 64-Bit, 62 Bit groß. Die Windows API ist in der Programmiersprache „C“ geschrieben. Deswegen war unter der 16-Bit Architektur der Datentyp *int* „nur“ 16 Bit lang(Zahlen von -32.768 bis 32.767) . In der Speicherverwaltung bestanden Speicheradressen aus einen 16-Bit Segment und einen 16-Bit Offset, der als Zeiger verwendet werden konnte. Für Programmierer war diese Verwaltung sehr Umständlich, da der Programmierer genaue Unterscheiden musste, zwischen *long* oder *far* und *short* oder *near* Zeiger.

Ab der 32-Bit Architektur entstand das „Flat Memory Model“, wo der Prozessor direkt die gesamte Speicheradressen ansprechen konnte, ohne Speichersegmentierung oder Pagingschemas. Somit wurde auch der *int* Datentyp auf 32 Bitgröße (Zahlen von -2.147.483.649 bis 2.147.483.647) definiert. Programme, die auf einem System mit einer 32-Bit Architektur geschrieben worden sind, benutzen einfache Zeigerwerte um direkt die Speicheradresse ansprechen zu können. Bei der Umstellung von 16-Bit auf 32-Bit blieben viele Funktionsaufrufe gleich, aber manche brauchten eine Umstellung auf 32-Bit, wie zum Beispiel das graphische Koordinatensystem für die GUI Darstellungen.

Aus Kompatibilitätsgründe sind die API’s Rückwärts kompatibel. Die Kompatibilität entsteht durch eine Übersetzungsschicht. Es gibt zwei Wege der Übersetzung. Im ersten Weg, werden 16-Bit Funktionsaufrufe durch eine Übersetzungsschicht in 32-Bit Funktionsaufrufe umgewandelt und dann vom Betriebssystem bearbeitet. Der andere Weg führt genau in die andere Richtung. Die 32-Bit Funktionsaufrufe werden durch die Übersetzungsschicht übersetzt und wandelt diese in 16-Bit Funktionsaufrufe um, und werden dann vom Betriebssystem bearbeitet.

Die Benutzung der API ist nicht die einzige Möglichkeit, Anwendung für die Windowsbetriebssysteme zu programmieren, aber durch die Benutzung der API ist eine bessere Effizienz gewährleistet. Die Anwendungen können auch in Visual Basic oder Borland Delphi geschrieben werden, wo

die objektorientierten Grundlagen von Pascal dem Programmierer viel Arbeit abnehmen. Durch Verschachtelungen mehrerer Programmierschichten, die im Endeffekt auf die API zugreifen, verlangsamen das Programm und im späteren Verlauf der Leistungsoptimierung des Programmes wird der Programmierer früher oder später mit der Komplexität der API konfrontiert.

API gegenüber .NET Framework

Microsoft hat dieses Framework speziell für die Windows-Plattformen entwickelt. Es ist eine Virtuelle Maschine mit einer Laufzeitumgebung für Microsoft Windows Anwendungen. Dieses Framework ähnelt in vielen Teilen der Java Virtual Machine. Das .NET Framework besteht aus eine Laufzeitumgebung und der .NET Framework-Bibliothek. Aus Sicht des Anwenders hat sich nichts geändert, aber für die Programmierer vieles. Das .NET Framework ist auf C++ und C# basierend und im Gegensatz zur API, objektorientiert. Die Framework-Bibliothek besteht aus verschiedenen Klassenbibliotheken, wie zum Beipiel die Windows Forms, Windows Presentation Foundations (GUI), Webdienste,... . Ein großer Vorteil ist die Portierung der Programme, dafür muss aber das .NET Framework installiert sein. Das ist für dieses Projekt essentiell, denn es soll auf Schreibgeschütze Medien ausführbar sein (Win PE) und aus diesem Grund wird es von mir nicht verwendet.

3.2.2 Windows.h

Die Window.h Headerdatei ist die Masterdatei, die alle anderen Headerdateien inkludiert. In diesen Dateien sind die Funktionsaufrufe, Konstanten, Typdefinitionen und Datenstrukturen für das Windowsbetriebssystem definiert. Diese Headerdateien sind Teil der Dokumentation der API. Im Folgenden sind ein Paar Headerdateien aufgeführt die in der Windwos.h inkludiert werden:

- WINBASE.H Kernenfunktionen
- WINDEF.H Typdefinitionen
- WINNT.H Typdefinitionen mit Unicode Unterstützung
- WINUSER.H Funktionen für die Benutzerschnittstelle
- WINGDI.H Graphische Schnittstelle

3.2.3 WinMain

C/C++ Programme fangen mit eine *int main()* Funktion an. Da die API auf *C* basiert, fangen Anwendungen mit Windows als Zielsystem mit einer ähnliche Funktion an. Unter Windows ist die *main* Funktion unter „WINBASE.H“ als:

```
int WINAPI WinMain(
        HINSTANCE hInstance,
        HINSTANCE hPrevInstance,
        LPSTR lpCmdLine,
        int nShowCmd
        );
```

Der Bezeichner *WINAPI* beschreibt einen Datentyp. Dieser Datentyp ist definiert als *WINAPI __stdcall*. *__stdcall* ist eine Aufrufkonvention um Funktionen aus der Win32 API aufzurufen. Die *WinMain* Funktion bekommt als Parameter zwei *HINSTANCE* Variablen, ein *LPSTR* (Zeiger auf

eine Zeichenkette) und ein *int*.

HINSTANCE ist ein Handle auf eine Instanz. Das ist die Basisadresse von einem Modul im Systemspeicher(eine 32-Bitzahl, dass auf ein Objekt zeigt). Der erste Parameter ist eine Instanz auf die aktuelle Anwendung. Der zweiter Parameter wurde nur unter Win16 benutzt und ist in der Win32 API irrelevant, er wird automatisch auf *NULL* gesetzt. Der dritte Parameter enthält die Befehlszeilenargumente als Zeichenfolge. Aus dieser Zeichenfolge wertet das Programm aus, wie es ausgeführt werden soll. Der vierte und letzte Parameter ist ein Flag und gibt an, wie das Anwendungsfenster angezeigt werden soll (Minimiert, Maximiert oder Normalgröße).

3.2.4 Graphic User Interface (GUI)

Durch die Begrenzung der einzubauenden Hardware in den frühen Jahren des Computerzeitalters (Speicher und Prozessoren) waren alle Betriebssysteme Kommandozeileorientiert. Dies sollte sich aber ändern durch die Recherche von Xerox Palo Alto Research Center(PARC). Mitte der 70 Jahren wurde im Xerox PARC nach graphische Benutzeroberflächen recherchiert. Aus diesen Ergebnissen profitierten Macintosh und Windows, und bauten darauf ihre Betriebssysteme mit graphischen Benutzeroberflächen(Mac OS und Windows). Heutzutage kann man sich als Benutzer kaum eine Computerwelt ohne Benutzeroberflächen vorstellen. Die Windows API hat seit der Ankündigung(1983) und Veröffentlichung(1985) von Windows als Betriebssystem Funktionsaufrufe in die Headerdateien für die Programmierung von GUI's. Die Benutzung der API für die Programmierung von GUI's ist vielleicht ältmodisch", aber immerhin sehr genau und präzise. Im Gegensatz zu „GUI Builders“ wird kein unnötiger, und oft für Programmierer, unverständlicher Code geschrieben. Der Überblick und Verständnis der GUI Elemente wird durch die direkte Benutzung der API vereinfacht. Vor allem wird das Verständnis wie eigentlich eine GUI unter Windows und Windows als Betriebssystem funktioniert, durch die Verwendung der API deutlicher.

3.2.5 Aufbau einer GUI

Nachdem die *WinMain* Funktion die nötigen Parameter bekommt und diese vom Programm ausgewertet werden, muss die GUI als solche gebaut und registriert werden. Eine GUI besteht aus drei Teilen.

Initialisierung und Erzeugung der GUI

Damit der Benutzer ein Programmfenster sehen kann, muss dieses zuerst deklariert und initialisiert werden. Danach wird dieses Fenster dem System bekannt gegeben, in dem es registriert wird. Um ein Fenster zu deklarieren muss eine Variable der Struktur *WNDCLASS* erzeugt werden. Diese Struktur wird in *WINUSER.H* definiert und beinhaltet verschiedene Variablen, die die Eigenschaften des jeweiliges Fenster beschreiben.

```
typedef struct
{
                UINT            style ;
                WNDPROC         lpfnWndProc ;
                int             cbClsExtra ;
                int             cbWndExtra ;
                HINSTANCE       hInstance ;
                HICON           hIcon ;
                HCURSOR         hCursor ;
                HBRUSH          hbrBackground ;
                LPCTSTR         lpszMenuName ;
                LPCTSTR         lpszClassName ;
}WNDCLASS,    * PWNDCLASS ;
```

Die zwei wichtigsten Variablen dieser Struktur ist die Zweite und die Letzte Variable. Die zweite Variable *WNDPROC lpfnWndProc* beschreibt die Fensterprozedur. Genaueres zu dieser Variable wird demnächst erklärt. Die letzte Variable *LPCSTR lpszClassName*, beschreibt die Klasse des Fensters. Die Variable *HINSTANCE hInstance* muss dem Leser bekannt sein. Diese Variable wird auf den Wert gesetzt, welches das Programm über die *WinMain* bekommen hat. Alle andere Variablen der Struktur beschreiben wie das Fenster aussehen soll und sind für das Verständnis weiterhin irrelevant.
Nachdem die *WNDCLASS* Struktur deklariert und initialisiert wird, muss diese registriert werden. Durch den Aufruf der Funktion *RegisterClass*, die als Übergabeparameter einen Zeiger auf eine *WNDCLASS* Struktur hat, wird dem System bekannt gegeben, dass ein Fenster aufgebaut werden soll (mit dem gesetzten Eigenschaften der *WNDCLASS* Struktur). Wenn das Registrieren erfolgreich war, muss das Fenster noch erzeugt werden. Das System kennt das Fenster, aber es ist noch nicht sichtbar.

Damit ein Fenster sichtbar wird, muss der Programmierer die *CreateWindow*[2] Funktion aufrufen.

```
HWND WINAPI CreateWindow (
                                In_opt  LPCTSTR lpClassName,
                                In_opt  LPCTSTR lpWindowName,
                                In      DWORD dwStyle,
                                In      int x,
                                In      int y,
                                In      int nWidth,
                                In      int nHeight,
                                In_opt  HWND hWndParent,
                                In_opt  HMENU hMenu,
                                In_opt  HINSTANCE hInstance,
                                In_opt  LPVOID lpParam
);
```

Diese Funktion ist sehr wichtig bei der Erzeugung von graphischen Oberflächen, denn man kann alle Arten von Fenstern damit erzeugen. Damit ist gemeint, dass diese Funktion nicht nur „Hauptfenster“, sondern auch die einzelnen GUI Elemente erzeugt. Der erste Parameter beschreibt dieses Verhalten. Die *CreateWindow* Funktion erwartet eine null terminierte Zeichenkette. Diese Zeichenkette ist immer eine vordefinierte Systemklasse. Zum Beispiel, um ein Schaltfläche zu erstellen, muss hier als Parameter „button“ angegeben werden. Wenn der Programmierer ein von ihm erzeugtes Fenster darstellen will, muss er deswegen vorher im System eine Fensterklasse regis-

[2] http://msdn.microsoft.com/en-us/library/windows/desktop/ms632679(v=vs.85).aspx; 27.08.2013

trieren. Aus diesem Grund muss der Programmierer eine Fensterstruktur *WNDCLASS* registrieren. Um danach ein selbst gebautes Fenster zu erzeugen, muss der Programmieren den registrierten Fensterklassenname(*WNDCLASS lpszClassName*) angeben (anstatt „button“ oder andere vordefinierte Fensterklassen).

Der zweite Parameter beschreibt der dargestellter Name des Fenster. *DWORD dwStyle* ist eine Bitmaske, die die Darstellungsart des Fensters beschreibt. Die Darstellungsarten sind als konstante Werte (Window Styles[3])in der *WINUSER.H* Headerdatei definiert. Um ein „traditioneles“ Fenster zu erzeugen muss hier *WS_OVERLAPPEDWINDOW* angegeben werden. Die Nächste vier Parameter beschreiben die Positionierung im Bildschirm und die Breite und Höhe des Fensters.

HWND hWndParent beschreibt ein Handle auf ein anderes Fenster. Im Falle, dass dieses Fenster das Hauptfenster ist, wird hier NULL übergeben, sonst muss das Handle auf das „Parent Windows“ zeigen. Das *HMENU hMenu* ist für das Hauptfenster irrelevant. Es ist aber wichtig für Unterfenster(Popup Fenster oder GUI Elemente wie Knöpfe). *HMENU* ist eine Typdefinition für ein Handle auf ein Fenstermenü oder ein Unterfenster. Somit kann das Hauptfenster ein Kommando eines GUI Elements erkennen und auswerten, mehr dazu im Kapitel 3.2.5. Da auf den Parameter *HINSTANCE hInstance* im Vorlauf schon drauf eingegangen wurde wird er hier nicht mehr auf ihn drauf eingegangen. Der letzte Parameter beinhaltet Extrainformation, üblicherweise wird hier NULL angegeben.

Die *CreateWindow* Funktion liefert als Rückgabewert ein *HWND*. Dies steht für Handle auf ein Fenster. Somit kann der Programmierer und das Programm auf ein gewünschtes Fenster zugreifen. Um die erwähnten Parameter genauer zu verstehen, werden diese in den folgenden Beispiele erläutert.

```
HWND hwnd_Fenster = CreateWindow (

szAwendungsName,                    //Fensterklasse vom Programmierer registriert
"Das ist eine Hauptfenster",        //Text auf der Titelleiste
WS_OVERLAPPEDWINDOW,                //Stil des Fensters
CW_USEDEFAULT,                      //Position in der X Koordinate des Bildschirmes
CW_USEDEFAULT,                      //Position in der Y Koordinate des Bildschirmes
CW_USEDEFAULT,                      //Lange des Fensters
CW_USEDEFAULT,                      //Breite des Fensters
NULL,                               //Kein zugehöriges Hauptfenster
NULL,                               //Keine Identifikationsnummer
hInstance,                          //Programminstanz
NULL                                //keine extra Information
);
```

Um verschiedene GUI Elemente im Hauptfenster darstellen zu können, wie z.B. eine Schaltfläche, muss die Funktion mit den folgenden Parametern gesetzt werden:

[3] http://msdn.microsoft.com/en-us/library/windows/desktop/ms632600(v=vs.85).aspx; 27.08.2013

```
HWND Schaltfläche = CreateWindow (

"button",                          //Fensterklasse Schaltfläche
"Das ist eine Schaltfläche",       //Text auf den Schaltfläche
WS_CHILD | WS_VISIBLE,             //Child Fenster und anzeigen
100,                               //Position in der X Koordinate des Hauptfensters
100,                               //Position in der Y Koordinate des Hauptfensters
50,                                //Länge der Schaltfläche
35,                                //Breite der Schaltfläche
hwnd_Fenster,                      //Zugehöriges Hauptfenster
(HMENU) ID_Schaltfläche,           //Identifikationsnummer, wird vorher deklariert
NULL,                              //keine Instanz
NULL                               //keine extra Information
);
```

Nachdem ein Hauptfenster erzeugt worden ist, muss dieses noch explizit mittels der Funktionen *ShowWindow(hwnd_Fenster, iCmdShow)* und *UpdateWindow(hwnd_Fenster)* angezeigt werden.

Die Message Loop

Nachrichten zwischen verschiedenen Programmen werden unter Windows in sog. Nachrichtenschleifen, auch *message loop* genannt, verarbeitet. In diesen *message loops* werden Events von Schaltflächen oder Tastatureingaben ausgewertet und dem entsprechend reagiert. Die ausgelösten Events werden von Windows in eine Nachricht übersetzt, jede dieser Nachrichten ist eindeutig durch eine *MSG* Struktur beschrieben und eindeutig identifizierbar, diese wird anschließend in die *message loop* hineingeschrieben und wenn möglich verarbeitet. Wird eine neue ankommende Nachricht empfangen während eine andere Nachricht bearbeitet wird, wird die neue Nachricht in die sog. „Message Queue“(Warteschlange) geschrieben.

```
typedef   struct tagMSG
{
          HWND        hwnd ;
          UINT        message ;
          WPARAM      wParam ;
          LPARAM      lParam ;
          DWORD       time ;
          POINT       pt ;
}
MSG,      * PMSG;
```

Jede Nachricht hat als ersten Parameter ein Handle auf das zugehörige Fenster. Danach wird als zweiter Parameter ein *unsigned int (UINT)* übergeben, welches als ID der jeweiligen Nachricht zu verstehen ist. Diese Nachrichten sind in *WINUSER.H* deklariert und habe den Präfix *WM*, welches für „Window Message“ steht. Zum Beispiel wird ein Mausklick einem Fenster ausgelöst, sendet Windows eine Nachricht an das Fenster(durch *hwnd* angegeben) mit der Nachricht *WM_RBUTTONDOWN*.

Die Parameter *wParam* und *lParam* sind 32-Bit Nachrichtenparametern, abhängig von der jeweiligen Nachricht. Wird zum Beispiel auf eine Schaltfläche gedrückt, wird die *WM_COMMAND* Nachricht verschickt. Damit die Anwendung genau erkennen kann welche Aktion ausgeführt worden ist, wird in *wParam* noch eine zwei Angabe mitgeschickt. In den oberen 16-Bits wird die Notifikation *BN_CLICKED* verschickt. Dies deutet an, dass ein Schaltfläche geklickt worden ist.

In den unteren 16-Bits von *wParam* wird die Identifikationsnummer(nach den obigen Beispiel: ID_Schaltfläche) der Schaltfläche mitgeschickt. Mit diesen Informationen kann ein Programm genau auf die vom Benutzer ausgelösten Events reagieren. Beide 16-Bit Werte werden mit Hilfe der Makros *HIWORD* [4] und *LOWORD* [5] ausgewertet.

Der Parameter *time* gibt die Uhrzeit an, wann die Nachricht verschickt worden ist. Der Parameter *pt* ist eine *POINT* Struktur, wo die X und Y Koordinaten des Mausklicks gespeichert sind. Mit diesen Informationen erkennt das Fenster welche Schaltfläche betätigt wurde.

Eine Nachrichten Schleife ist standardmäßig so aufgebaut:

```
while   (GetMessage (&msg, NULL, 0, 0))
{
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
}
```

Die *GetMessage* Funktion speichert in der *msg* Struktur die aktuelle Nachricht. Die drei anderen Parameter spezifizieren, dass alle Nachrichten(von der jeweilige Anwendung) in die Schleife geschrieben werden. Die Funktion liefert immer ein Wert ungleich Null, außer wenn die Nachricht *WM_QUIT* lautet, denn damit wird die Schleife verlassen und das Programm beendet.
Die Funktion *TranslateMessage* gibt die Nachricht an das Betriebssystem weiter und übersetzt die virtuellen Tasten-Nachrichten auf tatsächliche Zeichen[6]. Die *DispachtMessage* Funktion gibt die Nachricht an Windows weiter, dabei wird die richtige Fensterprozedur (*WndProc* wird später erklärt)aufgerufen. Wenn die Fensterprozedur die Nachricht bearbeitet hat, wird die Nachricht zurück an Windows gesendet. Danach gibt Windows die bearbeitete Nachricht weiter an die jeweilige Anwendung und so kann die Schleife die nächste Nachricht laden und bearbeiten.

Fensterprozedur: WndProc Funktion

Im vorigen Unterkapitel 3.2.5 wurde die schon einmal Fensterprozedur erwähnt, aber noch nicht detailliert auf sie eingegangen, in diesem Kapitel wird nun näher auf deren Eigenschaften und dessen Funktionen eingegangen. Der Name der Funktion kann wie eine Variable gehandhabt werden. In dieser Variable sind die Nachrichten die das jeweilige Fenster bearbeitet. Das hat zur Folge, dass eine Fensterstruktur die als *WNDCLASS wc* deklariert und mit den folgenden Parametern *WNDPROC lpfnWndProc* initialisiert *wc.lpfnWndProc = WndProc;* ist, muss die Fensterprozedur für dieses Fenster *WndProc* heißen. Wie die Fensterprozedur heißt ist irrelevant, solange die Namen in der Fensterklasse und die Prozedur übereinstimmen, somit sind Fenster und Prozedur verbunden. Eine Fensterprozedur wird wie folgt definiert:

```
LRESULT CALLBACK    WndProc(
                    HWND hwnd,
                    UINT message,
                    WPARAM wParam,
                    LPARAM lParam);
```

Der Datentyp *CALLBACK* ist schon aus den vorigen Kapiteln bekannt und *LRESULT* ist eine Typdefinition vom Typ *long*. Die vier Parameter der Funktion sind die gleichen wie die ersten vier Parameter bei einer *MSG* Struktur. Falls ein Programm mehrere Fenster von der gleichen Fens-

[4] http://msdn.microsoft.com/en-us/library/windows/desktop/ms632657(v=vs.85).aspx; 27.08.2013

[5] http://msdn.microsoft.com/en-us/library/windows/desktop/ms632659(v=vs.85).aspx; 27.08.2013

[6] http://msdn.microsoft.com/en-us/library/windows/desktop/ms644955(v=vs.85).aspx; 27.08.2013

terklasse hat, spezifiziert der erste Parameter welches Fenster(Handle) die Nachricht schickt. Programme rufen die Fensterprozedur in der Regel nicht auf, sondern Windows. Will ein Programm die Prozedur direkt aufrufen, dann wird die Funktion *SendMessage* benutzt. Auf diese Methode wird im Folgenden drauf eingegangen.

Eine selbst geschrieben Fensterprozedur hat den Vorteil, dass ein Programm auf die Event reagiert, auf die man reagieren möchte. Es müssen nicht alle Event programmiert werden. Um vorzubeugen, dass Nachrichten unbearbeitet bleiben, wird am Ende der selbstgeschriebenen Fensterprozedur die*DefWindowProc* Funktion aufgerufen. Diese Methode bearbeitet alle mögliche Nachrichten und ist die default Fensterprozedur. Wenn diese Funktion nicht aufgerufen wird, werden grundlegende Funktionen in einer GUI nicht funktionieren.

Die erste Nachricht die eine Fensterprozedur bekommt ist die *WM_CREATE* Nachricht. Hier baut der Programmierer alle GUI Elemente auf, mit Hilfe der *CreateWindow* Funktion(Schaltfläche Beispiel). Eine zweite wichtige Nachricht ist die *WM_DESTROY* Nachricht. Diese Nachricht wird verschickt wenn der Benutzer ein Fenster schließen möchte. Der Programmierer ruft die *PostQuitMessage(0)* Funktion auf und automatisch wird eine *WM_QUIT* Nachricht verschickt. Somit wird das Fenster geschlossen, die Message Loop verlassen und das Programm bzw. Fenster richtig beendet.

Es gibt noch weitere Nachrichten, die man „manuell“ verschicken kann. Mit Hilfe der erwähnten Funktion *SendMessage* [7]. Die Funktion hat die gleichen Parameter wie eine Fensterprozedur. Durch das Handle wird angegeben, an welches Fenster die Nachricht verschickt werden soll. Durch den zweiten Parameter wird angegeben, welche Nachricht verschickt wird, und die letzten zwei sind Extrainformationen der Nachricht(wenn nötig). Möchte man im Ablauf des Programms zum Beispiel der Darstellungstext von unserer Schaltfläche ändern, wird die *SendMessage* Funktion so aufgerufen:

```
SendMessage(hwnd_Schaltfläche, WM_SETTEXT, 0, (lParam)"NeuerText");
```

3.2.6 Die RS-232 Schnittstelle und die Windows API

Über die Windows API hat man direkt Zugriff auf die RS-232 Schnittstelle. Zum Verwalten der Schnittstelle und zum Setzen der Eigenschaften gibt es verschiedene Datenstrukturen, die man aufrufen und ändern muss je nach Bedarf.

Öffnen und Schließen eines Ports

Um Zugriff auf die Datenstrukturen zu bekommen, muss zuerst eine RS-232 Schnittstelle (COM Port) geöffnet. Durch die Funktion *CreateFile*[8] bekommt man ein Handle auf den angegeben Port. Ein Handle ist eine Referenzwert zu einer vom Betriebssystem verwalteten Systemressource, in diesem Fall eine im System vorhandene RS-232 Schnittstelle. Mit *CreateFile* kann man auch Zugriff auf Dateien, Datenstreams und andere Kommunikationsressourcen bekommen. Das Handle muss gespeichert werden, denn damit wird der jeweiliger Port identifiziert und angesprochen für weitere Operationen. Durch die *CreateFile* Funktion wird die Datei oder in diesem Fall die Input /

[7] http://msdn.microsoft.com/en-us/library/windows/desktop/ms644950(v=vs.85).aspx; 28.08.2013

[8] http://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx; 25.08.2013

Output Schnittstelle für diese Anwendung reserviert. Das heißt, für das Betriebssystem und andere Anwendungen steht diese Schnittstelle nicht mehr zur Verfügung.

Um den richtigen Zugriff auf einen Port zu haben, müssen auch die richtigen Flags bei dem Aufruf der *CreateFile* Funktion angegeben werden. Als Flags sind die folgende Parameter anzugeben:

- Schreib und Leserechte
- Non-Sharing Modus
- Öffnen vor nur existierende Schnittstellen
- Asynchron Modus

Um ein Programm „sauber“ zu beenden, müssen offene Handles geschlossen werden. Durch die Funktion *CloseHandle*[9] mit Angabe eines gültiges Handles wird dieses geschlossen und steht für das Betriebssystem und andere Anwendungen wieder zur Verfügung.

Konfiguration eines Ports

Es gibt drei wichtige Strukturen, die für die Konfiguration einer COM Schnittstelle relevant sind. Mittels dieser Strukturen werden die Eigenschaften, einstellbare Parameter und Wartezeiten einer Schnittstelle eingestellt.

Die *DCB*[10] Struktur definiert die Ansteuerungseigenschaften für die Schnittstelle. Um die Struktur für eine angegebene Schnittstelle zu laden, wird die Funktion *GetCommState* aufgerufen. Diese Struktur besteht aus 28 verschiedene Variablen. Diese Variablen beschreiben wie die Schnittstelle konfiguriert ist. Man kann die Baudrate, Parität, Stopbits, Datenbits, Flusssteuerung und Fehlerbenachrichtigung einstellen.

Um die Schnittstelle richtig einzustellen und keine falsche Eingaben in die *DCB* Struktur zu schreiben, ist es sinnvoll vorher die *COMMPROP* [11] Struktur auszuwerten. Durch die Funktion *GetCommProperties* wird die Struktur für die angegebene Schnittstelle geladen. Diese Struktur besteht aus 18 Variablen und beschreibt die möglichen Einstellungen für diese Schnittstelle. Die Einstellungen werden aus dem Treiber der jeweiliger Schnittstelle(Onboard Ports, Sunix, Intel SOL oder ITE) gelesen. Die für diese Arbeit relevanten Parameter ist die maximale einstellbare Baudrate.

Während der Übertragung von Daten über eine serielle Schnittstelle sind maximale Wartezeiten fällig. Dieses Werte entstehen durch die Übertragungslänge und der eingestellten Baudrate. Auch die Wartezeit zwischen zwei ankommenden Bytes ist wichtig um das Ende der Übertragung bestimmen zu können. Diese Werte sollten dynamisch berechnet werden und danach in die Schnittstelle eingestellt werden. Dafür ist die *COMMTIMEOUTS* [12] Struktur zuständig. Die Struktur besteht aus fünf Variablen, die die Wartezeiten in Millisekunden angeben. Die erste Variable beschreibt die Zeitüberschreitung zwischen zwei ankommende Bytes, wird dieser Wert überschritten,

[9] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724211(v=vs.85).aspx; 25.08.2013

[10] http://msdn.microsoft.com/en-us/library/windows/desktop/aa363214(v=vs.85).aspx; 28.08.2013

[11] http://msdn.microsoft.com/en-us/library/windows/desktop/aa363189(v=vs.85).aspx; 28.08.2013

[12] http://msdn.microsoft.com/en-us/library/windows/desktop/aa363190(v=vs.85).aspx; 28.08.2013

so wird die Leseoperation beendet. Der zweite und vierte Parameter(jeweils für Lesen und Schreiben) beschreiben die Zeit, die für die Übertragung aller Bytes nötig ist(für die aktuelle Schreibe- oder Leseoperation). Diese Parametern sind von der Baudrate und der Bitlänge eines Zeichens abhängig. Damit ist gemeint, wie unter 3.1.2 erläutert, ein Zeichen kann aus 8 Bits plus Startbit, Stopbit und Parität bestehen. Also werden nicht genau 8 Bits(1 Byte) per Charakter, sondern oft mehr (11 Bits zum Beispiel) übertragen. Die Wartezeit wird durch die folgende Formel berechnet:

$$Wartezeit = \frac{AnzahlderBits}{Baudrate} \times 1.1$$

In der Formel wird mal 1.1 multipliziert um eine 10% Kulanz zu haben. Der dritte und fünfte Parameter sind eine extra Wartezeit, für jeweils Lesen und Schreiben, die zu der gesamte berechnete Wartezeit pro Lese-/Schreibvorgang addiert wird. Falls der berechnete Wert unter zehn Millisekunden ist, wird dieser auf zehn Millisekunden gesetzt.

Wenn ein Port geöffnet wird, muss die Art der Kommunikation angeben werden. Die Kommunikation zwischen den Schnittstellen kann synchron(nonoverlapped) oder asynchron(overlapped) erfolgen. Für diese Arbeit habe ich mich für asynchrone Kommunikation entschieden. Dieser Art der Kommunikation ist etwas komplexer, bietet aber mehr Flexibilität und Effizienz.

Lesen und Schreiben

Nachdem ein Port konfiguriert ist, wie es nötig ist, kann man diesen Port benutzen um Informationen zu schicken oder zu empfangen. Um in einen Port schreiben oder aus einem Port lesen zu können werden zwei verschiedene Funktionen benötigt. Um in einen Port zu schreiben wird die Funktion *WriteFile*[13] benötigt. Diese Funktion benötigt als ersten Parameter das Handle auf den geöffnete Port. Als zweiten muss angegeben werden, welche Information geschrieben werden soll und die Größe dieser Information in Bytes. Als optionale Parameter kann man einen Zeiger wo die Menge der geschriebene Bytes geschrieben wird und ein Zeiger zu einer *OVERLAPPED* Struktur übergeben. Die Kommunikation zwischen Ports in dieser Arbeit ist Asynchron, deswegen wird hier eine Zeiger auf eine Struktur übergeben.

Das Lesen aus einem Port geschieht sehr ähnlich. Die Funktion dafür heißt *ReadFile*[14]. Die Lesefunktion benötigt genau die gleichen Parameter wie die Schreibfunktion. Da die Lese und Schreibvorgänge sehr ähnlich sind, ist der Aufbau beider Funktionsaufrufe auch ähnlich. Um aus einem Port zu lesen oder schreiben muss zuerst eine *OVERLAPPED* Struktur deklariert werden. Somit versichern wir uns, dass die Kommunikation asynchron ist. Danach muss in der Struktur der Parameter *hEvent* initialisiert werden. Dieser Parameter ist ein Handle auf ein Event. Ein Event in diesem Fall ist der Schreibe- oder Lesevorgang. Durch dieses Event kann der Status von dem Vorgang abgefragt werden. Als erstes wird die jeweilige Lese- oder Schreibfunktion aufgerufen. Wenn dieser Vorgang erfolgreich war, muss der Status abgefragt werden, um sicher zu sein, dass der Vorgang vollständig ist. Im Fall, dass sich das Schreiben oder Lesen noch verzögert, muss auf die Beendigung des Events gewartet werden. Zuerst muss abgefragt werden, ob ein Fehler entstanden ist. Wenn der Fehler *ERROR_IO_PENDING* heißt muss das Programm noch warten, sonst ist ein anderer Fehler entstanden. Um auf das Beenden eines Events zu warten und zu reagieren, wird die Funktion *WaitForSingleObject*[15] aufgerufen. Diese Funktion wartet bis das angegebene

[13] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365747(v=vs.85).aspx; 28.08..2013

[14] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365467(v=vs.85).aspx; 28.08.2013

[15] http://msdn.microsoft.com/en-us/library/windows/desktop/ms687032(v=vs.85).aspx; 28.08.2013

Event(lesen oder schreiben) sein Status ändert oder nach einer bestimmten Zeit abläuft. Diese Zeit ist der Timeout, welcher vorher berechnet wurde. Wenn die Zeit abgelaufen ist, wir ein Timeout Fehler ausgegeben. Das heißt, das Programm muss noch warten. Ändert sich der Status des Events, bevor die Zeit abläuft wird mittels der Funktion *GetOverlappedResult*[16] das Event noch abgefragt. Diese Funktion muss das Handle des Ports, sowie die *OVERLAPPED* Struktur und die Anzahl an Bytes die transferiert werden sollten, angeben. Wenn diese Funktion keine Fehler meldet, dann wurde der jeweilige Vorgang erfolgreich abgeschlossen. Im Fall eines Schreibvorgang, muss sichergestellt werden, dass alle Bytes geschrieben worden sind. Wird ein Lesevorgang durchgeführt, muss sicher gestellt werden, dass die komplette Information angekommen ist, bevor gelesen wird. Es kann daher zu Timeout Fehler kommen. Um solche Fehler nicht als Abbruchbedingung zu melden, wird nach einer Vereinbarung mit dem Auftragsgeber, fünf mal versucht die Zeichen zu lesen oder zu schreiben. Wenn danach der Fehlern noch vorliegt, wird dieser gemeldet.

[16] http://msdn.microsoft.com/en-us/library/windows/desktop/ms683209(v=vs.85).aspx; 28.08.2013

Kapitel 4

Lösungsansätze

4.1 Fehlererkennung

Das Ziel des COM-Port-Test-Tools ist Fehler in der Systemhardware beziehungsweise in der Kommunikation zwischen den Schnittstellen zu erkennen. Damit dieses Ziel erreicht wird, muss die Ermittlung eines Fehlers äußerst genau geschehen. So muss jeder Funktionsaufruf und Kommunikationsvorgang immer abgefragt werden. Bei jeder Abfrage müssen die Fehler ausgewertet und gemeldet werden. Nur so erfüllt das Testprogramm seinen Zweck. Die Herausforderung wird darin bestehen, die Fehler genau zu definieren und sie so zu melden, dass der Tester die Ursache sofort erkennt.

4.2 Master Slave Modus zwischen zwei Systeme

Die Synchronisierung zwischen zwei verschiedenen BEETLE Systeme wird eine große Herausforderung darstellen. Dafür wird ein kleines Protokoll oder Handshake definiert. In diesem Vorgang muss der Master wissen, dass der Slave fertig eingestellt ist und bereit für ein Datenverkehr ist. Dabei müssen beide Schnittstellen die richtige Einstellungsparameter (Baudrate, Paritätbits, Stopbits, Datenbits, etc.) besitzen und konfigurieren.

Um das Protokoll oder Handshake zu definieren gibt es verschiedene Möglichkeiten. Darunter müssen als erstes die Timeouts beachtete werden. Der Master darf keine Information oder Übertragungstexte senden, bevor der Slave nicht in der Lage ist, Daten zu empfangen. Wäre der Slave nicht bereit sein, wurde der Master ein Lese-Timeout melden, weil er Daten verschickt hat, aber der Empfänger keine Daten zurück versendet . Im Gegensatz, darf der Slave nicht Daten erwarten, so lange der Master nicht in der Lage ist, Daten zu verschicken.

Als Grundidee um dieses Problem zu lösen, werde ich Synchronisierungszeiten und Vorgänge einbauen, in dem der Master für eine definierte Zeit auf den Slave wartet, und gegenseitig auch. Wenn diese Zeit abläuft, wird den Benutzer gemeldet, dass es Probleme mit der Synchronisierung beider Systeme gab, und das kein Test stattfinden wird. Ein großer Nachteil dieser Lösung ist, dass durch das Abfragen auf Bereitschaft ein Pollingverfahren[1] entsteht.

[1] Das zyklische Abfragen eines Status

Im besten Fall, wo keine Fehler bei der Kommunikation auftreten, wird es nicht so nötig sein, sich ständig zu Synchronisieren. Im Fehlerfall, muss nach jedem Test eine Synchronisierung stattfinden. Empfängt der Slave verfälschte Informationen und meldet ein Fehler, wurde sich sofort auf den nächsten Test vorbereiten. Das wurde den gesamten Testvorgang verfälschen. Es werden weitere ähnliche solche Fälle auftreten auf denen sinngemäß reagiert werden muss.

Um das Pollingverfahren zu vermeiden, kann die Synchronisierung auf Basis von Events [2] geschehen. Die Schnittstelle erzeugt Interrupts bei der Ankunft von einem Byte. So kann der Slave reagieren, erst wenn der Master was tatsächlich verschickt hat. Um die Interrupts abzufragen, muss ich Kommunikation Events von Windows abfragen. Wenn das System ein Interrupt empfängt, wird dieses im Programm als ein Event weitergeleitet.

Im ersten Blick, wird das 100% Umgehen von einem Pollingverfahren schwer sein. Diese wird sich durch Testen während der Entwicklung des „WN Serial COM Port Test“ beweisen. Je nach Testergebnisse werde ich die entsprechende Lösung anwenden.

4.3 C++ und die GUI

Die graphischen Oberflächen mit Anwendung von der Windows API besteht aus „C“ Code, und muss eine statische Fensterprozedur besitzen. Das heißt, es können beliebig viele Benutzeroberflächen kreiert werden, aber es gibt nur eine Prozedur die die Nachrichten bearbeitet. Durch die Objektorientierung unter „C++“ ersteht ein Konflikt wegen der statische Fensterprozedur. Um eine statische Methode zu definieren, müssen auch statische Variablen erzeugt werden. So werden diese Variablen nur einmal erzegut, unabhängig davon, wie viele Fenster erstellt werden. Wäre die Methode und die Variablen nicht statisch und mehrere Fenster werden erstellt, wurden die versendetete Nachrichten durcheinander kommen.

Um dieses Problem zu umgehen wird mit Hilfe von einem *„Template Class“* die Oberfläche definiert. Eine Klasse wird definiert, in der die allgemeine Eigenschaften eines Fensters gesetzt werden und die Nachrichten von jedem Fenster richtig umgeleitet werden. Dieses geschieht indem die benötigte statische Fensterprozedur in dem Template definiert wird. Die aller ersten Nachricht die ein Programm bekommt um eine Oberfläche zu erzeugen ist die „WM_NCREATE“. Die Fensterprozedur fragt nach dieser Nachricht ab und wenn die Nachricht ankommt, wird durch Zeiger die Fensterhandles richtig gesetzt und weitergeleitet an die richtige Prozedur. Da diese Fensterprozedur statisch ist, egal wie viele Oberflächen kreiert werden, wird es nur einmal diese Fensterprozedur geben.

Die Fensterklasse erbt somit von das *Template Class* und kann eine eigene Fensterprozedur definieren. Der definierten Fensterprozedur in der Fensterklasse werden durch das Template die Nachrichten weitergeleitet. Dadurch werde ich auch in der Lage sein, meine eigene Fenster so zu gestalten wie die Voraussetzungen es definieren. Und falls es nötig ist, mehrere Fenster erzeugen.

[2] http://msdn.microsoft.com/en-us/library/aa450667.aspx; 30.08.2013

Kapitel 5

Systementwurf

5.1 Klasse Window

Die Klasse *Window* baut die Benutzeroberfläche auf und speichert die Eingabe des Benutzers in Variablen. Diese Werte werden dann an der Klasse *Interpreter* weiter gegeben. In der Benutzeroberfläche werden die Eigenschaften für ein Testvorgang eingestellt.

5.1.1 Design

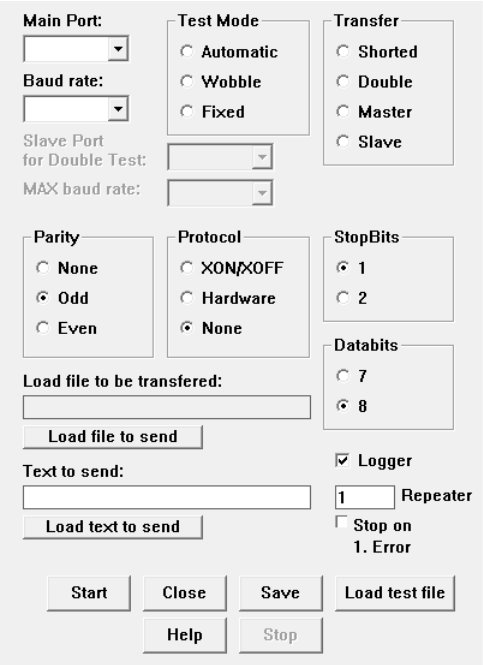


Abbildung 5.1: Design der Benutzeroberfläche

Die Benutzeroberfläche ist in drei Abteilungen geteilt. Die verschidene Abteilungen sind in Abbildung 5.1 zu erkennen. Genaueres zu alle Parametern wird in Kapitel 7 und 3 erläutert. Die erste Reihe von Einstellungen hantieren die Testeinstellungen. Diese Einstellung sind:

- Main Port: welches COM Port soll getestet werden.

- Baud rate: beschreibt mit welcher Baudrate der ausgewählte Port getestet werden soll.
- Test Mode: definiert welcher Testvorgang der Benutzer haben will.
- Transfer: legt fest, wie die Kommunikation dieses Ports stattfinden wird.

Die zwei ausgegraute Einstellungen („Slave Port for Double Test“ und „MAX baud rate“) sind abhängig vor der Einstellungen in „Test Mode“ und „Transfer“.

Die zweite Abteilung von Einstellungen sind die Übertragungsparameter und drei weitere Testeinstellungen. Diese Parameter sind:

- Parity: Paritätsüberprüfung.
- Protocol: Übertragungsprotokoll.
- Stopbits: Anzahl der Stopbits.
- Databits: Anzahl der Datenbits.
- Logger: definiert ob eine Log-Datei erstellt werden soll.
- Repeater: Anzahl der Testwiederholungen.
- Stop on 1. Error: bestimmt ob bei der ersten Fehlererkennung angehalten werden soll.
- Load file to be transfered: Übertragungsdatei.
- Text to send: Übertragungstext.

Die letzte Reihe von Elemente in der Benutzeroberfläche sind die Knöpfe. Diese führen die jeweilige Vorgange aus.

- Start: startet ein Test.
- Close: schließt das Test Tool.
- Save: speichert in eine Testdatei die Einstellungen.
- Load test file: ladet eine Datei die im Test übertragen wird.
- Help: zeigt ein Fenster mit Erklärungen zum Test Tool.
- Stop: stoppt der laufende Test.

Bei drücken des Start-, Save- oder „Load test file“knopfes werden die Testeinstellungen an die Klasse *Interpreter* weitergegeben und dort werden diese verarbeitet.

5.1.2 Aufbau

Die Klasse *Window* erbt von der *Template class* in die Headerdatei „BaseWindow“. Die Headerdatei beinhaltet die Definition des Templates. Im das Template wird die, für die Nachrichtenschleife nötige, statische Fensterprozedur deklariert. Hier wird nur nach der Nachricht *WN_NCCREATE* abgefragt, wie bereits im Kapitel 4.3 erwähnt. Wenn die Nachricht empfangen worden ist, werden die weitere Nachrichten an das Handle des Fensterobjekt weitergeleitet. In das Template wird außerdem auch eine Fensterklasse deklariert, registriert und danach erzeugt. Somit werden alle Nachrichten eines Fensterobjektes immer zuerst an die statische Fensterprozedur gesendet, diese leitet durch Zeiger und Handles die Nachricht an das korrekte Fenster. Um genauer die Funktionalität des Templates zu verstehen, bitte siehe Anhang A.1.

5.1.3 Aufgaben

Die Klasse hinter der Benutzeroberfläche besteht aus zwei Teilen. Das erste Teil bearbeitet die Eingabe des Benutzer und der Aufbau der Oberfläche, das zweite Teil die Weitererarbeitung der Daten.

Das erste Teil besteht aus die Methode *HandleMessage*, diese Methode ist die Fensterprozedur. Als erstes werden die Elemente, mit der Ankunft der Nachricht *WM_CREATE*, des Fensters aufgebaut. Danach wird auf die Nachricht *WM_COMMAND* gewartet und jeweils reagiert. Wird zum Beispiel das Start Knopf gedrückt, bekommt die *HandleMessage* Methode die *WM_COMMAND* Nachricht mit dem Parameter *ID_BT_START*. Das ist die Identifikationsnummer im Programm für die Startschaltfläche.

```
1     case WM_CREATE:
2         {
3             //Create all GUI Elements
4
5             //example: Start button
6             _hwnd_Start = CreateWindowA("button", "Start",
7         WS_CHILD | WS_VISIBLE,
8         POS_X + 20, POS_Y2 + 290, 70, 30, m_hwnd, (HMENU)ID_BT_START,
9         NULL, NULL);
10        }
11        break;
```

Wenn die Startschaltfläche gedrückt worden ist, fängt das zweite Teil der Klasse an. Das zweite Teil besteht aus die Weiterverarbeitung der Eingaben. Im Fall von der Startschaltfläche, wird als erstes einen neuen Thread aufgerufen. So kann die Benutzeroberfläche immer noch Nachrichten verarbeiten, während im Hintergrund, das Programm die Datenverarbeitung beginnt. Der gleiche Vorgang geschieht, wenn die *Load test file* Schaltfläche gedrückt wird.

Im Fall von der Startschaltfläche, ruft das Thread die Methode *sendTestSettings* auf, die die Eingaben weiter an ein Interpreter-Objekt gibt. Dies geschieht in dem ein Objekt der Klasse *Interpreter* erzeugt wird und dessen Eigenschaften gesetzt werden.

```
1   case WM_COMMAND:
2   {
3     //React to GUI Elements actions
4     //example: Start button
5     case ID_BT_START:
```

```
6
7       //hide the elements while testing
8       viewAllElements(FALSE);
9
10      //start a new thread
11      _t1 = thread(&Window::sendTestSettings, this);
12
13      //detach the thread so it can test and main thread waits
14      //for it to finish or waits for the user to press stop
15      _t1.detach();
16  }
17    break;
```

5.2 Klasse Interpreter

Die Klasse *Interpreter* trennt die Benutzeroberfläche von die Fachlogik des Programms. Die Benutzeroberfläche kann den Interpreter aufrufen und hat Zugriff auf die *public* Methoden der Klasse. Der Interpreter kann nicht auf Elemente oder Methoden der Klasse *Window* zugreifen. Durch diese Trennung kann der Benutzer nie "ùnkontrolliert“ auf die Fachlogik zugreifen, nur über die Möglichkeiten die der Programmierer anbietet.

5.2.1 Aufgaben

Eine Aufgabe des Interpreters ist die Überprüfung der Eingaben des Benutzers bevor die Tests beginnen. Durch „setter“ Methoden werden die Eingabeparameter der GUI in lokalen Variablen gespeichert.

```
1 void Interpreter::setTransfer(int iTransfer)
2 {
3   this->_iTransfer = iTransfer;
4 }
```

Danach werden die lokalen Variablen auf Fehleingaben geprüft. Sind Fehler erkannt worden, werden diese dem Benutzer bekannt gemacht und alle Parameter in der Klasse werden auf ein Standardwert zurückgesetzt. Sind keine Fehleingaben erkannt, werden die Werte der Variablen in einer Datenstruktur gespeichert. Mehr zu dieser Struktur erfahren sie im Kapitel 5.3. Folgende Codezeilen stellen dar, wie ein Fehlerüberprüfvorgang aussieht. Hier wird nach dem Übertragungsmodus abgefragt.

```
1 if (_iTransfer == DEFAULT_VALUE)
2 {
3   MessageBoxA(NULL,"Bitte␣wählen␣Sie␣ein␣Transfermodus␣aus",
4               WINDOW_TITLE, MB_OK | MB_ICONERROR);
5   setDefaultValues();
6 }
7 else
8 {
9   //speichern des Transfermodus
10  _testManager->testStruct.iTransfer = _iTransfer;
11 }
```

Die Werte in den Variablen müssen überprüft werden, um Fehler zu vermeiden. Da die Werte nicht nur aus Einstellungen in der Benutzeroberfläche stammen, sondern auch aus eine Testkonfigurationsdatei, können falsche Eingabe vorhanden sein.

Außer die Überprüfung der Eingabeparameter und setzen der lokalen Variablen, gibt diese Klasse die Testeinstellungen an die anderen Klassen weiter. Hier trennt sich das Programm in zwei Pfade. Das erste Pfad gibt die Datenstruktur an die Klasse *TestManager* weiter. Der zweite Pfad speichert oder liest eine Testkonfigurationsdatei. Im Fall, dass eine Testkonfigurationsdatei gelesen werden soll, wird als erstes die Klasse *IniFileHandler* aufgerufen und danach die Klasse *TestManager*, damit ein Test mit den gelesenen Einstellungen gestartet wird. Sollen die Eingabeparameter der Benutzeroberfläche in einer Testkonfigurationsdatei gespeichert werden, werden die überprüfte Eingabeparameter in einer vom Benutzer angegebene Datei gespeichert. Dafür wird auch die

Klasse *IniFileHandler* aufgerufen.

Da die Klasse *Interpreter* die Schnittstelle für die Kommunikation zwischen Fachlogik und Benutzer ist, wird auch durch diese Klasse die Meldung gegeben, ein Test anzuhalten. Durch das Klicken der Schaltfläche „Stop“ in der Benutzeroberfläche wird eine boolesche Variable weiter an die Testlogik gegeben. Diese Variable wird vor jedem Testschleife abgefragt, ist sie gesetzt, dann wird das Testen angehalten.

Die Klasse hat ein Objekt der Klasse *Com*, um in der Benutzeroberfläche die im System zur Verfügung stehende COM Ports aufzulisten. Durch die Trennung der GUI mit der Fachlogik, darf die *Window* Klasse kein Objekt der Klasse *Com* haben.

5.3 Struktur TestStruct

Ein *struct* oder Struktur dient dazu, mehrere logische zusammenhängende Variablen verschiedener Datentypen zusammenzufassen. In *C* beinhalten Strukturen nur Variablen, in *C++* wurden die Strukturen erweitert und dürfen auch Funktionen beinhalten. Dadurch ist der Unterschied zu einer Klasse nur die Zugriffsrechte auf die Eigenschaften. In einer Struktur sind die Zugriffsrechte auf Elemente mit *public* definiert, in einer Klasse mit *private*[1]. Da ich nur Variablen benötigte, und keine Methoden, habe ich mich für eine Struktur entschieden.

5.3.1 Aufbau

Die Datenstruktur fasst alle Eigenschaften für einem Test zusammen. Die *TestStruct* beinhaltelt folgende Variablen und wird so definiert:

```
1  struct TestStruct
2  {
3    string sMasterPort;
4    string sSlavePort;
5    string sTextToTransfer;
6    string sFilePath;
7    int iTransfer;
8    int iBaud;
9    int iBaudrateMax;
10   int iTestMode;
11   int iParity;
12   int iProtocol;
13   int iStopbits;
14   int iDatabits;
15   int iTransTextMode;
16   int iRepeater;
17   bool bLoggerState;
18   bool bStopOnError;
19   vector<string> svBaudrates;
20 }
```

Jeder dieser Variablen stellt ein Wert dar, für eine Einstellung in der Benutzeroberfläche. Für eine genauere Erläuterung der Variablen und dessen Werte, bitte siehe Kapitel 7 und 5.8.

[1] [Lou10]

5.4 Klasse Com

Die Klasse *Com* umfasst alles um die Verwaltung eines COM Ports. Um aus einem COM Port lesen und schreiben zu können, müssen zuerst die Eigenschaften des jeweiligen Ports gesetzt werden. Alle Ports in einem System werden beim Systemstart mit einer Standardkonfiguration konfiguriert. Will der Benutzer Peripherieelemente an einem COM Port anschließen, die nicht die gleiche Konfiguration besitzt wie die im System vordefiniert, muss die Konfiguration des Ports angepasst werden. Dafür muss ein COM Port zuerst geöffnet werden, danach kann die Konfiguration geändert werden. Wenn diese beide Schritte erfolgreich abgelaufen sind, können aus dem COM Port Informationen gesendet und empfangen werden. Als letzter Schritt ist sehr wichtig der öffene COM Port wieder zu schließen. Somit steht der COM Port wieder andere Applikationen und das System zur Verfügung.

5.4.1 Aufgaben

Um einem COM Port zu öffnen, wird die gleiche Systemfunktion aufgerufen, wie um eine Datei zu erzeugen. Die Funktion *CreateFile* gibt ein Handle auf das gewünschte COM Port. Durch dieses Handle wird der jeweilige Port für andere Operationen identifiziert.

```
1 hCom = CreateFile(portNumber.c_str(),
2             GENERIC_READ | GENERIC_WRITE,
3             0,
4             NULL,
5             OPEN_EXISTING,
6             FILE_FLAG_OVERLAPPED,
7             NULL);
```

Das erste Parameter der Funktion ist der Name des gewünschtes Ports. Besitzt der COM Port einen Wert von eins bis neun wird ein Wert wie „COM5“ gesetzt. Ist der Wert des Ports größer neun und maximal 256, muss der erste Parameter folgendermaßen gesetzt werden: „\\.\COM255“. Der zweite Parameter beschreibt die Zugriffsrechte auf der COM Port. Da wir Informationen senden und empfangen möchten, braucht das Programm Lese- und Schreibrechte. Sehr wichtig ist der fünfte Parameter. Dieser beschreibt das nur existierende COM Ports geöffnet werden sollen. Da die Funktion auch Dateien erzeugen kann, sind diese nicht vorhanden, werden sie kreiert. Im Fall von COM Ports, keine COM Ports sollen erzeugt werden, sondern die als Hardware im System vorhanden öffnen. *FILE_FLAG_OVERLAPPED* setzt die Eingenschaft, dass der COM Port im asynchron Modus geöffnet werden soll.

Damit den Benutzer, in der Benutzeroberfläche, nicht vorhandene COM Ports angeboten werden, wird bei der Aufbau der GUI alle zur Verfügung stehende COM Ports aufgelistet. Dafür wird in einer Schleife versucht alle COM Ports zwischen Null bis 256 zu öffnen, alle die Ports die einen gültigen Handle zurück geben werden aufgelistet.

Wenn ein COM Port nicht erfolgreich geöffnet werden konnte, gibt die *CreateFile* Funktion ein nicht valides Handle zurück. Wird ein Port erfolgreich geöffnet, können die Eigenschaften des Ports bearbeitet werden. Wie in Unterkapitel 3.2.6 erklärt, werden die Eigenschaften des Ports durch verschiedene Strukturen gesetzt.

Für jeden Test kann der Benutzer die Baudrate einstellen. In der Benutzeroberfläche wird in Form einer Liste, in einer „Combo Box“, die unterstützten Baudrate aufgelistet. Um die Baudraten zu

ermitteln muss zuerst die Struktur *COMMPROP* geladen werden. In der Variable *dwSettableBaud* sind als Bitmaske die unterstützten Baudraten gespeichert. Das Algorithmus dafür sieht so aus:

```
1  //Lade die COMMPROP Stuktur des geöffneten Ports
2  GetCommProperties(hCom, &commProp);
3
4  //Bitmaske mit den unterstützten Baudraten
5  bitset<32> bitMask ((int)commProp.dwSettableBaud);
6
7  //ignoriere die letzte 4 bits
8  for( int i = 0; i < 28; i++)
9  {
10   //falls das Bit gesetzt ist
11   if (bitMask.test(i) == true)
12   {
13     //dann wird die Baudrate unterstützt
14     vBaud.push_back(saDefaultBaudrates[i]);
15   }
16 }
```

In der Schleife wird geprüft ob der jeweilige Bit an der Stelle „i“ eine Eins oder Null ist. Falls es eine Eins ist, dann wird die Baudrate im Array an der Stelle „i“ in eine Vektor gespeichert. In diesem Vektor werden alle Baudraten gespeichert, die vom angegebene Port unterstützt werden. Der Vektor gehört zur *Com* Klasse und wird durch das ganze Programm benutzt.

Die Timeoutwerte für Lese- und Schreibzugriffe werden auch in dieser Klasse berechnet und gesetzt. Dafür muss eine Struktur *COMMTIMEOUTS* deklariert werden. Die Struktur wird nach .....Wünsch und Zweck..... des Programms bearbeitet. Für das Test Tool werden in der Struktur die Zeit für die Übertragung eines Zeichens geschrieben. Für die Berechnung für dieses Wert und die Erklärung siehe bitte Unterkapitel 3.2.6.

```
1  //Port Timeout Struktur
2  COMMTIMEOUTS timeouts;
3
4  timeouts.ReadIntervalTimeout         = 20;
5  timeouts.ReadTotalTimeoutMultiplier  = iTimeOut;
6  timeouts.ReadTotalTimeoutConstant    = 100;
7  timeouts.WriteTotalTimeoutMultiplier = iTimeOut;
8  timeouts.WriteTotalTimeoutConstant   = 100;
9
10 SetCommTimeouts(hCom, &timeouts);
```

Durch die Funktion *SetCommTimeouts* mit Angaben des geöffneten Ports und eine Referenz auf einer Variable der Struktur *COMMTIMEOUTS* werden die Timeouts gesetzt. Davor müssen die Elemente von der Struktur initialisiert werden. Die Werte *ReadTotalTimeoutMultiplier* und *WriteTotalTimeoutMultiplier* werden mit der berechnete Zeit pro Zeichen gesetzt. Für die andere drei Variablen worden die empfohlene Werte aus der MSDN Online Bibliothek[Mic95] übernommen.

Objekte der Klasse *Com* können durch der Aufruf von zwei verschiedene Konstruktoren erzeugt werden. Der erste Konstruktor ist der Standardkonstruktor, dieser wird für die Auflistung von die zur Verfügung stehende Ports und dessen Baudraten verwendet. Der zweite Konstruktor bekommt als Parameter ein String mit dem Namen des Ports. Dieser Port wird geöffnet und die Eigenschaften werden in lokale Variablen geladen und für zukünftige Operationen in den Tests gespeichert.

5.5 Klasse PortCommunications

In der Klasse *PortCommunications* werden die Lese- und Schreiboperation ausgeführt. Die Klasse besteht aus zwei Konstruktoren, drei Methoden und ein Destruktor. Obwohl die Klasse nur aus drei Methoden besteht, diese ist die Klasse die tatsächliche Systemaufrufe für Senden und Empfangen von Daten befasst.

Die erste Methode in der Klasse speichert ein Handle von ein offenen COM Port in einer lokalen Variable. Durch dieses Handle weist die Klasse aus welchem Port die Lese- oder Schreibmethoden ihre Zugriffe ausüben sollen. Dieses Handle kann auch durch den Aufruf eines personalisiertes Konstruktor gesetzt werden, dass als Parameter an den Konstruktor übergeben wird.

Das Schreiben zu einem COM Port ist sehr ähnlich wie das Lesen. Beide Operationen werden in einer Schleife wiederholt bis alles Bytes aus dem Puffer gelesen worden sind. Der ganze Programmcode zu diesen Methoden kann in Anhang A.3 und A.4 gelesen werden. Im folgenden werden nur Ausblicke der beiden Vorgänge erklärt.

5.5.1 Schreiben

Die Methode *writeData* erhält als Parametern der zu übertragene Text und die Länge des Textes. Die Länge des Textes ist für den Schreibvorgang nötig, damit die Schreibmethode ermitteln kann, wenn alle Bytes geschrieben worden sind. Bevor der Schreibvorgang beginnt, muss eine *OVERLAPPED* Struktur deklariert werden, zuständig für das asynchrone Schreiben. In der Struktur wird die Variable *hEvent* initialisiert, in dem ein neues Event deklariert wird.

Der Schreibvorgang wird in einer Schleife, die im Fehlerfall maximal fünf mal wiederholt wird, ausgeführt. In der Schleife wird die Methode *WriteFile* folgendermaßen aufgerufen:

```
1  WriteFile(hCom, lpBuf, dwSize, &dwWritten, &osWrite);
```

Als Parameter benötigt die Methode das Handle aus dem Port wo gelesen werden soll. Somit kann die Methode in den verschiedene Testmodi der Sender und Empfänger unterscheiden. Als zweiter Parameter wird der zu schreibende Text und als dritter Parameter die Länge des Textes übergeben. Der vierte und fünfte Parameter sind die Adresse der jeweiligen Variablen. In der Variable *dwWritten* werden die Anzahl der geschriebene Bytes geschrieben und die*osWrite* Variable ist die am Anfang des Schreibvorgang erstellte *OVERLAPPED* Struktur. War der Schreibversuch Fehlerhaft, wird es dem Benutzer ermittelt, sonst wird die Methode *WaitForSingleObject* aufgerufen.

```
1  WaitForSingleObject(osWrite.hEvent, INFINITE);
```

Die Methode *WaitForSingleObject* bekommt als Parametern das am Anfang kreierte Event in der *OVERLAPPED* Struktur und eine Ablaufzeit. Die Methode wartet so lange bis das Event signalisiert wird. Im diesem Fall ist das Timeout auf *INFINITE* gesetzt, damit das Schreiben vollständig beendet wird. Dieser Wert habe ich als Empfehlung aus der MSDN Online Bibliothek[Mic95] entnommen. Wenn das Event signalisiert wird, wird die Methode *GetOverlappedResult* auf dieser Weise aufgerufen:

```
1  GetOverlappedResult(hCom, &osWrite, &dwWritten, FALSE);
```

Wieder bekommt die Methode das Handle auf das COM Port, die Adresse der *OVERLAPPED* Struktur und die Adresse der Variable *dwWritten*. Das letzte Parameter spezifiziert ob auf das

Event gewartet werden soll. Da aber in *WaitForSingleObject* schon gewartet wird, bis das Event signalisiert wird, muss bei dem Aufruf von *GetOverlappedResult* nicht nochmal gewartet werden, deswegen wird der letzte Parameter mit *FALSE* belegt.

Wenn die Methode *GetOverlappedResult FALSE* zurückliefert, wird der Benutzer den letzten Fehler benachrichtigt. Ist der Rückgabewert der Methode *TRUE*, bedeutet es, dass der Schreibzugriff beendet worden ist. Um zu Erfahren ob der Zugriff erfolgreich war, werden die Variablen mit den zu schreibende Bytes und die geschriebene Bytes verglichen. Wenn diese Werte sich von einander unterscheiden, dann ist die Zeit für den Schreibzugriff abgelaufen (time out). Sind beide Werte gleich, dann war die der Vorgang erfolgreich und die Schreibmethode wird beendet.

5.5.2 Lesen

Die Lesemethode erhält die gleiche Parameter wie die Schreibmethode. Sie braucht auch eine *OVERLAPPED* Struktur um das Event abfragen zu können. Das Grundgerüst für die Lesemethode ist der gleiche wie die von der Schreibmethode. Beide Vorgänge werden in einer Schleife wiederholt, bis der Vorgang beendet wird oder ein Fehler gemeldet wird. Der Unterschied ist das beim Lesen die Schleife nicht fünf mal wiederholt wird, sonder die Schleife ist unendlich. Obwohl das programmiertechnisch nicht empfehlenswert ist, muss die Leseoperation so oft wiederholt werden, bis alle Bytes angekommen sind. Der Empfänger muss diese Daten durch ein Pollingverfahren aus dem Puffer lesen. Die Methode zum Lesen lautet:

```
1   ReadFile(hCom, ourBuf, dwSize, NULL, &osReader)
```

Die *ReadFile* Methode bekommt die gleiche Parameter wie die Schreibmethode. Der Unterschied ist, dass durch die asynchrone Kommunikation die Lesemethode nicht die Anzahl der gelesene Bytes speichert, weil diese verfälscht sein können. Wird der Lesezugriff Fehlerhaft ausgeführt, wird der Benutzer benachrichtigt, ansonsten werden die Methoden *WaitForSingleObject* und *GetOverlappedResult* aufgerufen. Die *WaitForSingleObject* Methode bekommt als Ablaufzeit ein vordefiniertes Wert von 500 Millisekunden [2]. Die Methode *GetOverlappedResult* bekommt die gleiche Parameter wie im Schreibzugriff.

Um zu ermitteln ob der Lesevorgang erfolgreich abgeschlossen worden ist, müssen die Anzahl der gelesene Bytes mit der Anzahl der zu lesende Bytes verglichen werden. Da die Lesemethode mehrmals aufgerufen wird, werden bei jedem Vorgang nicht die ganze Anzahl an Bytes gelesen. Die gelesene Bytes müssen in einem Puffer gespeichert werden, ohne die schon gelesene Bytes zu überschreiben. Um dieses Problem zu lösen, wird ein Zeiger deklariert, der auf dem Anfang des Puffers im ersten Lesezugriff zeigt. Mit Hilfe von Zeigerarithmetik, wird der Zeiger immer um die gelesene Anzahl von Bytes addiert. Um zu ermitteln wann alle Bytes gelesen worden sind, wird einer Variable die Anzahl an gelesenen Bytes subtrahiert. Wenn diese Variable gleich Null ist, dann worden alle Bytes gelesen und der Lesevorgang wird erfolgreich abgebrochen. Um ein Fehler im Lesevorgang zu erkennen, darf eine Leseoperation maximal fünf mal in ein time-out geraten, dann wird der Vorgang abgebrochen und der Benutzer wird benachrichtigt.

[2] [Mic95]

5.6 Klasse TestManager

Die Klasse *TestManager* bereitet die Testeinstellungen und die Testlogistik vor. In der Klasse werden die letzten Parameter überprüft, besonders für den Wobbelntest, und verschiedene Variablen auf seine Standardeinstellung gesetzt, damit der Testvorgang fehlerfrei beginnen kann.

5.6.1 Aufbau

Die Klasse besteht aus vier Methoden, wo drei die Klasse *FixedTest* (Kapitel 5.7) aufrufen und die Test beginnen. Die andere Methode bereitet die Eigenschaften der Klasse *TestManager* vor um die drei anderen Methoden aufrufen. Die Methode *startManager* erzeugt ein Objekt der Klasse *Logger* (Kapitel 5.11) wenn die Log-Option gesetzt ist. Danach werden die lokalen Variablen auf seine Standardwerte gesetzt und je nach Testeinstellung wird die richtige Testmethode aufgerufen.

Es gibt drei Testmodi (Fixed, Wobble und Automatic), und jedes Testmodi hat verschiedene Testeinstellungen. Im Fall von *Fixed*, werden alle mögliche Test und Port Einstellungen berücksichtigt. Im *Wobble* Test wird zwischen eine Unter- und Obergrenze der Baudrate und / oder der Parität gewobbelt. Der *Automatic* Test ist ein vordefiniertes Testfall. Hier muss der Benutzer nur der zu testende COM Port auswählen und welche Transfermöglichkeit er sich wünscht.

Im Fall von einem *Fixed* Test, wird in der Methode *startManager* die Methode *startFixedTest* aufgerufen. In der Methode *startFixedTest* wird als erstes ein Objekt der Klasse *FixedTest* erzeugt mit der aktuellen Teststruktur als Übergabeparameter. Ein Schleifenzähler wird initialisiert, dieser repräsentiert der *Repeater* aus der GUI oder Konfigurationsdatei. Wenn die Log-Option gesetzt ist, wird die Testeinstellungen in einer Testkonfigurationsdatei gespeichert. Als nächstes in einer „*do while*“ Schleife werden die Methoden des Objektes *FixedTest* aufgerufen, abhängig von die Transfereinstellung (Shorted, Double, Master, Slave). Ist der Transfermodus *Master* gesetzt, werden drei Sekunden gewartet, damit der Benutzer Zeit hat, das *Slave* Modul in einem anderen System zu starten. Die Schleife wird wiederholt so lange der Schleifenzähler nicht seine Grenze erreicht oder der Benutzer den Test abbricht (durch drücken der Stopp-Schaltfläche in der Benutzeroberfläche oder der „ESC“ Taste in der Kommandozeile). Ist die Option „stop on first error“ gesetzt und ein Fehler in der Kommunikation wird erkannt, so wird auch die Schleife unterbrochen und der Errorcode zurückgegeben.

Hat der Benutzer ein *Wobble* Test eingestellt, ist der Ablauf der Testvorbereitungen und Abbruchkriterien gleich wie in einem *FixedTest*. Der einzige Unterschied zu einem *Fixed* Test ist, dass zu einer Verschachtlung von Schleifen kommt. In diesem Test wird zwischen eine minimale und maximale Baudrate gewobbelt. Dafür werden alle Standard einstellbare Baudraten zwischen den Grenzen betrachten. Möchte der Benutzer auch durch die drei verschiedene Paritätseinstellungen (gerade, ungerade oder keine Parität) wobbeln, gibt es eine zweite Schleife, wo diese Eigenschaft wechseln eingestellt wird. So wird im ....breiten - längsten..... Testfall für eine Baudrate die drei verschiedene Paritätseinstellungen eingestellt und für jede Paritätseinstellung der Test so oft wiederholt, wie im *Repeater* angegeben.

Der *Automatic* Test besitzt eine vordefinierte Testeinstellung. Für den angegebene COM Port wird eine feste Testkonfiguration gesetzt und diese so lange wiederholt, bis der Benutzer mit einem Abbruchkriterium den Test beendet. Für weitere Informationen über die Testeinstellungen und Testeigenschaften bitte siehe Kapitel 7.

5.7 Klasse FixedTest

Die Klasse *FixedTest* ist die Klasse die Kommunikation zwischen ein oder zwei COM Ports organisiert. Alle Testeinstellungen und organisatorisches wurde schon eingestellt. Die Klasse besitzt Objekte der Klassen *PortCommunications* und *Com*, um COM Ports zu verwalten und zwischen Ports zu kommunizieren.

5.7.1 Ablauf eines Tests

Für ein Test muss als erstes die ausgewählte Schnittstelle mit den ausgewählten Einstellungen eingestellt werden, unabhängig davon was für ein Testfall betrachtet wird. Die *Master* und *Slave* Tests sind komplexer als ein *Shorted* oder *Double* Test. Als erstes muss der ausgewählte Port geöffnet werden und die Port Eigenschaften geladen werden. Mit dem Aufruf der Methode *setPortSettings* werden die angegebene Schnittstelleneinstellungen für den Test eingestellt. Im Fall von einem *Shorted* Test muss nur eine Schnittstelle initialisiert werden. Ist es ein *Double* Test müssen die ausgewählte Ports mit den gleichen Parametern initialisiert werden.

Ist dieser Schritt erfolgreich abgeschlossen worden, dann kann der Informationsaustausch beginnen. In einem *Shorted* Test sendet und empfängt die gleiche Schnittstelle die Information. Diese wird verglichen um Fehler bei der Übertragung zu ermitteln. Ist es ein *Double* Test, gibt es getrennte Sender und Empfänger. Da aber beide Schnittstellen sich im gleichen System befinden, kann die gesendete und empfangene Information auch verglichen werden. Das Ergebnis des Vergleichs wird in einem Vektor geschrieben, der am Ende der Übertragung als Ergebnis ausgegeben wird. Wurde die Option „stop on first error“ gesetzt und ein Unterschied wurde im Vergleich erkannt, dann wird der Testvorgang abgebrochen. Wenn das Ende der Übertragungsinformation (eine Zeile, eine Datei oder der vordefinierte Text) erreicht wird, werden die geöffnete Schnittstellen geschlossen und das Testergebnis dem Benutzer gemeldet. Dieser Vorgang wird so oft wiederholt, wir es im *Repeater* angegeben worden ist.

Zuständig für die Kommunikation einer Schnittstelle ist die Klasse *PortCommunications*(siehe Kapitel 5.5, für den Zugriff auf die Lese- und Schreibmethoden gibt es jeweils einer „Wrapper“ Methode, die unter Master und Slave Schnittstellen unterscheidet. So bleibt die Testlogik getrennt von der Kommunikationslogik.

5.7.2 Master und Slave

Der Grundgerüst für ein *Master* und *Slave* Testvorgang ähnelt sich zu einem *Shorted* Test. Der wesentliche Unterschied liegt an das Problem, das beide Schnittstellen sich nicht kennen und müssen synchronisiert werden. Als erster werden beide Schnittstellen für ein Test vorbereitet, das hießt, die COM Ports werden geöffnet, aber anstatt die Porteigenschaften auf die Testeinstellungen zu setzen, werden hier die Default-Einstellungen benutzt. Der Slave kennt die Testkonfiguration nicht, er kennt nur wie oft ein Test wiederholt wird und welcher Transfermodus er hat.

Die Synchronisation der beiden COM Ports geschieht indem der Master ein Fluchtzeichen schickt und der Slave antwortet. Der Master schickt ein „ESC“ (ASCII 0x1B) und wartet auf die Antwort vom Slave, was in Form von einem „ACK“ (ASCII 0x06) gesendet wird. Wenn beide COM Ports synchron sind, schick der Master den Slave ein „Kopfzeile“. In dieser Kopfzeile steht die Nummer

der zu übertragende Zeile, so wie die Anzahl der zu Übertragene Bytes, jeweils mit immer einer Breite von drei Zeichen. Die Kopfzeile hat folgendes Format:

<Zeilennummer-Anzahl_An_Bytes>
<001-010>

Die Kopfzeile ist nötig, damit der Slave wissen kann, wie viele Zeichen an Übertragungstext, bei der nächsten Übertragung, ankommen werden, damit die richtig Anzahl an Bytes aus den Puffer gelesen werden. Nachdem die Kopfzeile empfange worden ist, und der Slave sein Inhalt interpretieren kann, wird ein „ACK“ gesendet, damit der Master wissen kann, dass die Kopfzeile erfolgreich übertragen worden ist. Wenn der Master das „ACK“ aus seinen Puffer liest, schickt er die erste Information an das Slave. Diese Information ist die gewünschte Testeinstellung.

baudrate;parität;stopbits;datenbits;protokoll
9600;1;2;8;2

Der Slave liest die ankommende Information und schickt ein „ACK“ zurück. Danach wird die Information auf valide Werte geprüft und die Einstellungen werden gespeichert. In diesem Moment sind beide Schnittstellen synchron und bereit den Testvorgang mit den gewünschten Testeinstellungen zu starten. In beiden Schnittstellen werden die gewünschte Einstellungen gesetzt, und die Schnittstellen versuchen sich erneut zu synchronisieren. Ist die Synchronisation erfolgreich, schickt der Master erneut eine Kopfzeile an den Slave und danach der Übertragungstext. Der Slave antwortet auf die Kopfzeile bei erfolgreichen Empfang mit einem „ACK“ und bei dem Empfang von den Übertragungstext wird dieser zurück geschickt. Nachdem der Master der Übertragungstext versendet, wartet er auf die Antwort des Slaves in Form von dem versendeten Text. Der Master liest der Empfangene Text aus seinem Puffer und vergleicht es, mit dem versendeten Text. Werden keine Fehler festgestellt, wird der Vorgang wiederholt bis der Ende der Übertragung. Um diesem Vorgang ergänzen zu erklären, bitte siehe Anhang A.6 für ein Ablaufdiagramm dieses Testmodus.

5.8 Klasse IniFileHandler

Die Klasse *IniFileHandler* behandelt das lesen und speichern von Testeinstellungen in einer Datei, eine so genannte Testkonfigurationsdatei. Das Format dieser Datei basiert sich auf die von Windows vorgegebene Initialisierungsdateien (.ini, INI-Datei). Mit Hilfe der Methoden *WritePrivateProfileString*[3] und *GetPrivateProfileString*[4] können jeweils die Testparameter geschrieben und gelesen werden. Das Format einer INI-Datei ist folgendermaßen aufgebaut:

[Sektion]
;Kommentar
Variable=Wert

In Kapitel 7 wird genauer die Parameter und Werte für eine Datei erläutert. Die Klasse hat wird in drei arten von Methoden getrennt, diese sind Lese-, Schreib- und Übersetzermethoden.

5.8.1 Übersetzer

Die Übersetzermethoden bekommen ein Parameter der entweder geschrieben oder im Programm gespeichert wird. In beiden Fällen muss der Parameter auf gültigkeit geprüft werden. Wird der Parameter in einer INI-Datei geschrieben, muss er in manchen Fällen, wie zum Beispiel bei der Baudrate, zuerst von einer Zahl auf einen String gewandelt werden. Wird die Baudrate gelesen, muss dieser String auf eine Zahl umgewandelt werden, und geprüft, ob der gelesene/umgewandelte Wert ein gültiger Wert ist. Dieser Vorgang gilt für alle anderen Parametern auch.

5.8.2 Schreiben

Durch der Aufruf der Methode *WritePrivateProfileString* mit folgender Parameter wird die Baudrate in eine Datei gespeichert:

```
1 WritePrivateProfileString(ComPort,"Baudrate","9600", Dateipfad);
```

Die Variable *Dateipfad* spezifiziert die Zieldatei wo der Schlüssel/Variable *Baudrate* geschrieben werden soll. Der Wert für den Eintrag ist *9600*. Der Eingabeparameter *ComPort* ist die Bezeichnung für ein COM Port und gibt an in welcher Sektion die Variable *Baudrate* geschrieben werden soll. Der Eintrag für den Aufruf dieser Methode lautet:

[COM2]
;Das ist die Baudrate
Baudrate=9600

Auf dieser Weise werden alle Testeigenschaften in einer INI-Datei gespeichert. Durch die Trennung mit Sektionen werden die Parameter für verschiedene COM Ports beim einem neuen Eintrag nicht überschrieben.

[3] http://msdn.microsoft.com/en-us/library/windows/desktop/ms725501(v=vs.85).aspx; 22.09.2013

[4] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724353(v=vs.85).aspx; 22.09.2013

5.8.3 Lesen

Will der Benutzer Werte, beziehungsweise die Baudrate, aus einer INI-Datei lesen, muss er die Methode *GetPrivateProfileString* mit folgenden Parametern aufrufen:

```
1 GetPrivateProfileString(ComPort, "Baudrate", NULL,
2                         Puffer, GrößeDesPuffers, Dateipfad);
```

Die bekannte Parameter sind die gleichen wie bei der Schreibmethode. *Puffer* ist eine Variable in der der gelesene Wert geschrieben wird, und *GrößeDesBuffers* gibt die Größe des Puffers an. So wird der String *9600* in der Variable *Puffer* geschrieben. Durch Aufruf der entsprechende Übersetzermethode, wird die gelesene Baudrate in einer Zahl umgewandelt und im Programm gespeichert.

5.9 Klasse Tools

Die Klasse *Tools* ist eine Sammlung von verschiedene Hilfsmethoden die in den anderen Klassen aufgerufen werden.

5.9.1 String Methoden

Unter die String Methoden ist zu verstehen, die Methoden die Zeichenketten bearbeiten. Entlang der Logik in der anderen Klassen ist immer wieder nötig Variablen in Strings umzuwandeln. Die Methode *convertToString* konvertiert ganze Zahlen, Multibyte Zeichenketten und Zeiger auf Zeichenketten in einem String. Der Eingabeparameter der Funktion wird in eine *stringstream* geschrieben und dieser Stream wird als String zurückgegeben.

Eine weitere Methode heißt *printTime*, diese Methode fragt die Uhrzeit im System ab, und schreibt diese in einem String, der die aufrufende Funktion zurück gibt. Die Methode *delSpaces* löscht alle Leerzeichen in das an die Methode übergebene String.

```
1  s.erase(remove_if(s.begin(), s.end(), isspace),s.end());
```

Der Löschvorgang wird mit Hilfe von der String-Klassenmethode *erase* und des Templateclass *remove_if* durchgeführt. Die Anfang- und Enditeratoren des Strings werden an das Templaclass gegeben so wie der Zeichenart was gelöscht werden soll. Die Zeichenart sind alle Leerzeichenmöglichkeiten (Tabs, CarriageReturn, NewLine, etc.), diese werden durch die Funktion *isspace* abgefragt und von der *erase* Methode gelöscht.

Die letzte Methode teilt die durch Leerzeichen übergebene Zeichenkette in einzelnen Strings und schreibt diese in einem Vektor vom Typ String. Diese Methode wird angewendet um die, über die Kommandozeile eingegebene, Parameter zu erkennen und einzeln auswerten zu können. Die letzte Stringmethode wird für die Übertragungstexte benutzt. Wenn ein Übertragungstext so genannte Fluchtzeichen beinhalten soll, werden diese als Hexadezimalwerte eingegeben und in dieser Methode durch ein Zeichen ersetzt.

5.9.2 Weitere Methoden

Eine der anderen Methoden in dieser Klasse ist die *printErrorVector* Methode. Diese Methode gibt die Ergebnisse der Tests aus, jeweils über die GUI als Fenster oder in der Logdatei. Die Methode *errorCodeParser* bekommt als Eingabeparameter ein Exitcode eines Funktionsaufrufes und gibt die Erklärung dieses Exitcodes als String zurück. Die *wait* Methode bekommt eine Zahl als Eingabeparameter. Diese Zahl bestimmt wie oft zehn Millisekunden gewartet werden sollen. Die Methode wird bei der Synchronisierung von ein Master und ein Slave benutzt. Die letzte Methode in der Klasse *Tools* zeigt dem Benutzer ein Hilfetext wenn falsche Eingabeparameter in der Kommandozeile gegeben worden sind.

5.10 Klasse TransferFileHandler

Die Klasse *TransferFileHandler* behandelt die Dateien die in einer Übertragung gesendet werden sollen. Die Klasse öffnet, liest und schließt die Datei.

5.10.1 Aufbau

Die Klasse öffnet mittel der Methode *openFile* eine Datei die als Eingabeparameter angegeben worden ist. Eine Variable vom Typ *ifstream* (input file stream) wird deklariert. durch die Methode *open* wird die Pfad der Datei angegeben, und die Methode versucht diese Datei zu öffnen. Falls die Datei nicht geöffnet werden konnte, wird der Benutzer benachrichtigt. Ist die Datei erfolgreich geöffnet worden, dann wird jede Zeile in der Datei gelesen in einem Vektor gespeichert bis das Ende der Datei erreicht wird. Danach wird die Datei geschlossen.

5.11 Klasse Logger

Die Klasse *Logger* erzeugt auf Wunsch des Benutzers eine Logdatei, wo alle Testereignisse geschrieben werden. Wird ein Objekt von der Klasse *Logger* erzeugt, wird als erstes eine Datei im „%temp%“ Ordner angelegt und der Zielpuffer des Objekts *clog* (output stream) auf die Logdatei umgeleitet. Bei Schreibgeschützten Medien, wie im Fall von Windows PE , darf diese Option nicht gewählt werden. Für ein Beispiel einer Logdatei, bitte siehe Anhang A.5.

5.12 Header Constants

In die Headerdatei *Constants* werden alle Konstanten für das Programm deklariert. Unter anderem sind hier die Werte für:

- Programmversion
- ID der GUI-Elemente
- Errorcodes
- die Baudraten
- TestStrukt (siehe 5.3)

deklariert.

Kapitel 6

Realisierung

Selbstverständlich sollte Realisierung auch in Ihrer Arbeit abgehandelt werden. Aus der Sicht der Softwaretechnik stellt sie aber nur der kronende Abschluß der Arbeit dar. Hier können die realisierungsspezifischen Probleme (z.B. mit der Implementierungssprache) und das Testkonzept inkl. der protokollierten Testergebnisse dargestellt werden. Wichtige, komplexe oder besonders interessante Systemteile können auch im Programmcode dargestellt werden. Bitte aber Hinweis 12 (Programm-Listings) beachten!

READ TEST

Kapitel 7

Bedienungsanleitung

Das Test Programm kann über die Kommandozeile oder direkt über die Benutzeroberfläche bedient werden. In beiden Fällen muss der Benutzer folgende Parameter angeben, je nach Testmodus, um erfolgreich ein Test zu starten:

Parameter		**GUI Element**
COM Port	:	Main Port
Baudrate	:	Baud rate
Test Modus	:	Test Mode
Übertragungsmodus	:	Transfer
Slave Port	:	Slave Port
Maximale Baudrate	:	MAX baud rate
Parität	:	Parity
Übertragungsprotokoll	:	Protocol
Stopbits	:	Stopbits
Datenbits	:	Databits
Übertrangsdatei	:	Load file to be transfered
Übertragungstext	:	Send text
Logger	:	Logger
Wiederholzähler	:	Repeater
Beim ersten Fehler anhalten	:	Stop on 1. error
Start	:	Start
Schließen	:	Close
Speichern	:	Save
Testdatei laden	:	Load test file
Hilfe	:	Help
Stop	:	Stop

7.1 Bedienung über die GUI

Mit der GUI kann der Benutzer ein Test starten, mit bestimmten Parametern oder durch das Laden einer Testdatei. Der Benutzer kann auch eine Testdatei erstellen, in dem er die Parameter in der GUI einstellt und diese speichert.

7.2 Bedienung über die Kommandozeile

Über die Kommandozeile kann der Benutzer eine Testdatei und das zu testende Port angeben. Wenn keine Parameter angegeben worden sind, wird die GUI gestartet.

7.3 Testmodus

7.3.1 Automatic

Der Automatic Test ist vordefiniert. Für diesen Test muss der Benutzer nur den Port angeben, der er teste will, und welches Übertragungsmodus. Der Test wird immer mit folgenden Parameter gestartet.

Parameter		**Wert**
Main Port	:	Von Benutzer wählbar
Baud rate	:	9600
Test Mode	:	Automatic
Transfer	:	Von Benutzer wählbar
Slave Port	:	Wenn nötig, von Benutzer wählbar
Parity	:	Odd
Protocol	:	Hardware
Stopbits	:	1
Databits	:	8
Send text	:	Default, fest kodiert
Logger	:	Ja
Repeater	:	Unendlich

7.3.2 Wobble

In einem Wobble Test kann der Benutzer alle Parameter einstellen. Die wichtigsten Parametern sind die minimale und maximale Baudrate. Diese Werte geben an, mit welchen Baudraten getestet wird.

7.3.3 Fixed

Im Fixed Test muss der Benutzer alle Parameter einstellen. Im diesem Test werden sich die Parameter nicht ändern. Der Benutzer kann nach Fehler mit spezifischen Einstellungen testen.

7.4 Übertragungsmodus

Jedes Testmodus kann in vier Übertragungsmodus eingestellt werden, die den Test ablauft bestimmen

7.4.1 Shorted

Im Shorted Modus wird an der angegebene Schnittstelle eine Kurzschlussstecker angeschlossen. Hier ist der Port Master und Slave gleichseitig. Der Port schickt und empfängt die Daten sofort.

7.4.2 Double

Hier werden zwei Ports in einem System getestet. Mittels eines Null-Modem-Kabel wird Port1 and Port2 angeschlossen. Port1 ist der Master und Port2 ist der Slave. So sendet der Master Informationen und der Slave empfängt sie.

7.4.3 Master

Der Master Modus involviert zwei verschiedene Systeme. Im System1 schickt der Master Daten und wartet auf eine Antwort. Der Master kennt sein Slave nicht. System1 und System2 werden durch ein Null-Modem-Kabel verwunden. Der Master macht die Fehlerauswertung, so lange keine Lese- oder Schreibfehler im Slave geschehen sind.

7.4.4 Slave

Der Slave im System2 wartet auf den Empfang von Daten aus System1. Der Slave kennt sein Master nicht, er empfangt Daten und schickt die gleiche Daten zurück, damit der Master die Fehlerauswertung machen kann.

7.5 Schnittstelleneigenschaften

Baudrate Die Baudrate mit der getestet werden soll.

Parity Hier kann der Benutzer zwischen gerade, ungerade oder keine Parität auswählen.

Protocol Der Benutzer ist in der Lage das Übertragungsprotokoll zu bestimmen. Zur Auswahl steht Xon/Xoff, Hardware oder kein Protokoll.

Stopbits Mit diesem Parameter werden die Stopbits eingestellt, entweder 1 oder 2 Stopbits.

Databits Hier wird bestimmt, wie viele Bits ein Zeichen hat, 7 oder 8 Bits pro Zeichen.

7.6 Sendedaten

Durch klicken des Buttons „Load file to send“ kann der Benutzer eine Datei auswählen, diese wird dann beim testen geöffnet und Zeilenweise verschickt. Will der Benutzer ein spezifischer Text senden, muss er dieser unter „Send text“ eingeben und „Load text to send“ klicken. Wenn keiner dieser beiden Möglichkeiten benutzt wird, wird ein im Programm fest kodierter Text übertragen.

7.7 Logger

Wenn die Checkbox ausgewählt ist, wie eine Textdatei im „%Temp%“ Verzeichnis angelegt und alle Meldung dort geloggt. Die Datei hat immer den Namen des Ports dass getestet wird.

7.8 Wiederholzähler

Der Wiederholzähler gibt an, wie oft ein Testschritt wiederholt werden soll. Unter Testschritt ist zu verstehen, jedes Schreibe- und Lesevorgang mit verschiedene einstellbare Parameter.

7.9 Beim ersten Fehler anhalten

Diese Option gilt für die Fehlersuche. Wenn diese Option gesetzt ist, haltet das Test Tool beim ersten erkannten Fehler. So kann der Tester nach Fehler suchen. Wenn diese Option nicht gesetzt ist, werden die Fehler am Ende des Testvorgangs gemeldet oder geloggt. So kann eine Fehlerstatistik erzeugt werden.

7.10 Buttons

Start Beginnt ein Test mit den eingegeben Parametern.

Close Beendet das Test Tool.

Save Speichert die ausgewählten Einstellungen in einer Testdatei.

Load test file Ladet eine vom Benutzer ausgewählte Testdatei.

Help Zeigt die Hilfe zum Programm.

Stop Stoppt der Testvorgang.

7.11 Testdatei

In einer Testdatei werden die Testkonfiguration gespeichert. Eine Testdatei kann manuell oder mit Hilfe der GUI erzeugt werden. Die Testdatei ist wie eine Datei aus der Registrierungsdatenbank (Windows Registry)aufgebaut. In „

“ wird der Port angegeben und darunter die jeweilige Testparametern. Für ein Beispiel bitte siehe Anhang.

7.12 Einfaches Beispiel

Will zum Beispiel ein Benutzer der Port COM1 testen. Dieser soll an COM2 die Wörter „Hallo Welt“ schicken. Als Baudrate ist 9600 vorgegeben, mit einer geraden Parität, Xon/Xoff Protokoll, sieben Datenbits und 2 Stopbits. Der Test soll fünfmal wiederholt werden und eine Log-Datei soll geschrieben werden.

Als erstes muss der Benutzer unter *Main Port* „COM1“ auswählen. Danach werden die mögliche Baudraten in *Baud rate* aufgelistet. Dort muss der Benutzer 9600 wählen. Da nur eine bestimmte Konfiguration gegeben ist, ohne Variablen, wird auf „Fixed“ unter *Test Mode* geklickt und als Übertragungsmodus „Double“ ausgewählt. Unter *Parity* wird „Even“, *Protocol* wird „XON/-XOFF“, *Stopbits* wird „2“ und *Databits* wird „7“ ausgewählt.

BILD

Unter *Send text* soll der Benutzer „Hallo Welt“ schreiben. Will der Benutzer Escape-Sequenzen schicken, müssen diese als Hexadezimal Werte angegeben werden(zum Beispiel \0a oder \0d). Danach wird auf *Load text to send* geklickt. Ein Pop-Up Fenster meldet, dass der Text geladen worden ist. Damit der Test fünfmal wiederholt wird, muss in *Repeater* eine fünf anstatt der eins geschrieben werden. Danach ist die GUI konfiguriert und kann auf *Start* geklickt werden.

BILD

Durch ein Pop-Up Fenster wird dem Benutzer gemeldet, wenn der Test fertig ist. Um eine Evaluation des Tests machen zu können muss die Log-Datei ausgewertet werden.

Kapitel 8

Zusammenfassung und Ausblick

Lehnen Sie sich zurück von Ihrem Terminal und versuchen ein wenig Abstand zu den vielen Detail-Problemen Ihrer Diplomarbeit zu gewinnen: Was war wirklich wichtig bei der Arbeit? Wie sieht das Ergebnis aus? Wie schätzen Sie das Ergebnis ein? Gab es Randbedingungen, Ereignisse, die die Arbeit wesentlich beeinflußt haben? Gibt es noch offene Probleme? Wie könnten diese vermutlich gelöst werden?

Durch die Verwendung der API versteht man Windows als Betriebssystem besser.

Master wartet 5 sekunden, bessere lösung?

Abbildungsverzeichnis

Anhang A

Anhang

A.1 Template Class

```
1  // Quelle:
2  // http://msdn.microsoft.com/en-us/library/windows/desktop/
3
4  template <class DERIVED_TYPE>
5  class BaseWindow
6  {
7  public:
8    //Windows default STATIC message handler
9    static LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg,
10                                       WPARAM wParam, LPARAM lParam)
11   {
12     DERIVED_TYPE *pThis = NULL;
13
14     if (uMsg == WM_NCCREATE)
15     {
16       //extract pointer
17       CREATESTRUCT* pCreate = (CREATESTRUCT*)lParam;
18       pThis = (DERIVED_TYPE*)pCreate->lpCreateParams;
19       SetWindowLongPtr(hwnd, GWLP_USERDATA, (LONG_PTR)pThis);
20
21       pThis->m_hwnd = hwnd;
22     }
23     else
24       pThis = (DERIVED_TYPE*)GetWindowLongPtr(hwnd, GWLP_USERDATA);
25
26     if (pThis)
27       return pThis->HandleMessage(uMsg, wParam, lParam);
28     else
29       return DefWindowProc(hwnd, uMsg, wParam, lParam);
30   }//WindowProc
31
32
33   BaseWindow() : m_hwnd(NULL) { }
34
35   BOOL Create(LPCSTR lpWindowName,
36         DWORD dwStyle,
37         DWORD dwExStyle = 0,
38         int x = WINDOW_X,
39         int y = WINDOW_Y,
```

```
40          int nWidth = WIN_WIDTH,
41          int nHeight = WIN_HEIGHT,
42          HWND hWndParent = 0,
43          HMENU hMenu = 0)
44      {
45      //Window properties
46      WNDCLASS wc = {0};
47
48      //wc.lpszClassName = L"Serial Port Tester";
49      wc.hbrBackground = GetSysColorBrush(COLOR_3DFACE);
50      wc.hCursor       = LoadCursor(0, IDC_ARROW);
51      wc.hIcon     = LoadIcon(NULL, IDI_APPLICATION);
52      wc.lpfnWndProc   = DERIVED_TYPE::WindowProc;
53      wc.hInstance     = GetModuleHandle(NULL);
54      wc.lpszClassName =  ClassName();
55
56      RegisterClass(&wc);
57
58      //main window creation and handle
59      m_hwnd = CreateWindowEx(
60        dwExStyle, ClassName(), lpWindowName, dwStyle, x, y,
61        nWidth, nHeight, hWndParent, hMenu, GetModuleHandle(NULL), this);
62
63          return (m_hwnd ? TRUE : FALSE);
64      }//create
65
66    HWND window() const { return m_hwnd;}
67
68 protected:
69
70      virtual LPCSTR  ClassName() const = 0;
71      virtual LRESULT HandleMessage(UINT uMsg,
72                      WPARAM wParam, LPARAM lParam) = 0;
73
74      HWND m_hwnd;
75 };
```

A.2 HandleMessage

```
1  LRESULT Window::HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam)
2  {
3      switch (uMsg)
4      {
5
6      case WM_CREATE:
7          {
8              //Create all GUI Elements
9
10             //example: Start button
11             _hwnd_Start = CreateWindowA("button", "Start",
12         WS_CHILD | WS_VISIBLE,
13         POS_X+20, POS_Y2 + 290, 70, 30, m_hwnd, (HMENU)ID_BT_START,
14         NULL, NULL);
15         }
16         break;
17
18     case WM_COMMAND:
19         {
20             //React to GUI Elements actions
21
22             //example: Start button
23             case ID_BT_START:
24
25         //hide the elements while testing
26         viewAllElements(FALSE);
27
28         //start a new thread
29         _t1 = thread(&Window::sendTestSettings, this);
30
31         //detach the thread so it can test and
32         //main thread waits for it to finish
33         //or waits for the user to press stop
34         _t1.detach();
35         }
36         break;
37
38     case WM_DESTROY:
39         PostQuitMessage(0);
40         break;
41
42   //call the default window procedure for the other messages
43     default:
44         return DefWindowProc(m_hwnd, uMsg, wParam, lParam);
45     }
46     return TRUE;
47 }
```

A.3 Readdata

```
1  bool PortCommunications::readData(char * lpBuf, DWORD dwSize)
2  {
3    int iCounter = 0;
4    int iErr;
5    DWORD dwRead;
6    DWORD dwRes;
7    char ourBuf[100] = {0};
8    int ourCount = dwSize;
9    char *ourPtr = lpBuf;
10
11   BOOL fWaitingOnRead = FALSE;
12   OVERLAPPED osReader = {0};
13
14   // Create the overlapped event. Must be closed before exiting
15   // to avoid a handle leak.
16   osReader.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
17
18   if (osReader.hEvent == NULL)
19   {
20     // Error creating overlapped event; abort.
21     clog<<"error creating overlapped event, abort"<<endl;
22     CloseHandle(osReader.hEvent);
23     return FALSE;
24   }
25
26   while(true)
27   {
28     //if not waiting on read operation, then read
29     if (!fWaitingOnRead)
30     {
31       // Issue read operation.
32       if (!ReadFile(hCom, ourBuf, dwSize, NULL, &osReader))
33       {
34         //if not true
35         iErr = GetLastError();
36         if (iErr != ERROR_IO_PENDING)     // read not delayed?
37         { // Error in communications; report it.
38           clog << "Error reading port. System Error: " << iErr << endl;
39           return FALSE;
40         }
41         else
42         {
43           clog << "\noperation not completed yet. buffer ->"
44                << ourBuf << endl;
45           fWaitingOnRead = TRUE;
46         }
47       }
48       else
49       {
50         // read completed immediately
51         clog << "read completed immediately" << endl;
52         CloseHandle(osReader.hEvent);
53         return TRUE;
54       }
55     }
```

```
56
57
58      if (osReader.hEvent == NULL)
59      {
60        clog << "Unexpected␣NULL␣object␣" << endl;
61        return FALSE;
62      }
63
64      dwRes = WaitForSingleObject(osReader.hEvent, WAIT_FOR_READ_OBJ);
65
66      switch(dwRes)
67      {
68        // Read completed. The state of the specified object is signaled
69        case WAIT_OBJECT_0:
70          if (!GetOverlappedResult(hCom, &osReader, &dwRead, FALSE))
71          {
72            // Error in communications; report it.
73            iErr = GetLastError();
74            clog << "Error␣reading␣port.␣System␣Error:␣" << iErr << endl;
75            CloseHandle(osReader.hEvent);
76            return FALSE;
77          }
78
79          clog << "GetOverlappedResult␣was␣ok" << endl;
80
81          if(dwRead == 0)
82          {
83            clog << "No␣data␣available␣to␣be␣read.␣Buffer␣empty" << endl;
84            CloseHandle(osReader.hEvent);
85            return FALSE;
86          }
87
88          //reset for next read, this was successful
89          iCounter = 0;
90
91          memcpy(ourPtr, ourBuf, dwRead);
92          ourPtr   += dwRead;
93          ourCount -= dwRead;
94
95          if(ourCount <= 0)
96          {
97            clog << "read␣operation␣completed" << endl;
98            CloseHandle(osReader.hEvent);
99            return TRUE;
100         }
101
102         //  Reset flag so that another read operation can be issued.
103         fWaitingOnRead = FALSE;
104         break;
105
106       case WAIT_TIMEOUT:
107         //the time out interval elapsed,
108         //and the objects state is nonsignaled
109
110         // Operation isn't complete yet. fWaitingOnRead flag isn't
111         // changed since I'll loop back around, and I don't want
112         // to issue another read until the first one finishes.
113         iCounter++;
```

```
114         if(iCounter == 5)
115         {
116           clog << "to␣many␣object␣timeouts.␣ABORT" << endl;
117           return FALSE;
118         }
119         clog << "operation␣isnt␣complete␣yet,␣carry␣on..."<<endl;
120
121         break;
122
123       default:
124         // Error in the WaitForSingleObject; abort.
125         // This indicates a problem with the OVERLAPPED structure's
126         // event handle.
127         clog << "Error␣while␣reading␣in␣the␣WaitForSingleObject,\n"
128           << "problem␣with␣the␣overlapped␣stucture␣handle" << endl;
129         iErr = GetLastError();
130         clog << "Unexpected␣Error␣WaitForSingleObject.␣System␣Error:␣"
131              << iErr << endl;
132         CloseHandle(osReader.hEvent);
133         return FALSE;
134     }//switch
135
136   }//while
137 }
```

A.4 Writedata

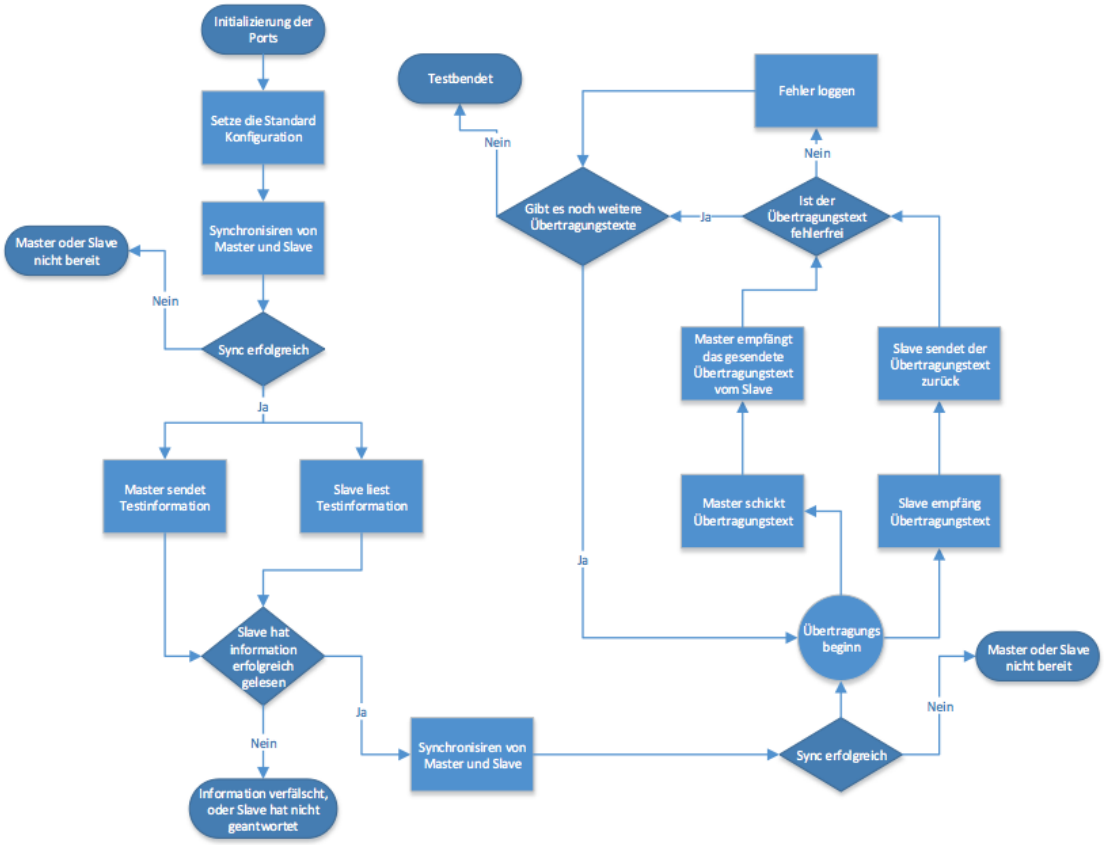
```
1
2 bool PortCommunications::writeData(const char * lpBuf, DWORD dwSize)
3 {
4   int iCounter = 0;
5   int iErr;
6   OVERLAPPED osWrite = {0};
7   DWORD dwWritten;
8   DWORD dwRes;
9   BOOL  fRes = FALSE;
10
11  // Create this write operation's OVERLAPPED structure hEvent.
12  osWrite.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
13  if (osWrite.hEvent == NULL)
14    // Error creating overlapped event handle.
15    return FALSE;
16
17  do
18  {
19    clog << "...Write␣attempt␣number:␣" << iCounter + 1 << endl;
20     // Issue write
21    if (!WriteFile(hCom, lpBuf, dwSize, &dwWritten, &osWrite))
22    {
23      iErr = GetLastError();
24      if (iErr != ERROR_IO_PENDING)
25      {
26        // WriteFile failed, but it isn't delayed. Report error.
27        fRes = FALSE;
28        iCounter++;
29        clog << "Error␣writing␣to␣port.␣System␣Error:␣"
30             << iErr << endl;
31      }
32      else
33      {
34        // Write is pending.
35        dwRes = WaitForSingleObject(osWrite.hEvent, INFINITE);
36        switch(dwRes)
37        {
38          // Overlapped event has been signaled.
39          case WAIT_OBJECT_0:
40            if(!GetOverlappedResult(hCom, &osWrite, &dwWritten, FALSE))
41            {
42              iErr = GetLastError();
43              fRes = FALSE;
44              iCounter++;
45              clog << "Error␣writing␣to␣port.␣System␣Error:␣"
46                   << iErr << endl;
47            }
48            else
49            {
50
51              if (dwWritten != dwSize)
52              {
53                // The write operation timed out.
54                clog << "The␣write␣operation␣timed␣out" << endl;
55                fRes = FALSE;
```

```
56                iCounter++;
57              }
58              else
59              {
60                //Write operation completed successfully
61                fRes = TRUE;
62              }
63            }
64            break;
65
66          default:
67            // An error has occurred in WaitForSingleObject.
68            iErr = GetLastError();
69            clog <<"Write error in WaitForSingleObject.\nThis usually "
70                 <<"indicates a problem with the overlapped "
71                 << "event handle." << endl;
72            clog << "Error writing to port. System Error: "
73                 << iErr << endl;
74            fRes = FALSE;
75            iCounter++;
76            break;
77        }//switch
78
79      }//else to error pending
80
81    }//writefile
82    else
83    {
84      // WriteFile completed immediately.
85      if (dwWritten != dwSize) {
86        // The write operation timed out.
87        clog << "The write operation timed out" << endl;
88        fRes = FALSE;
89        iCounter++;
90      }
91      else
92        fRes = TRUE;
93    }
94
95    if(fRes == TRUE)
96    {
97      CloseHandle(osWrite.hEvent);
98      return fRes;
99    }
100
101  }while(iCounter < 5);
102
103  CloseHandle(osWrite.hEvent);
104
105  return fRes;
106 }
```

A.5 Logdatei

hier kommt die logdatei für ein Kurzchlussstecker

A.6 Ablaufdiagramm eines Master - Slave Test



Literaturverzeichnis

[Cam90] Joe Campbell. *V 24 / RS-232 Kommunikation*. Sybex-Verlag GmbH, 1990.

[Cam94] Joe Campbell. *C Programmers Guide to Serial Communications*. SAMS Publishing, 1994.

[Lou10] Dirk Louis. *Visual C++ 2010*. Addison-Wesley, 2010.

[Mic95] Microsoft, `http://msdn.microsoft.com/en-us/library/ff802693.aspx`. *Serial Communications*, 1995.

[Pet98] Charles Petzold. *Programming Windows*. Microsoft Press, 1998.

[Wik13] Wikipedia, http://de.wikipedia.org/wiki/RS-232. *RS-232*, 2013.